FUNDAMENTOS DE POSTGRESQL

Este libro es una introducción al Gestor de base de datos PostgreSQL, su instalación, configuración, administración de usuarios y grupos, accesos y rutas, manejo del editor psql; así como aspectos de seguridad. Además, presenta la programación básica, las funciones y disparadores que es la forma de programar al servidor.

Este libro forma parte de la serie "Bases de Datos Relacionales"

JORGE DOMÍNGUEZ CHÁVEZ

Publicado por

IEASS, Editores

ORCID: 0000-0002-5018-3242

ISBN 9789806366077

Publicado por IEASS, Editores
ieass@blogspot.com
Venezuela, 2019

La portada y contenido de este libro fue editado y maquetado en TeXstudio 2.12.16

FUNDAMENTOS DE POSTGRESQL

JORGE DOMÍNGUEZ CHÁVEZ
UNIVERSIDAD POLITÉCNICA TERRITORIAL DE ARAGUA

Venezuela, 2019

Acrónimos

Palabra	Descripción
ACID	Atomicidad, Consistencia, aIslamiento, Durabilidad.
API	interfaz de programación de aplicaciones.
Bit	binary digit, dígito binario.
buffer	porción, de tamaño variable, de la MP utilizada en caché de páginas.
DLL	librerías de enlaces dinámicos para crear y definir nuevas bases de datos, campos e índices.
DML	librerías de gestión dinámicas para generar consultas para ordenar, filtrar y extraer datos de la base de datos.
GIS	sistema de información geográfica.
GUI	Abreviatura de Graphic User Interface o interfaz gráfica de usuario.
IDE	Entorno Integrado de Desarrollo, es una aplicación para desarrollar que va mucho más allá de lo que ofrece un simple editor.
MVCC	Acceso concurrente multiversión
POO	Programación Orientada a Objetos.
PL	lenguaje procedural o PL
SQL	lenguaje de consultas estructuradas.
SPI	interfaz de programación del servidor.
Puntero (pointer)	dirección de memoria
VCS	sistema de control de versiones o Version Control System.

Prólogo

PostgreSQL puede ser usado, modificado o distribuido para uso privado, comercial o Académico.

Hoy en día, casi la totalidad de los sistemas de información existentes consiste en el uso de Sistemas Gestores de Bases de Datos (SGBD) relacionales. En el mercado del software existen múltiples opciones propietarias bien conocidas para suplir esta necesidad, sin embargo en el presente también existen opciones libres que, por su escasa capacidad publicitaria, son notablemente desconocidas, al menos en nuestro entorno.

PostgreSQL es un sistema de gestión de bases de datos orientado a objetos y relacional (ORDBMS) que hace énfasis en la extensibilidad y conformidad con los estándares. Está liberado bajo la licencia free/open source PostgreSQL, similar a la licencia MIT. PostgreSQL está desarrollado por el Grupo de Desarrollo Global de PostgreSQL, el cual consiste en un grupo de voluntarios empleados y supervisados por empresas como Red Hat y EnterpriseDB.

Este texto pretende enseñar las características básicas del SGBD PostgreSQL como alternativa en prestaciones y robustez para cualquier tipo de demanda a los SGBD propietarios más extendidos.

PostgreSQL es un sistema manejador de bases de datos relacionales basado en postgres 4.2, desarrollado en el departamento Berkeley de Ciencias de la computación en la universidad de California. Postgres es Open Source descendiente del código original "Berkeley", soporta el estándar SQL y ofrece otros recursos tales como:

1. complex queries
2. foreign keys
3. triggers
4. views
5. transactional integrity
6. multiversion concurrency control
7. index methods
8. procedural languages

0.1. Objetivo general

Brindar al lector los conocimientos necesarios para que sea capaz de gestionar bases de datos utilizando el SGBD Postgresql.

0.2. Objetivos específicos

Al final del texto el lector estará en capacidad de:

1. Llevar a cabo el proceso de instalación y configuración de Postgresql 8.3.
2. Configurar el SGBD de acuerdo con los recursos del equipo.
3. Asegurar los datos mediante mecanismos de autenticación de clientes.
4. Realizar operaciones básicas de tablas y datos mediante el lenguaje PSQL.
5. Administrar gráficamente bases de datos.
6. Llevar a cabo rutinas de mantenimiento y monitoreo de las bases de datos.
7. Respaldar y recuperar los datos.
8. Utilizar algunas características avanzadas del SGBD como lo son dblik, herencia y particionamiento de tablas.
9. Hacer replicación de bases de datos.

0.3. Metodología

El objetivo principal de este manual es lograr transmitir al estudiante los conocimientos necesarios para que sea capaz de gestionar bases de datos utilizando el SGBD Postgresql. Para ello se hará uso de otros materiales adicionales, los cuales deben ser proveídos al estudiante, ya que este documento hará referencia a ellos.

0.4. Requisitos mínimos

Este curso está dirigido a personas con conocimientos básicos de: Computación, (especialmente en el sistema operativo GNU/Linux), Lenguaje SQL Estándar.

Índice general

Índice de figuras

Índice de cuadros

Introducción

En este capítulo se presenta una introducción a los principios de los sistemas de bases de datos.

Un sistema gestor de bases de datos (SGBD) es una colección de datos interrelacionados y un conjunto de programas para acceder a dichos datos. La colección de datos, denominada base de datos, contiene información relevante para una empresa. El objetivo principal de un SGBD es proporcionar una forma de almacenar y recuperar la información de una base de datos de manera que sea tanto práctica como eficiente.

Los sistemas de bases de datos se diseñan para gestionar grandes cantidades de información. La gestión de los datos implica tanto la definición de estructuras para almacenar la información como la provisión de mecanismos para la manipulación de la información. Además, los sistemas de bases de datos deben garantizar la fiabilidad de la información almacenada, a pesar de las caídas del sistema o de los intentos de acceso no autorizados. Si los datos van a ser compartidos entre diferentes usuarios, el sistema debe evitar posibles resultados anómalos.

Dado que la información es tan importante en la mayoría de las organizaciones, los científicos informáticos han desarrollado una gran cuerpo de conceptos y técnicas para la gestión de los datos. Estos conceptos y técnicas constituyen el objetivo central de este libro.

0.5. Aplicaciones de los sistemas de bases de datos

Las bases de datos se usan ampliamente. Algunas de sus aplicaciones representativas son:

1. Banca: para información de los clientes, cuentas, préstamos y transacciones bancarias.
2. Líneas aéreas: para reservas e información de horarios. Las líneas aéreas fueron de las primeras en usar las bases de datos de forma distribuida geográficamente.
3. Universidades: para información de los estudiantes, matrículas en las asignaturas y cursos.
4. Transacciones de tarjetas de crédito: para compras con tarjeta de crédito y la generación de los ex- tractos mensuales.
5. Telecomunicaciones: para guardar un registro de las llamadas realizadas, generar las facturas men- suales, mantener el saldo de las tarjetas telefónicas de prepago y para almacenar información sobre las redes de comunicaciones.
6. Finanzas: para almacenar información sobre compañías tenedoras, ventas y compras de productos financieros, como acciones y bonos; también para almacenar datos del mercado en tiempo real para permitir a los clientes la compraventa en línea y a la compañía la compraventa automática.

7. Ventas: para información de clientes, productos y compras.

8. Comercio en línea: para los datos de ventas ya mencionados y para el seguimiento de los pedidos Web, generación de listas de recomendaciones y mantenimiento de evaluaciones de productos en línea.

9. Producción: para la gestión de la cadena de proveedores y para el seguimiento de la producción de artículos en las factorías, inventarios en los almacenes y pedidos.

10. Recursos humanos: para información sobre los empleados, salarios, impuestos sobre los sueldos y prestaciones sociales, y para la generación de las nóminas.

0.6. Visión de los datos

Un sistema de bases de datos es una colección de datos interrelacionados y un conjunto de programas que permiten a los usuarios tener acceso a esos datos y modificarlos. Una de las principales finalidades de los sistemas de bases de datos es ofrecer a los usuarios una visión abstracta de los datos. Es decir, el sistema oculta ciertos detalles del modo en que se almacenan y mantienen los datos.

0.6.1. Abstracción de datos

Para que el sistema sea útil debe recuperar los datos eficientemente. La necesidad de eficiencia ha llevado a los diseñadores a usar estructuras de datos complejas para la representación de los datos en la base de datos. Dado que muchos de los usuarios de sistemas de bases de datos no tienen formación en informática, los desarrolladores ocultan esa complejidad a los usuarios mediante varios niveles de abstracción para simplificar la interacción de los usuarios con el sistema:

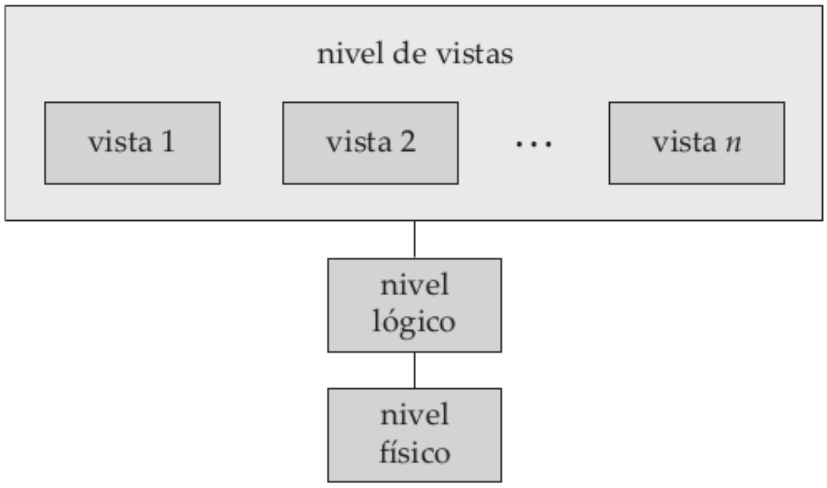


Figura 1: Los tres niveles de abstracción de datos.

- Nivel físico. El nivel más bajo de abstracción describe cómo se almacenan realmente los datos. El nivel físico describe en detalle las estructuras de datos complejas de bajo nivel.

- Nivel lógico. El nivel inmediatamente superior de abstracción describe qué datos se almacenan en la base de datos y qué relaciones existen entre esos datos. El nivel lógico, por tanto, describe toda la base de datos en términos de un número pequeño de estructuras relativamente simples. Aunque la implementación de esas estructuras simples en el nivel lógico puede involucrar estructuras complejas del nivel físico, los usuarios del nivel lógico no necesitan preocuparse de esta complejidad. Los administradores de bases de datos, que deben decidir la información que se guarda en la base de datos, usan el nivel de abstracción lógico.

- Nivel de vistas. El nivel más elevado de abstracción sólo describe parte de la base de datos. Aunque el nivel lógico usa estructuras más simples, queda algo de complejidad debido a la variedad de información almacenada en las grandes bases de datos. Muchos usuarios del sistema de bases de datos no necesitan toda esta información; en su lugar sólo necesitan tener acceso a una parte de la base de datos. El nivel de abstracción de vistas existe para simplificar su interacción con el sistema. El sistema proporciona muchas vistas para la misma base de datos.

La Figura 1 muestra la relación entre los tres niveles de abstracción.

Una analogía con el concepto de tipos de datos en lenguajes de programación puede clarificar la diferencia entre los niveles de abstracción. La mayoría de los lenguajes de programación de alto nivel soportan el concepto de tipo estructurado. Por ejemplo, en los lenguajes tipo Pascal se pueden declarar registros de la manera siguiente:

```
type cliente = record
  id_cliente : string;
  nombre_cliente : string;
  calle_cliente : string;
  ciudad_cliente : string;
end;
```

Este código define un nuevo tipo de registro denominado cliente con cuatro campos. Cada campo tiene un nombre y un tipo asociados. Una entidad bancaria puede tener varios tipos de estos registros, incluidos:

- cuenta, con los campos número_cuenta y saldo.

- empleado, con los campos nombre_empleado y sueldo.

En el nivel físico, los registros cliente, cuenta o empleado se pueden describir como bloques de posiciones consecutivas de almacenamiento (por ejemplo, palabras o bytes). El compilador oculta este nivel de detalle a los programadores. De manera parecida, el sistema de base de datos oculta muchos de los detalles de almacenamiento de los niveles inferiores a los programadores de bases de datos. Los administradores de bases de datos, por otro lado, pueden ser conscientes de ciertos detalles de la organización física de los datos.

En el nivel lógico cada registro de este tipo se describe mediante una definición de tipo, como en el fragmento de código anterior, y también se define la relación entre estos tipos de registros. Los programadores que usan un lenguaje de programación trabajan en este nivel de abstracción. De manera parecida, los administradores de bases de datos suelen trabajar en este nivel de abstracción.

Finalmente, en el nivel de vistas, los usuarios de computadoras ven un conjunto de programas de aplicación que ocultan los detalles de los tipos de datos. De manera parecida, en el nivel de vistas se definen varias vistas de la base de datos y los usuarios de la base de datos pueden verlas. Además de

ocultar los detalles del nivel lógico de la base de datos, las vistas también proporcionan un mecanismo de seguridad para evitar que los usuarios tengan acceso a ciertas partes de la base de datos. Por ejemplo, los cajeros de un banco sólo ven la parte de la base de datos que contiene información de las cuentas de los clientes; no pueden tener acceso a la información referente a los sueldos de los empleados.

0.6.2. Ejemplares y esquemas

Las bases de datos van cambiando a lo largo del tiempo conforme la información se inserta y se elimina. La colección de información almacenada en la base de datos en un momento dado se denomina ejemplar de la base de datos. El diseño general de la base de datos se denomina esquema de la base de datos. Los esquemas se modifican rara vez, si es que se modifican.

El concepto de esquemas y ejemplares de las bases de datos se puede comprender por analogía con los programas escritos en un lenguaje de programación. El esquema de la base de datos se corresponde con las declaraciones de las variables (junto con las definiciones de tipos asociadas) de los programas. Cada variable tiene un valor concreto en un instante dado. Los valores de las variables de un programa en un instante dado se corresponden con un ejemplar del esquema de la base de datos.

Los sistemas de bases de datos tiene varios esquemas divididos según los niveles de abstracción. El esquema físico describe el diseño de la base de datos en el nivel físico, mientras que el esquema lógico describe su diseño en el nivel lógico. Las bases de datos también pueden tener varios esquemas en el nivel de vistas, a veces denominados subesquemas, que describen diferentes vistas de la base de datos.

De éstos, el esquema lógico es con mucho el más importante en términos de su efecto sobre los programas de aplicación, ya que los programadores crean las aplicaciones usando el esquema lógico. El esquema físico está oculto bajo el esquema lógico, y generalmente puede modificarse fácilmente sin afectar a los programas de aplicación. Se dice que los programas de aplicación muestran independencia física respecto de los datos si no dependen del esquema físico y, por tanto, no hace falta volver a escribirlos si se modifica el esquema físico.

Se estudiarán los lenguajes para la descripción de los esquemas, después de introducir el concepto de modelos de datos en el apartado siguiente.

0.6.3. Modelos de datos

Bajo la estructura de las bases de datos se encuentra el modelo de datos: una colección de herramientas conceptuales para describir los datos, sus relaciones, su semántica y las restricciones de consistencia. Los modelos de datos ofrecen un modo de describir el diseño de las bases de datos en los niveles físico, lógico y de vistas.

En este texto se van a tratar varios modelos de datos diferentes. Los modelos de datos pueden clasificarse en cuatro categorías diferentes:

- Modelo relacional. El modelo relacional usa una colección de tablas para representar tanto los datos como sus relaciones. Cada tabla tiene varias columnas, y cada columna tiene un nombre único. El modelo relacional es un ejemplo de un modelo basado en registros. Los modelos basados en registros se denominan así porque la base de datos se estructura en registros de formato fijo de varios tipos. Cada tabla contiene registros de un tipo dado. Cada tipo de registro

define un número fijo de campos, o atributos. Las columnas de la tabla se corresponden con los atributos del tipo de registro. El modelo de datos relacional es el modelo de datos más ampliamente usado, y una gran mayoría de sistemas de bases de datos actuales se basan en el modelo relacional. El modelo relacional es tratado más adelante.

- El modelo entidad-relación. El modelo de datos entidad-relación (E-R) se basa en una percepción del mundo real que consiste en una colección de objetos básicos, denominados entidades, y de las relaciones entre ellos. Una entidad es una ”cosa” u ”objeto” del mundo real que es distinguible de otros objetos. El modelo entidad-relación se usa mucho en el diseño de bases de datos.

- Modelo de datos orientado a objetos. El modelo de datos orientado a objetos es otro modelo de datos que está recibiendo una atención creciente. El modelo orientado a objetos se puede con- siderar como una extensión del modelo E-R con los conceptos de la encapsulación, los métodos (funciones) y la identidad de los objetos.

- Modelo de datos semiestructurados. El modelo de datos semiestructurados permite la especificación de datos donde los elementos de datos individuales del mismo tipo pueden tener diferentes conjuntos de atributos. Esto lo diferencia de los modelos de datos mencionados anteriormente, en los que cada elemento de datos de un tipo particular debe tener el mismo conjunto de atributos. El lenguaje de marcas extensible (XML, eXtensible Markup Language) se emplea mucho para representar datos semiestructurados.

El modelo de datos de red y el modelo de datos jerárquico precedieron cronológicamente al relacional. Estos modelos estuvieron íntimamente ligados a la implementación subyacente y complicaban la tarea del modelado de datos. En consecuencia, se usan muy poco hoy en día, excepto en el código de bases de datos antiguas que sigue estando en servicio en algunos lugares. Se describen brevemente en los Apéndices A y B para los lectores interesados.

0.7. Lenguajes de bases de datos

Los sistemas de bases de datos proporcionan un lenguaje de definición de datos para especificar el esquema de la base de datos y un lenguaje de manipulación de datos para expresar las consultas y las modificaciones de la base de datos. En la práctica, los lenguajes de definición y manipulación de datos no son dos lenguajes diferentes; en cambio, simplemente forman parte de un único lenguaje de bases de datos, como puede ser el muy usado SQL .

0.7.1. Lenguaje de manipulación de datos

Un lenguaje de manipulación de datos (LMD) es un lenguaje que permite a los usuarios tener acceso a los datos organizados mediante el modelo de datos correspondiente o manipularlos. Los tipos de acceso son:

- La recuperación de la información almacenada en la base de datos.

- La inserción de información nueva en la base de datos.

- El borrado de la información de la base de datos.

- La modificación de la información almacenada en la base de datos.

Hay fundamentalmente dos tipos:

- Los LMD s procedimentales necesitan que el usuario especifique qué datos se necesitan y cómo obtener esos datos.
- Los LMD s declarativos (también conocidos como LMD no procedimentales) necesitan que el usuario especifique qué datos se necesitan sin que haga falta que especifique cómo obtener esos datos.

Los LMD s declarativos suelen resultar más fáciles de aprender y de usar que los procedimentales.

Sin embargo, como el usuario no tiene que especificar cómo conseguir los datos, el sistema de bases de datos tiene que determinar un medio eficiente de acceso a los datos.

Una consulta es una instrucción que solicita que se recupere información. La parte de los LMD implicada en la recuperación de información se denomina lenguaje de consultas. Aunque técnicamente sea incorrecto, resulta habitual usar las expresiones lenguaje de consultas y lenguaje de manipulación de datos como sinónimas.

Existen varios lenguajes de consultas de bases de datos en uso, tanto comercial como experimentalmente. El lenguaje de consultas más ampliamente usado, SQL , se estudiará más adelante.

Los niveles de abstracción que se trataron en el Apartado 1.3 no sólo se aplican a la definición o estructuración de datos, sino también a su manipulación. En el nivel físico se deben definir los algoritmos que permitan un acceso eficiente a los datos. En los niveles superiores de abstracción se pone el énfasis en la facilidad de uso. El objetivo es permitir que los seres humanos interactúen de manera eficiente con el sistema. El componente procesador de consultas del sistema de bases de datos traduce las consultas LMD en secuencias de acciones en el nivel físico del sistema de bases de datos.

0.7.2. Lenguaje de definición de datos

Los esquemas de las bases de datos se especifican mediante un conjunto de definiciones expresadas mediante un lenguaje especial denominado lenguaje de definición de datos (LDD). El LDD también se usa para especificar más propiedades de los datos.

La estructura de almacenamiento y los métodos de acceso usados por el sistema de bases de datos se especifican mediante un conjunto de instrucciones en un tipo especial de LDD denominado lenguaje de almacenamiento y definición de datos. Estas instrucciones definen los detalles de implementación de los esquemas de las bases de datos, que suelen ocultarse a los usuarios.

Los valores de los datos almacenados en la base de datos deben satisfacer ciertas restricciones de consistencia. Por ejemplo, supóngase que el saldo de una cuenta no debe caer por debajo de 100 e. El LDD proporciona facilidades para especificar tales restricciones. Los sistemas de bases de datos las comprueban cada vez que se modifica la base de datos. En general, las restricciones pueden ser predicados arbitrarios relativos a la base de datos. No obstante, los predicados arbitrarios pueden resultar costosos de comprobar. Por tanto, los sistemas de bases de datos se concentran en las restricciones de integridad que pueden comprobarse con una sobrecarga mínima:

- Restricciones de dominio. Se debe asociar un dominio de valores posibles a cada atributo (por ejemplo, tipos enteros, tipos de carácter, tipos fecha/hora). La declaración de un atributo como

parte de un dominio concreto actúa como restricción de los valores que puede adoptar. Las restricciones de dominio son la forma más elemental de restricción de integridad. El sistema las comprueba fácilmente siempre que se introduce un nuevo elemento de datos en la base de datos.

- Integridad referencial. Hay casos en los que se desea asegurar que un valor que aparece en una relación para un conjunto de atributos dado aparece también para un determinado conjunto de atributos en otra relación (integridad referencial). Las modificaciones de la base de datos pueden causar violaciones de la integridad referencial. Cuando se viola una restricción de integridad, el procedimiento normal es rechazar la acción que ha causado esa violación.

- Asertos. Un aserto es cualquier condición que la base de datos debe satisfacer siempre. Las restricciones de dominio y las restricciones de integridad referencial son formas especiales de asertos. No obstante, hay muchas restricciones que no pueden expresarse empleando únicamente esas formas especiales. Por ejemplo: "Cada préstamo tiene como mínimo un cliente tenedor de una cuenta con un saldo mínimo de 1.000,00 e" debe expresarse en forma de aserto. Cuando se crea un aserto, el sistema comprueba su validez. Si el aserto es válido, cualquier modificación futura de la base de datos se permite únicamente si no hace que se viole ese aserto.

- Autorización. Puede que se desee diferenciar entre los usuarios en cuanto al tipo de acceso que se les permite a diferentes valores de los datos de la base de datos. Estas diferenciaciones se expresan en términos de autorización, cuyas modalidades más frecuentes son: autorización de lectura, que permite la lectura pero no la modificación de los datos; autorización de inserción, que permite la inserción de datos nuevos, pero no la modificación de los datos ya existentes; autorización de actualización, que permite la modificación, pero no la eliminación, de los datos; y la autorización de eliminación, que permite la eliminación de datos. A cada usuario se le pueden asignar todos, ninguno o una combinación de estos tipos de autorización.

El LDD , al igual que cualquier otro lenguaje de programación, obtiene como entrada algunas instrucciones y genera una salida. La salida del LDD se coloca en el diccionario de datos, que contiene metadatos— es decir, datos sobre datos. El diccionario de datos se considera un tipo especial de tabla, a la que sólo puede tener acceso y actualizar el propio sistema de bases de datos (no los usuarios normales). El sistema de bases de datos consulta el diccionario de datos antes de leer o modificar los datos reales.

Sistemas de bases de datos

Un sistema de gestión de bases de datos (SGBD o DBMS "Database Management System") consiste en una colección de datos interrelacionados y un conjunto de programas que permiten a los usuarios acceder y modificar dichos datos. La colección de datos se denomina base de datos El primer objetivo de un SGBD es proporcionar un entorno que sea tanto práctico como eficiente de usar en la recuperación y el almacenamiento de la información de la base de datos. Otro de los objetivos principales de un SGBD es proporcionar al usuario una visión abstracta de la información, es decir, el sistema oculta detalles como los relativos a la forma de almacenar y mantener los datos, de tal forma que para que el sistema sea útil la información ha de recuperarse de forma eficiente.

La búsqueda de la eficiencia conduce al diseño de estructuras complejas para usuarios sin conocimientos de computación, para lo cual esta complejidad ha de estar oculta. Para poder lograr lo anterior es necesario definir los distintos niveles de abstracción de una base de datos, lo que constituirá el marco necesario para identificar las diferentes funciones que han de cumplir estos sistemas.

0.8. El sistema de gestión de bases de datos

Un Sistema de Gestión de Bases de Datos (SGBD) es el conjunto de programas que permiten definir, manipular y utilizar la información que contienen las bases de datos, realizar todas las tareas de administración necesarias para mantenerlas operativas, mantener su integridad, confidencialidad y seguridad. Una BD nunca se accede o manipula directamente sino a través del SGBD. Se puede considerar al SGBD como el interfaz entre el usuario y la BD. El funcionamiento del SGBD está muy interrelacionado con el del Sistema Operativo, especialmente con el sistema de comunicaciones. El SGBD utilizará las facilidades del sistema de comunicaciones para recibir las peticiones del usuario (que puede estar utilizando un terminal físicamente remoto) y para devolverle los resultados. Conceptualmente, lo que ocurre es lo siguiente:

1. Un usuario hace una petición de acceso, usando algún lenguaje en particular (normalmente SQL).
2. El SGBD intercepta esa petición y la analiza.
3. El SGBD inspecciona el esquema externo de ese usuario, la correspondencia externa/conceptual, el esquema conceptual, la correspondencia conceptual/interna, y la definición de la estructura de almacenamiento.
4. El SGBD ejecuta las operaciones necesarias en la base de datos almacenada.

El SGBD tiene las siguientes funciones:

1. Definición de datos

   El SGBD debe ser capaz de aceptar definiciones de datos (esquemas externos, el esquema conceptual, el esquema interno, y todas las correspondencias asociadas) en versión fuente y convertirlas en la versión objeto apropiada. Dicho de otro modo, el SGBD debe incluir componentes procesadores de lenguajes para cada uno de los diversos lenguajes de definición de datos (DDL). El SGBD también debe entender las definiciones en DDL, en el sentido en que, por ejemplo, entiende que los registros externos "Empleado" contienen un campo "Salario"; y debe poder utilizar estos conocimientos para interpretar y responder las solicitudes de los usuarios (por ejemplo una consulta de todos los empleados cuyo salario sea inferior a 100.000 pts).

2. Manipulación de datos

   El SGBD debe ser capaz de atender las solicitudes del usuario para extraer, y quizá actualizar, datos que ya existen en la base de datos, o para agregar en ella datos nuevos. Dicho de otro modo, el SGBD debe incluir un componente procesador de lenguaje de manipulación de datos (DML).

3. Seguridad e integridad de los datos El SGBD debe supervisar las solicitudes de los usuarios y rechazar los intentos de violar las medidas de seguridad e integridad definidas por el administrador de la base de datos. Recuperación y concurrencia de los datos El SGBD (o en su defecto algún componente de software relacionado con él, al que normalmente se le denomina administrados de transacciones) debe cuidar del cumplimiento de ciertos controles de recuperación y concurrencia.

4. Diccionario de datos

   El SGBD debe incluir una función de diccionario de datos. Puede decirse que el diccionario de datos es una base de datos (del sistema, no del usuario). El contenido del diccionario puede considerarse como "datos acerca de los datos" (metadatos), es decir, definiciones de otros objetos en el sistema, y no sólo datos en bruto. En particular, en el diccionario de datos se almacenarán físicamente todos los diversos esquemas y correspondencias (externos, conceptuales,etc.) tanto en sus versiones fuente como en las versiones objeto. Un diccionario completo incluirá también referencias cruzadas para indicar, por ejemplo, qué bases de datos utilizan los programas, que informes requieren los usuarios, qué terminales están conectadas al sistema... Es más, el diccionario podría (y quizá debería) estar integrado a la base de datos a la cual define, e incluir por tanto su propia definición. Deberá ser posible consultar el diccionario igual que cualquier otra base de datos de modo que se puede saber, por ejemplo, qué programas o usuarios podrían verse afectados por alguna modificación propuesta para el sistema.

Como conclusión, podemos decir que el SGBD constituye la interfaz entre el usuario y el sistema de bases de datos. La interfaz del usuario puede definirse como una frontera del sistema, más allá de la cual todo resulta invisible para el usuario. Por definición, entonces, la interfaz del usuario está en el nivel externo.

0.8.1. Características de extensibilidad de los SGBD

Los SGBD deben reunir una serie de características que contemplen las nuevas funcionalidades que deben proporcionar en estos momentos. Dichas características son:

1. Soporte ODBC

   ODBC (siglas que significan Open DataBase Conectivity, Conectividad Abierta de Bases de Datos) se define como un método común de acceso a bases de datos, diseñado por Microsoft para simplificar la comunicación en Bases de Datos Cliente/Servidor. ODBC consiste en un conjunto de llamadas de bajo nivel que permite a las aplicaciones en el cliente intercambiar instrucciones con las aplicaciones del servidor y compartir datos, sin necesidad de conocer nada unas respecto a las otras. Las aplicaciones emplean módulos, llamados controladores de bases de datos, que unen la aplicación con el SGBD concreto elegido. Se emplea el SQL como lenguaje de acceso a los datos. El SGBD debe proporcionar los controladores adecuados para poder ser empleados por los distintos lenguajes de programación que soporten ODBC.

2. Orientación a objetos

   Los SGDB relacionales tradicionales sólo pueden almacenar y tratar con números y cadenas de caracteres. Las mejoras en el terreno de la multimedia obligan a que las aplicaciones desarrolladas actualmente precisen cada vez más almacenar, junto con la información numérica y de caracteres, tipos de datos más complejos que permitan gestionar objetos de sonido, imágenes, vídeos, etc. Algunos SGBD relacionales avanzaron en este sentido dando cabida en sus BD a tipos de datos binarios (donde se puede guardar código binario, que es el que forma los objetos de sonido, imágenes, programas ejecutables, etc.), pero esto no es suficiente. La aparición de SGBD relacionales Orientados a Objetos (SGBDROO) proporcionan toda la potencia y robustez de los SGBD relacionales, y al mismo tiempo, permiten gestionar objetos de un modo nativo, así como los campos numéricos y de caracteres que se han visto recogidos tradicionalmente. Los SGBDROO cuentan con todas las posibilidades de un motor de consultas SQL clásico, pero el lenguaje puede manipular tipos definidos por el usuario, de la misma manera que gestiona los tipos predefinidos de los sistemas más antiguos. Por lo tanto, se trata de un SGBD relacional, pero extendido y ampliado de manera que soporte la gestión de objetos. Por otra parte, la tendencia a la generación de aplicaciones distribuidas, donde los usuarios, datos y componentes de la aplicación están físicamente separados, facilita e impulsa el uso de SGDROO, pues los objetos y las aplicaciones distribuidas están "hechos el uno para el otro".

3. Conectividad en Internet

   Los distintos SGDB existentes incorporan en sus últimas versiones software de tipo middleware (capa de software que se sitúa sobre el SGBD) para añadir conectividad a la base de datos a través de Internet. Microsoft ha desarrollado los ADO (Access Database Object, Objetos de Acceso a Bases de Datos) que, incorporados en scripts dentro de páginas Web en HTML, proporcionan conexión con Bases de Datos, tanto locales como remotas, empleando ODBC. Los middleware desarrollados en los distintos SGBDs suelen emplear ODBC (o JDBC, conectividad abierta de bases de datos preparada para el lenguaje Java) para conectar con la BD, junto con diversos conjuntos de herramientas para facilitar al usuario la implementación de la comunicación con la BD a través de Internet.

4. Soporte de estándares objetuales

   Hay varios estándares de objetos diseñados para proporcionar una guía en el diseño y desarrollo de aplicaciones distribuidas que trabajen con BD relaciones con orientación a objetos. Los SGBDs actuales hacen uso de software del tipo middleware que asumen las tareas de servicio de transacciones de objeto siguiendo alguno de los estándares de objetos existentes. Los principales estándares de objeto son:

a) CORBA (Common Object Broker Architecture, o Arquitectura común de gestores de solicitudes de objetos), del Object Management Group (OMG).

b) DCOM (Distributed Component Model) de Microsoft.

c) Java Remote Method Invocation de Sun.

Los actuales SGBD proporcionan soporte, como mínimo, a CORBA y DCOM. Data Mining, Data Warehousing, OLAP

Los SGBD deben incorporar una serie de herramientas que permitan, de forma cómoda, sencilla e intuitiva, la extracción y disección-minería de datos (Data Mining), y soporte para OLAP (OnLine Analytical Processing, Procesamiento Análitico de Datos En Vivo), que se trata de una categoría de las nuevas tecnologías del software que permite obtener y extraer información mediante un complejo análisis y procesamiento del contenido de una Base de Datos, todo ello en tiempo real.

También deben proporcionar una estabilidad y robustez cada vez mejores, que permitan mejorar los almacenes de datos (Data Warehousing), mercados de datos (Data Marts) y Webs de datos, procesos de transacciones y otras aplicaciones de misión crítica.

0.9. Arquitecturas de sistemas de bases de datos

Anteriormente se ha analizado con cierto detalle la arquitectura ANSI/SPARC para sistemas de bases de datos. En esta sección vamos a examinar los sistemas de bases de datos desde un punto de vista diferente.

La arquitectura de un sistema de base de datos está influenciada en gran medida por el sistema informático subyacente en el que se ejecuta el sistema de base de datos. En la arquitectura de un sistema de base de datos se reflejan aspectos como la conexión en red, el paralelismo y la distribución:

- La arquitectura centralizada es la más clásica. En ella, el SGBD está implantado en una sola plataforma u ordenador desde donde se gestiona directamente, de modo centralizado, la totalidad de los recursos. Es la arquitectura de los centros de proceso de datos tradicionales. Se basa en tecnologías sencillas, muy experimentadas y de gran robustez.

- La conexión en red de varias computadoras permite que algunas tareas se ejecuten en un sistema servidor y que otras se ejecuten en los sistemas clientes. Esta división de trabajo ha conducido al desarrollo de sistemas de bases de datos cliente-servidor.

- La distribución de datos a través de las distintas sedes o departamentos de una organización permite que estos datos residan donde han sido generados o donde son más necesarios, pero continuar siendo accesibles desde otros lugares o departamentos diferentes. El hecho de guardar varias copias de la base de datos en diferentes sitios permite que puedan continuar las operaciones sobre la base de datos aunque algún sitio se vea afectado por algún desastre natural, como una inundación, un incendio o un terremoto. Se han desarrollado los sistemas de bases de datos distribuidos para manejar datos distribuidos geográfica o administrativamente a lo largo de múltiples sistemas de bases de datos.

- El procesamiento paralelo dentro de una computadora permite acelerar las actividades del sistema de base de datos, proporcionando a las transacciones unas respuestas más rápidas, así como la capacidad de ejecutar más transacciones por segundo. Las consultas pueden procesarse

de manera que se explote el paralelismo ofrecido por el sistema informático subyacente. La necesidad del procesamiento paralelo de consultas ha conducido al desarrollo de los sistemas de bases de datos paralelos.

No debe confundirse el SGBD con la arquitectura que se elige para implantarlo. Algunos SGBD sólo se pueden implantar en una de las arquitecturas y otros en todas ellas.

0.9.1. Arquitectura centralizada

Los sistemas de bases de datos centralizados son aquellos que se ejecutan en un único sistema informático sin interaccionar con ninguna otra computadora. Tales sistemas comprenden el rango desde los sistemas de bases de datos monousuario ejecutándose en computadoras personales hasta los sistemas de bases de datos de alto rendimiento ejecutándose en grandes sistemas.

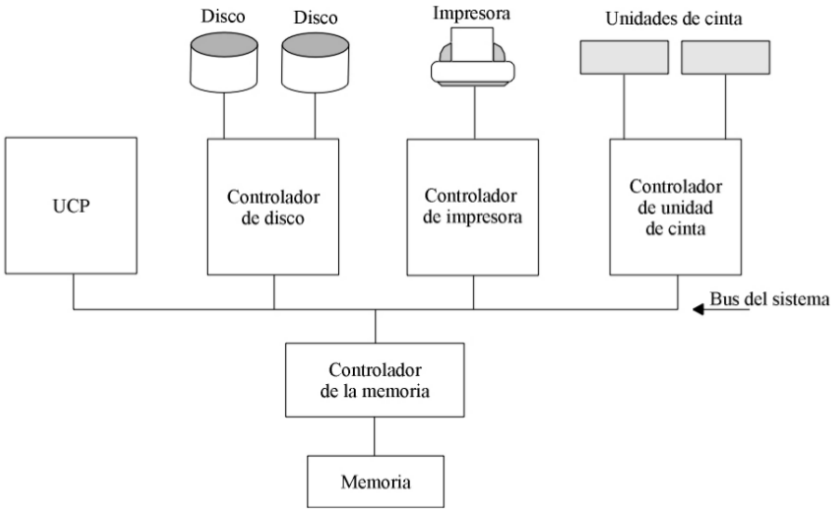


Figura 2: Un sistema informático centralizado.

Una computadora moderna de propósito general consiste en una o unas pocas CPU"s y un número determinado de controladores para los dispositivos que se encuentren conectados a través de un bus común, el cual proporciona acceso a la memoria compartida. Las CPU"s poseen memorias caché locales donde se almacenan copias de ciertas partes de la memoria para acelerar el acceso a los datos. Cada controlador de dispositivo se encarga de un tipo específico de dispositivos (por ejemplo, una unidad de disco, una tarjeta de sonido o un monitor). La CPU y los controladores de dispositivo pueden ejecutarse concurrentemente, compitiendo así por el acceso a la memoria. La memoria caché reduce la disputa por el acceso a la memoria, ya que la CPU necesita acceder a la memoria compartida un número de veces menor.

Se distinguen dos formas de utilizar las computadoras: como sistemas monousuario o como sistemas multiusuario. En la primera categoría están las computadoras personales y las estaciones de trabajo. Un sistema monousuario típico es una unidad de sobremesa utilizada por una única persona que dispone de una sola CPU, de uno o dos discos fijos y que trabaja con un sistema operativo que sólo permite un único usuario. Por el contrario, un sistema multiusuario típico tiene más discos y más memoria, puede disponer de varias CPU y trabaja con un sistema operativo multiusuario. Se

encarga de dar servicio a un gran número de usuarios que están conectados al sistema a través de terminales. Estos sistemas se denominan con frecuencia sistemas servidores.

Normalmente, los sistemas de bases de datos diseñados para funcionar sobre sistemas monousuario, como las computadoras personales, no suelen proporcionar muchas de las facilidades que ofrecen los sistemas multiusuario. En particular, no tienen control de concurrencia, que no es necesario cuando solamente un usuario puede generar modificaciones. Las facilidades de recuperación en estos sistemas, o no existen o son primitivas; por ejemplo, realizar una copia de seguridad de la base de datos antes de cualquier modificación. La mayoría de estos sistemas no admiten SQL y proporcionan un lenguaje de consulta muy simple, que en algunos casos es una variante de QBE (Query By Example).

Aunque hoy en día las computadoras de propósito general tienen varios procesadores, utilizan paralelismo de grano grueso, disponiendo de unos pocos procesadores (normalmente dos o cuatro) que comparten la misma memoria principal. Las bases de datos que se ejecutan en tales máquinas habitualmente no intentan dividir una consulta simple entre los distintos procesadores, sino que ejecutan cada consulta en un único procesador, posibilitando la concurrencia de varias consultas. Así, estos sistemas soportan una mayor productividad, es decir, permiten ejecutar un mayor número de transacciones por segundo, a pesar de que cada transacción individualmente no se ejecuta más rápido.

Las bases de datos diseñadas para las máquinas monoprocesador ya disponen de multitarea, permitiendo que varios procesos se ejecuten a la vez en el mismo procesador, usando tiempo compartido, mientras que de cara al usuario parece que los procesos se están ejecutando en paralelo. De esta manera, desde un punto de vista lógico, las máquinas paralelas de grano grueso parecen ser idénticas a las máquinas monoprocesador, y pueden adaptarse fácilmente los sistemas de bases de datos diseñados para máquinas de tiempo compartido para que puedan ejecutarse sobre máquinas paralelas de grano grueso. Por el contrario, las máquinas paralelas de grano fino tienen un gran número de procesadores y los sistemas de bases de datos que se ejecutan sobre ellas intentan paralelizar las tareas simples (consultas, por ejemplo) que solicitan los usuarios.

Arquitectura de bases de datos cliente/servidor

0.10. Introducción

Con el aumento de la velocidad y potencia de las computadoras personales y el decremento en su precio, los sistemas se han ido distanciando de la arquitectura centralizada. Los terminales conectados a un sistema central han sido suplantados por computadoras personales. De igual forma, la interfaz de usuario, que solía estar gestionada directamente por el sistema central, está pasando a ser gestionada cada vez más por las computadoras personales. Como consecuencia, los sistemas centralizados actúan hoy como sistemas servidores que satisfacen las peticiones generadas por los sistemas clientes.

La computación cliente/servidor es la extensión lógica de la programación modular. El supuesto principal de la programación modular es la división de un programa grande en pequeños programas (llamados módulos), siendo más fáciles el desarrollo y la mantenibilidad (divide y vencerás).

Cualquier LAN (red de área local) puede ser considerada como un sistema cliente/servidor, desde el momento en que el cliente solicita servicios como datos, ficheros o imprimir desde el servidor. Cuando un usuario se conecta a Internet, interactua con otros computadores utilizando el modelo cliente/servidor. Los recursos de Internet son proporcionados a través de computadores host, conocidos como servidores. El servidor es el computador que contiene información (bases de datos, ficheros de texto...). El usuario, o cliente, accede a esos recursos vía programas cliente (aplicaciones) que usan TCP/IP para entregar la información a su computadora.

Según esta definición, los sistemas cliente/servidor no están limitados a aplicaciones de bases de datos. Cualquier aplicación que tenga una interfaz de usuario (front-end, sección frontal o parte cliente) que se ejecute localmente en el cliente y un proceso que se ejecute en el servidor (back-end, sección posterior, o sistema subyacente) está en forma de computación cliente/servidor.

Conceptualmente, las plataformas cliente/servidor son parte del concepto de sistemas abiertos, en el cual todo tipo de computadores, sistemas operativos, protocolos de redes y otros, software y hardware, pueden interconectarse y trabajar coordinadamente para lograr los objetivos del usuario. Sin embargo, en la práctica, los problemas de alcanzar tal variedad de sistemas operativos, protocolos de redes, sistemas de base de datos y otros, que trabajen conjuntamente pueden ser muchos. El objetivo de los sistemas abiertos consiste en lograr la interoperabilidad, que es el estado de dos o más sistemas heterogéneos comunicándose y contribuyendo cada uno a alguna parte del trabajo que corresponde a una tarea común.

En cierto sentido, el enfoque cliente/servidor es la culminación de una percepción temprana de la potencia del cálculo distribuida conjuntamente con el control y el acceso a los datos inherentes a un

computador centralizado.

0.11. Características de un sistema cliente/servidor

Un sistema cliente/servidor es aquel en el que uno o más clientes y uno o más servidores, conjuntamente con un sistema operativo subyacente y un sistema de comunicación entre procesos, forma un sistema compuesto que permite cómputo distribuido, análisis, y presentación de los datos. Si existen múltiples servidores de procesamiento de base de datos, cada uno de ellos deberá procesar una base de datos distinta, para que el sistema sea considerado un sistema cliente/servidor. Cuando dos servidores procesan la misma base de datos, el sistema ya no se llama un sistema cliente/servidor, sino que se trata de un sistema de base de datos distribuido.

Los clientes, a través de la red, pueden realizar consultas al servidor. El servidor tiene el control sobre los datos; sin embargo los clientes pueden tener datos privados que residen en sus computadoras. Las principales características de la arquitectura cliente/servidor son:

- El servidor presenta a todos sus clientes una interfaz única y bien definida.
- El cliente no necesita conocer la lógica del servidor, sólo su interfaz externa.
- El cliente no depende de la ubicación física del servidor, ni del tipo de equipo físico en el que se encuentra, ni de su sistema operativo.
- Los cambios en el servidor implican pocos o ningún cambio en el cliente.

Como ejemplos de clientes pueden citarse interfaces de usuario para enviar comandos a un servidor, APIs (Aplication Program Interface) para el desarrollo de aplicaciones distribuidas, herramientas en el cliente para acceder a servidores remotos (por ejemplo, servidores de SQL) o aplicaciones que solicitan acceso a servidores para algunos servicios.

Como ejemplos de servidores pueden citarse servidores de ventanas como X-windows, servidores de archivos como NFS, servidores para el manejo de bases de datos (como los servidores de SQL), servidores de diseño y manufactura asistidos por computador, etc.

0.12. Partes de un sistema cliente/servidor

Los principales componentes de un sistema cliente/servidor son:

- El núcleo (back-end o sección posterior). Es el SGBD propiamente (servidor).
- El interfaz (front-end o sección frontal). Aplicaciones que funcionan sobre el SGBD (cliente).

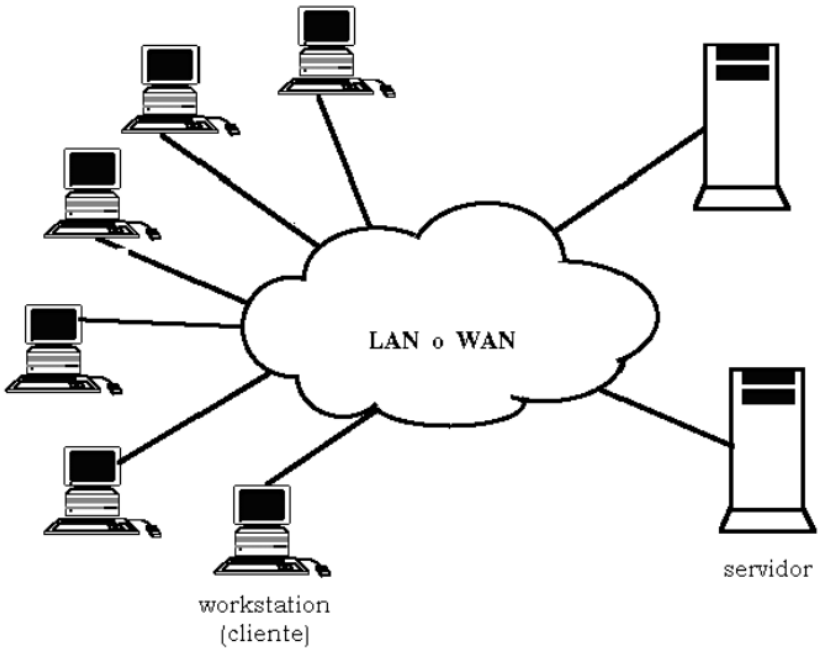


Figura 3: Ambiente cliente-servidor genérico.

La diferencia entre la computación cliente/servidor y la computación centralizada multiusuario es que el cliente no es un terminal "tonto". El computador cliente tiene su propio sistema operativo y puede manejar entradas (teclado, ratón, etc...) y salidas (pantalla, impresora local, sonido, etc...) sin el servidor. El papel del servidor es esperar pasivamente la petición de servicio del cliente. Esta distribución del proceso permite al cliente ofrecer un ambiente de trabajo más amigable que un terminal "tonto" (interfaz de usuario gráfica, aplicaciones locales, ratón, etc...) y permite al servidor ser menos complejo y caro que los sistemas mainframe. El conjunto de la computación cliente/servidor conduce a un ambiente flexible y dinámico. La parte cliente de la aplicación maneja la entrada de datos, acepta consultas de los usuarios y muestra los resultados. La parte cliente no procesa las consultas. En su lugar, envía la consulta del usuario al computador servidor, donde la parte servidor de la aplicación procesa la consulta. El servidor devuelve los resultados al cliente, que es quien se las muestra al usuario.

0.12.1. La sección frontal

Las secciones frontales son las diversas aplicaciones ejecutadas dentro del SGBD, tanto las escritas por los usuarios como las "integradas" que son las proporcionadas por el proveedor del SGBD o bien por otros proveedores de programas (aunque para la sección posterior no existe diferencia entre las aplicaciones escritas por los usuarios y las integradas, ya que todas utilizan la misma interfaz con la sección posterior).

0.12.1.1. Funciones del cliente

- Administrar la interfaz gráfica de usuario.
- Aceptar datos del usuario.

- Procesar la lógica de la aplicación.
- Generar las solicitudes para la base de datos.
- Transmitir las solicitudes de la base de datos al servidor.
- Recibir los resultados del servidor.
- Dar formato a los resultados.

0.12.1.2. Cómo trabaja la sección frontal

La secuencia de eventos que tiene lugar cuando el usuario accede al servidor de la base de datos se puede generalizar en 6 pasos básicos ilustrados en la figura. Para mayor simplicidad, el término consulta representa cualquier acción que el usuario pueda hacer en la base de datos, como actualizar, insertar, borrar o pedir datos de la base de datos.

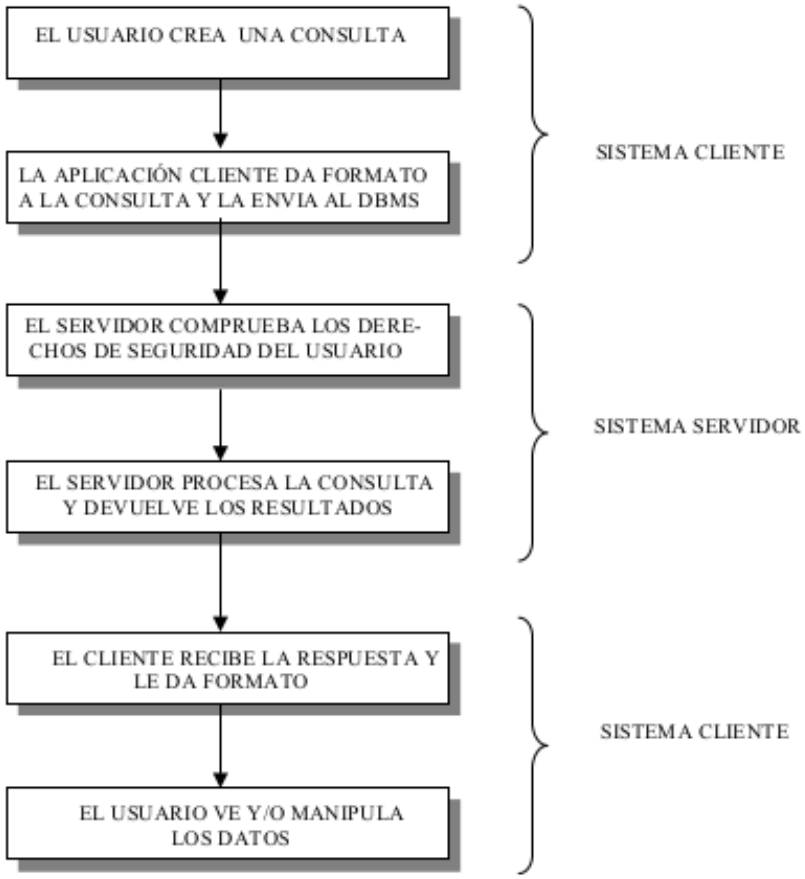


Figura 4: Diagrama que muestra como trabaja la sección frontal.

Primero, el usuario crea la consulta. Puede ser una consulta creada en el instante o puede ser una consulta preprogramada o almacenada anteriormente. Después la aplicación cliente convierte la

consulta al SQL usado por el servidor de la base de datos y la envía a través de la red al servidor.

El servidor verifica que el usuario tiene los derechos de seguridad apropiados a la consulta de datos requerida. Si es así, verifica la consulta y envía los datos apropiados de vuelta al cliente. La aplicación cliente recibe la respuesta y le da formato para presentarlo al usuario.

Finalmente, el usuario ve la respuesta en la pantalla y puede manipular los datos, o modificar la consulta y empezar el proceso de nuevo.

0.12.1.3. Tipos de aplicaciones cliente

Las aplicaciones pueden dividirse en varias categorías más o menos bien definidas: En primer lugar, aplicaciones escritas por los usuarios. Casi siempre se trata de programas comunes de aplicación, escritos (normalmente) en un lenguaje de programación convencional (por ejemplo Cobol), o bien en algún lenguaje propio (como Focus), aunque en ambos casos el lenguaje debe acoplarse de alguna manera con un sublenguaje de datos apropiado.

En segundo lugar, aplicaciones suministradas por los proveedores (herramientas). El objetivo general de estas herramientas es ayudar en el proceso de creación y ejecución de otras aplicaciones, o sea, aplicaciones hechas a la medida para alguna tarea específica (aunque la aplicación creada quizá no parezca una aplicación en el sentido convencional; de hecho, la verdadera razón para utilizar herramientas es que los usuarios, sobre todo los finales, puedan crear aplicaciones sin tener que escribir programas convencionales).

Por ejemplo, una de las herramientas suministradas por los proveedores sería un procesador de lenguaje de consulta, cuyo propósito es desde luego permitir a los usuarios finales hacer consultas al sistema. En esencia, cada una de esas consultas no es más que una pequeña aplicación hecha a la medida para realizar alguna función de aplicación específica.

A su vez, las herramientas suministradas por los proveedores se dividen en varias clases distintas:

- Procesadores de lenguajes de consulta
- Generadores de informes
- Subsistemas de gráficas para negocios
- Hojas electrónicas de cálculo
- Procesadores de lenguajes naturales
- Paquetes estadísticos
- Herramientas para administrar copias
- Generadores de aplicaciones (incluyendo procesador de lenguajes de cuarta generación o 4GL)
- Otras herramientas para desarrollar aplicaciones, incluyendo productos para ingeniería de software asistida por computador (CASE).

0.13. La sección posterior

La sección posterior es el SGBD en sí. Permite llevar a cabo todas las funciones básicas de un SGBD: definición de datos, manipulación de datos, seguridad, integridad, etc... En particular, permite establecer todos los aspectos de los niveles externo, conceptual e interno (arquitectura ANSI/SPARC)

0.13.1. Funciones del servidor

- Aceptar las solicitudes de la base de datos de los clientes.
- Procesar dichas solicitudes.
- Dar formato a los resultados y transmitirlos al cliente.
- Llevar a cabo la verificación de integridad.
- Mantener los datos generales de la base de datos.
- Proporcionar control de acceso concurrente.
- Llevar a cabo la recuperación.
- Optimizar el procesamiento de consultas y actualización.

3.2.2 Tipos de servidores Podemos dividir los servidores en dos clases: iterativos y concurrentes. Un servidor iterativo realiza los siguientes pasos:

- Espera que llegue una consulta de un cliente.
- Procesa la consulta.
- Envía la respuesta al cliente que envió la consulta.
- Vuelve al estado inicial.

El problema del servidor iterativo es el paso 2. Durante el tiempo en el que el servidor está procesando la consulta, ningún otro cliente es servido. Un servidor concurrente realiza los siguientes pasos:

- Espera que llegue la consulta de un cliente.
- Cuando le llega una nueva consulta, comienza un nuevo proceso para manejar esta consulta (cómo se realiza este paso depende del sistema operativo). El nuevo servidor maneja la totalidad de la consulta. Cuando se ha procesado completamente, este nuevo proceso termina.
- Se vuelve al primer paso.

La ventaja del servidor concurrente es que el servidor ejecuta un nuevo proceso para manejar cada consulta. Cada cliente tiene su "propio"servidor. Asumiendo que el sistema operativo permite la multiprogramación, clientes múltiples y servicio concurrente.

También podemos dividir los servidores en servidores de transacciones y servidores de datos. Los sistemas servidores de transacciones, también llamados sistemas servidores de consultas, proporcionan una interfaz a través de la cual los clientes pueden enviar peticiones para realizar una acción que el servidor ejecutará y cuyos resultados se devolverán al cliente. Los usuarios pueden especificar sus peticiones con SQL o mediante la interfaz de una aplicación utilizando un mecanismo de llamadas a procedimientos remotos (RPC: 'Remote Procedure Call').

Los sistemas servidores de datos permiten que los clientes puedan interaccionar con los servidores realizando peticiones de lectura o modificación de datos en unidades tales como archivos o páginas. Por ejemplo, los servidores de archivos proporcionan una interfaz de sistema de archivos a través de

la cual los clientes pueden crear, modificar, leer y borrar archivos. Los servidores de datos de los sistemas de bases de datos ofrecen muchas más funcionalidades; soportan unidades de datos de menor tamaño que los archivos, como páginas, tuplas u objetos. Proporcionan facilidades de indexación de los datos, así como facilidades de transacción, de modo que los datos nunca se quedan en un estado inconsistente si falla una máquina cliente o un proceso.

0.13.1.1. Servidores de transacciones

En los sistemas centralizados, un solo sistema ejecuta tanto la parte visible al usuario como el sistema subyacente. Sin embargo, la arquitectura del servidor de transacciones responde a la división funcional entre la parte visible al usuario y el sistema subyacente. Las computadoras personales gestionan la funcionalidad de la parte visible al usuario debido a las grandes necesidades de procesamiento que requiere el código de la interfaz gráfica, ya que la creciente potencia de las computadoras personales permite responder a esas necesidades. Los sistemas servidores tienen almacenado un gran volumen de datos y soportan la funcionalidad del sistema subyacente, mientras que las computadoras personales actúan como clientes suyos. Los clientes envían transacciones a los sistemas servidores, que se encargan de ejecutar aquellas transacciones y enviar los resultados de vuelta a los clientes, que se encargan a su vez de mostrar los datos en pantalla.

Se han desarrollado distintas normas como ODBC ('Open Database Connectivity', Conectividad abierta de bases de datos), para la interacción entre clientes y servidores. ODBC es una interfaz de aplicación que permite que los clientes generen instrucciones SQL para enviarlas al servidor en donde se ejecutan. Cualquier cliente que utilice la interfaz ODBC puede conectarse a cualquier servidor que proporcione dicha interfaz. Los sistemas de bases de datos de generaciones anteriores, al no haber tales normas, necesitaban que la parte visible al usuario y el sistema subyacente fueran proporcionados por el mismo fabricante.

Con el crecimiento de las normas de interfaz, las herramientas de la parte visible al usuario y los servidores del sistema subyacente son administrados cada vez por más marcas. Gupta SQL y PowerBuilder son ejemplos de sistemas visibles al usuario independientes de los servidores del sistema subyacente. Es más, ciertas aplicaciones, como algunas hojas de cálculo o algunos paquetes de análisis estadístico, utilizan directamente la interfaz cliente-servidor para acceder a los datos desde un servidor del sistema subyacente. Como resultado, proporcionan un sistema visible al usuario especializado en tareas concretas.

Algunos sistemas de procesamiento de transacciones utilizan, además de ODBC, otras interfaces cliente-servidor. Éstas se definen por medio de una interfaz de programación de aplicaciones y son utilizadas por los clientes para realizar llamadas a procedimientos remotos de transacciones sobre el servidor. Para el programador, estas llamadas son como las llamadas normales a procedimientos, pero en realidad todas las llamadas a procedimientos remotos desde un cliente se engloban en una única transacción en el servidor. De esta forma, si la transacción aborta, el servidor puede deshacer los efectos de las llamadas a procedimientos remotos individuales.

El hecho de que adquirir y mantener una máquina pequeña sea ahora más barato, ha desembocado en una tendencia hacia el fraccionamiento de costes cada vez más extendida en la industria. Muchas compañías están reemplazando sus grandes sistemas por redes de estaciones de trabajo o computadoras personales conectadas a máquinas servidoras del sistema subyacente. Entre las ventajas, destacan una mejor funcionalidad respecto del coste, una mayor flexibilidad para localizar recursos y facilidades de expansión, mejores interfaces de usuario y un mantenimiento más sencillo.

0.13.1.2. Servidores de datos

Los sistemas servidores de datos se utilizan en redes de área local en las que se alcanza una alta velocidad de conexión entre los clientes y el servidor, las máquinas clientes son comparables al servidor en cuanto a poder de procesamiento y se ejecutan tareas de cómputo intensivo. En este entorno, tiene sentido enviar los datos a las máquinas clientes, realizar allí todo el procesamiento (que puede durar un tiempo) y después enviar los datos de vuelta al servidor. Nótese que esta arquitectura necesita que los clientes posean todas las funcionalidades del sistema subyacente. Las arquitecturas de los servidores de datos se han hecho particularmente populares en los sistemas de bases de datos orientadas a objetos.

En esta arquitectura surgen algunos aspectos interesantes, ya que el coste en tiempo de comunicación entre el cliente y el servidor es alto comparado al de acceso a una memoria local (milisegundos frente a menos de 100 nanosegundos)

Envío de páginas o envío de elementos

La unidad de comunicación de datos puede ser de grano grueso (como una página), o de grano fino (como una tupla). Si la unidad de comunicación de datos es una única tupla, la sobrecarga por la transferencia de mensajes es alta comparada con el número de datos transmitidos. En vez de hacer esto, cuando se necesita una tupla, cobra sentido la idea de enviar junto a aquella otras tuplas que probablemente vayan a ser utilizadas en un futuro próximo. Se denomina preextracción a la acción de buscar y enviar tuplas antes de que sea estrictamente necesario. Si varias tuplas residen en una página, el envío de páginas puede considerarse como una forma de preextracción, ya que cuando un procese desee acceder a una única tupla de la página, se enviarán todos las tuplas de esa página.

Bloqueo

La concesión del bloqueo de las tuplas de datos que el servidor envía a los clientes la realiza habitualmente el propio servidor. Un inconveniente del envío de páginas es que los clientes pueden recibir bloqueos de grano grueso (el bloqueo de una página bloquea implícitamente a todas las tuplas que residan en ella).

El cliente adquiere implícitamente bloqueos sobre todas las tuplas preextraídas, incluso aunque no esté accediendo a alguno de ellos. De esta forma, puede detenerse innecesariamente el procesamiento de otros clientes que necesitan bloquear esos elementos.

Se han propuesto algunas técnicas para la liberación de bloqueos, en las que el servidor puede pedir a los clientes que le devuelvan el control sobre los bloqueos de las tuplas preextraídas.

Si el cliente no necesita la tupla preextraída, puede devolver los bloqueos sobre esa tupla al servidor para que éstos puedan ser asignados a otros clientes.

Caché de datos

Los datos que se envían al cliente a favor de una transacción se pueden alojar en una caché del cliente, incluso una vez completada la transacción, si dispone de suficiente espacio de almacenamiento libre.

Las transacciones sucesivas en el mismo cliente pueden hacer uso de los datos en caché.

Sin embargo, se presenta el problema de la coherencia de la caché: si una transacción encuentra los datos en la caché, debe asegurarse de que esos datos están al día, ya que después de haber sido almacenados en la caché pueden haber sido modificados por otro cliente.

Así, debe establecerse una comunicación con el servidor para comprobar la validez de los datos y poder adquirir un bloqueo sobre ellos.

Caché de bloqueos

Los bloqueos también pueden ser almacenados en la memoria caché del cliente si la utilización de los datos está prácticamente dividida entre los clientes, de manera que un cliente rara vez necesita datos que están siendo utilizados por otros clientes. Supóngase que se encuentran en la memoria caché tanto el elemento de datos que se busca como el bloqueo requerido para acceder al mismo.

Entonces, el cliente puede acceder al elemento de datos sin necesidad de comunicar nada al servidor. No obstante, el servidor debe seguir el rastro de los bloqueos en caché; si un cliente solicita un bloqueo al servidor, éste debe comunicar a todos los bloqueos sobre el elemento de datos que se encuentran en las memorias caché de otros clientes.

La tarea se vuelve más complicada cuando se tienen en cuenta los posibles fallos de la máquina. Esta técnica se diferencia de la liberación de bloqueos en que la caché de bloqueo se realiza a través de transacciones; de otra forma, las dos técnicas serían similares.

0.14. Tipos de arquitecturas cliente/servidor

0.14.1. Arquitectura de 2 capas

La arquitectura cliente/servidor tradicional es una solución de 2 capas. La arquitectura de 2 capas consta de tres componentes distribuidos en dos capas: cliente (solicitante de servicios) y servidor (proveedor de servicios). Los tres componentes son:

- Interfaz de usuario.
- Gestión del procesamiento.
- Gestión de la base de datos.

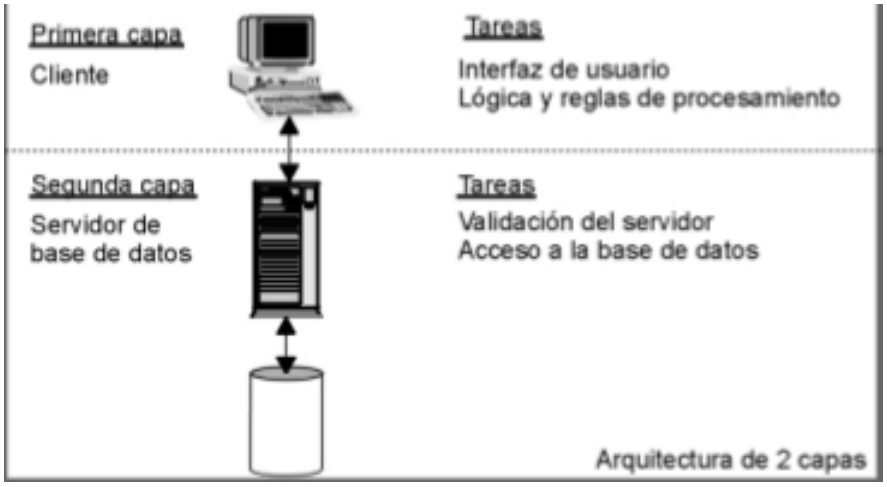


Figura 5: Arquitectura de dos capas.

Hay 2 tipos de arquitecturas cliente servidor de dos capas:

- Clientes obesos (thick clients): La mayor parte de la lógica de la aplicación (gestión del procesamiento) reside junto a la lógica de la presentación (interfaz de usuario) en el cliente, con la porción de acceso a datos en el servidor.

- Clientes delgados (thin clients): solo la lógica de la presentación reside en el cliente, con el acceso a datos y la mayoría de la lógica de la aplicación en el servidor. Es posible que un servidor funcione como cliente de otro servidor. Esto es conocido como diseño de dos capas encadenado.

Limitaciones

- El número usuarios máximo es de 100. Más allá de este número de usuarios se excede la capacidad de procesamiento.
- No hay independencia entre la interfaz de usuario y los tratamientos, lo que hace delicada la evolución de las aplicaciones.
- Dificultad de relocalizar las capas de tratamiento consumidoras de cálculo.
- Reutilización delicada del programa desarrollado bajo esta arquitectura.

0.14.2. Arquitectura de 3 capas

La arquitectura de 3 capas surgió para superar las limitaciones de la arquitectura de 2 capas. La tercera capa (servidor intermedio) está entre el interfaz de usuario (cliente) y el gestor de datos (servidor). La capa intermedia proporciona gestión del procesamiento y en ella se ejecutan las reglas y lógica de procesamiento. Permite cientos de usuarios (en comparación con sólo 100 usuarios de la arquitectura de 2 capas).

La arquitectura de 3 capas es usada cuando se necesita un diseño cliente/servidor que proporcione, en comparación con la arquitectura de 2 capas, incrementar el rendimiento, flexibilidad, mantenibilidad, reusabilidad y escalabilidad mientras se esconde la complejidad del procesamiento distribuido al usuario.

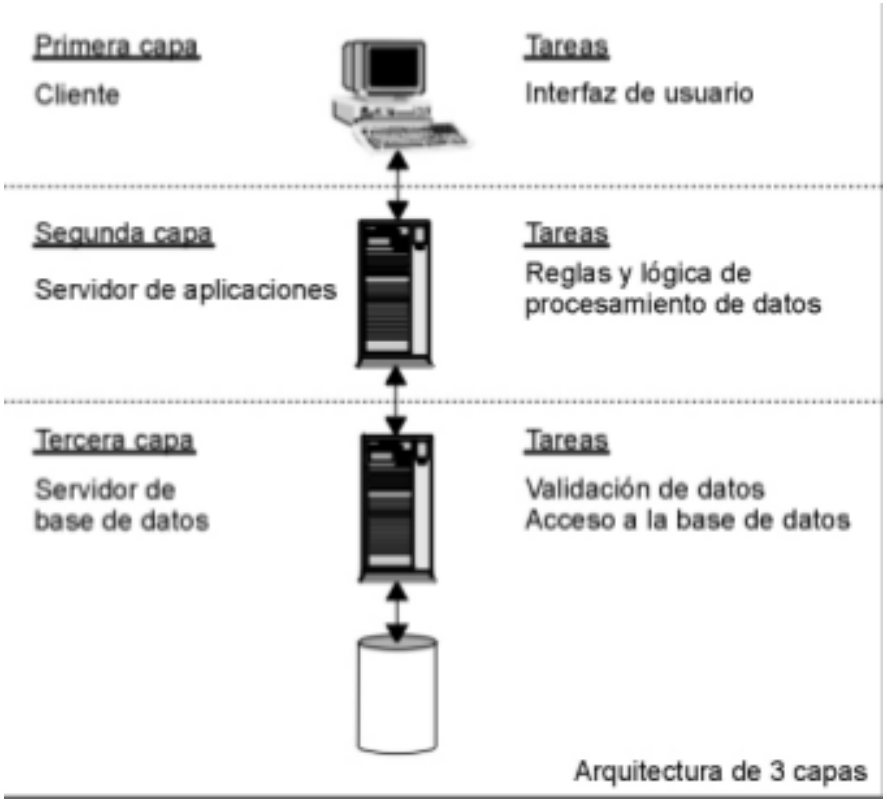


Figura 6: Arquitectura de dos capas.

Limitaciones Construir una arquitectura de 3 capas es una tarea complicada. Las herramientas de programación que soportan el diseño de arquitecturas de 3 capas no proporcionan todos los servicios deseados que se necesitan para soportar un ambiente de computación distribuida.

Un problema potencial en el diseño de arquitecturas de 3 capas es que la separación de la interfaz gráfica de usuario, la lógica de gestión de procesamiento y la lógica de datos no es siempre obvia. Algunas lógicas de procesamiento de transacciones pueden aparecer en las 3 capas. La ubicación de una función particular en una capa u otra debería basarse en criterios como los siguientes:

- Facilidad de desarrollo y comprobación
- Facilidad de administración.
- Escalabilidad de los servidores.
- Funcionamiento (incluyendo procesamiento y carga de la red)

El middleware

Como hemos visto, las capas están localizadas en máquinas diferentes que están conectadas a través de la red. El middleware es el software que proporciona un conjunto de servicios que permite el acceso transparente a los recursos en una red. El middleware es un módulo intermedio que actúa como conductor entre dos módulos de software. Para compartir datos, los dos módulos de software no necesitan saber cómo comunicarse entre ellos, sino cómo comunicarse con el módulo de middleware.

Es el encargado del acceso a los datos: acepta las consultas y datos recuperados directamente de la

aplicación y los transmite por la red. También es responsable de enviar de vuelta a la aplicación, los datos de interés y de la generación de códigos de error.

0.15. Ventajas e inconvenientes

0.15.1. Ventajas

Las principales ventajas del modelo Cliente/Servidor son las siguientes:

- Interoperabilidad: los componentes clave (cliente, servidor y red) trabajan juntos.
- Flexibilidad: la nueva tecnología puede incorporarse al sistema.
- Escalabilidad: cualquiera de los elementos del sistema puede reemplazarse cuando es necesario, sin impactar sobre otros elementos. Si la base de datos crece, las computadoras cliente no tienen que equiparse con memoria o discos adicionales. Esos cambios afectan solo a la computadora en la que se ejecuta la base de datos.
- Usabilidad: mayor facilidad de uso para el usuario.
- Integridad de los datos: entidades, dominios, e integridad referencial son mantenidas en el servidor de la base de datos.
- Accesibilidad: los datos pueden ser accedidos desde múltiples clientes.
- Rendimiento: se puede optimizar el rendimiento por hardware y procesos.
- Seguridad: la seguridad de los datos está centralizada en el servidor.

0.15.2. Inconvenientes

- Hay una alta complejidad tecnológica al tener que integrar una gran variedad de productos. El mantenimiento de los sistemas es más difícil pues implica la interacción de diferentes partes de hardware y de software, distribuidas por distintos proveedores, lo cual dificulta el diagnóstico de fallos.
- Requiere un fuerte rediseño de todos los elementos involucrados en los sistemas de información (modelos de datos, procesos, interfaces, comunicaciones, almacenamiento de datos, etc.). Además, en la actualidad existen pocas herramientas que ayuden a determinar la mejor forma de dividir las aplicaciones entre la parte cliente y la parte servidor.
- Es más difícil asegurar un elevado grado de seguridad en una red de clientes y servidores que en un sistema con un único ordenador centralizado (cuanto más distribuida es la red, mayor es su vulnerabilidad).
- A veces, los problemas de congestión de la red pueden reducir el rendimiento del sistema por debajo de lo que se obtendría con una única máquina (arquitectura centralizada). También la interfaz gráfica de usuario puede a veces ralentizar el funcionamiento de la aplicación.
- Existen multitud de costes ocultos (formación en nuevas tecnologías, cambios organizativos, etc.) que encarecen su implantación.

- La arquitectura cliente/servidor es una arquitectura que está en evolución, y como tal no existe estandarización.

0.16. Integridad de la base de datos

En la arquitectura cliente/servidor, todo el procesamiento de la base de datos queda consolidado en una sola computadora, y esta consolidación permite un alto grado de integridad de los datos. Al procesar las solicitudes a la base de datos en el servidor, si las restricciones han quedado definidas en el servidor, se pueden aplicar consistentemente.

Para comprender esto, veamos la siguiente figura:

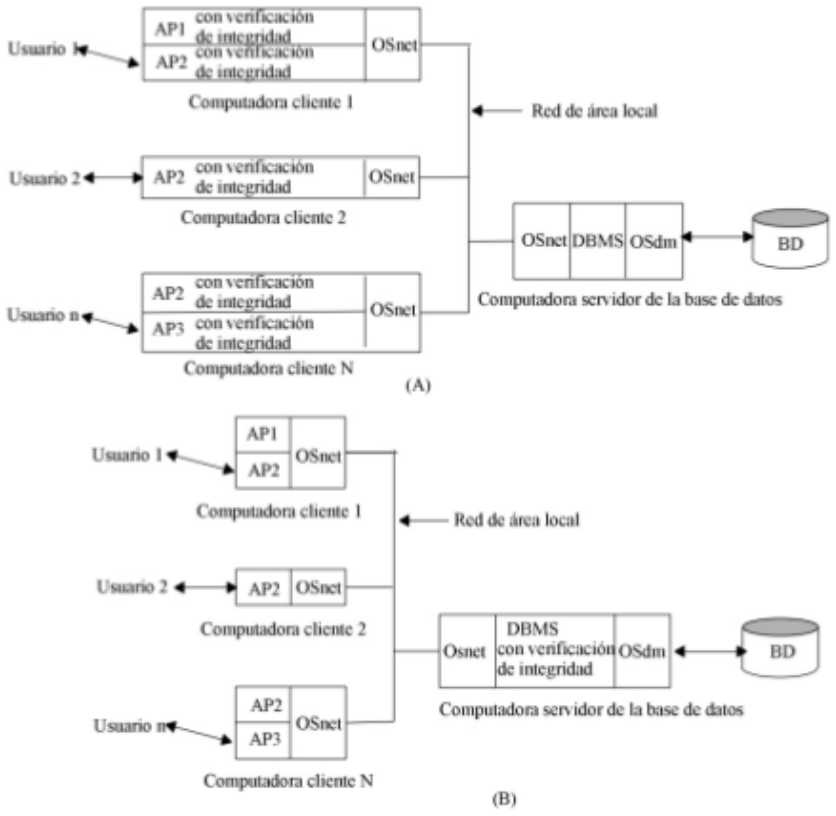


Figura 7: Verificación de la integridad cliente-servidor: a) verificación de la aplicación, b) verificación de la inegridad centralizada.

En la parte (a), la verificación de integridad no es llevada a cabo por el servidor. En vez de ello, los programas de aplicación de las computadoras cliente requieren que durante su procesamiento se verifiquen la integridad. En la parte (b), el SGBD verifica la integridad en el servidor.

Si se comparan ambas figuras, se verá por qué es preferible la verificación en el servidor. Si los que verifican la integridad son los clientes, la lógica de revisión deberá estar incluida en cada uno de los programas de aplicación, lo que no sólo es un desperdicio y resulta ineficiente, sino que también puede ser susceptible de errores. Los programadores de la aplicación pueden entender las restricciones de forma distinta y al programarlas pueden cometer equivocaciones. Siempre que se desarrolla una

aplicación, todas las verificaciones de las restricciones deberán duplicarse.

No todas las modificaciones de datos se efectúan a través de los programas de aplicación. Los usuarios pueden efectuar modificaciones a través de un lenguaje de consulta/actualización, y los datos pueden importarse de forma masiva. En demasiadas aplicaciones la integridad no es verificada al importar datos de estas fuentes.

Si el servidor se ocupa de la verificación de la integridad, las restricciones sólo necesitan definirse, verificarse y validarse una sola vez. Las modificaciones en los datos de todas las fuentes serán verificadas para asegurar la integridad. No tendrá ninguna importancia si una modificación ha sido sometida por un programa de aplicación, o por un proceso de consulta/actualización, o ha sido importada. Independientemente de la fuente, el SGBD se asegurará de que la integridad no ha sido violada.

0.17. Triggers

Un trigger[1] es un procedimiento de aplicación invocado de forma automática por el SGBD cuando ocurra algún evento. Por ejemplo, en una aplicación de inventario se puede escribir un gatillo para generar un pedido siempre que la cantidad de algún elemento se reduzca por debajo de algún valor de umbral. Para poner en práctica un gatillo, el desarrollador escribe un código de gatillo e informa al SGBD de su existencia y de las condiciones bajo las cuales se deberá invocar dicho gatillo. Cuando se cumplen tales condiciones, el SGBD llamará al gatillo. En un sistema cliente/servidor, los gatillos residen en y son invocados por el servidor.

Son útiles, pero los gatillos pueden resultar un problema. En presencia de ellos, un usuario puede provocar una actividad en la base de datos que no espera o que ni siquiera conoce.

En un entorno multiusuario, los procedimientos de tipo gatillo necesitan bloqueos. La transacción que genera el evento que dispara el gatillo pudiera ya haber obtenido bloqueos que entran en conflicto con dicho gatillo. En tal situación, una transacción puede crear un interbloqueo consigo misma.

Los gatillos se pueden disparar en cascada e incluso formar un ciclo cerrado. Se ponen en cascada cuando un gatillo genera una condición que hace que se invoque otro gatillo que a su vez crea una condición que genera la invocación a un tercer gatillo y así sucesivamente. Los gatillos forman ciclos cuando en cascada regresan a ellos mismos. En caso de que ocurra lo anterior, el SGBD deberá tener algún medio de evitar un ciclo infinito.

0.18. Control de procesamiento concurrente

Uno de los retos del desarrollo de las aplicaciones cliente/servidor es obtener el mayor paralelismo posible de las computadoras cliente, mientras se protege contra problemas como actualizaciones perdidas y lecturas inconsistentes.

Ya que varios usuarios podrían procesar los mismos datos de la base de datos, es necesario proteger contra problemas de actualización perdida y lecturas inconsistentes. Existen dos formas de hacer esto: Control mediante el bloqueo pesimista

La primera estrategia, a veces conocida como bloqueo pesimista (los bloqueos se colocan en anticipación de algún conflicto). Para poner en práctica esta estrategia, el SGBD coloca bloqueos

[1] GATILLO

implícitos en cada comando SGBD ejecutado después de START TRANSACTION. Tales bloqueos se mantienen entonces hasta que se emita un comando COMMIT o ROLLBACK.

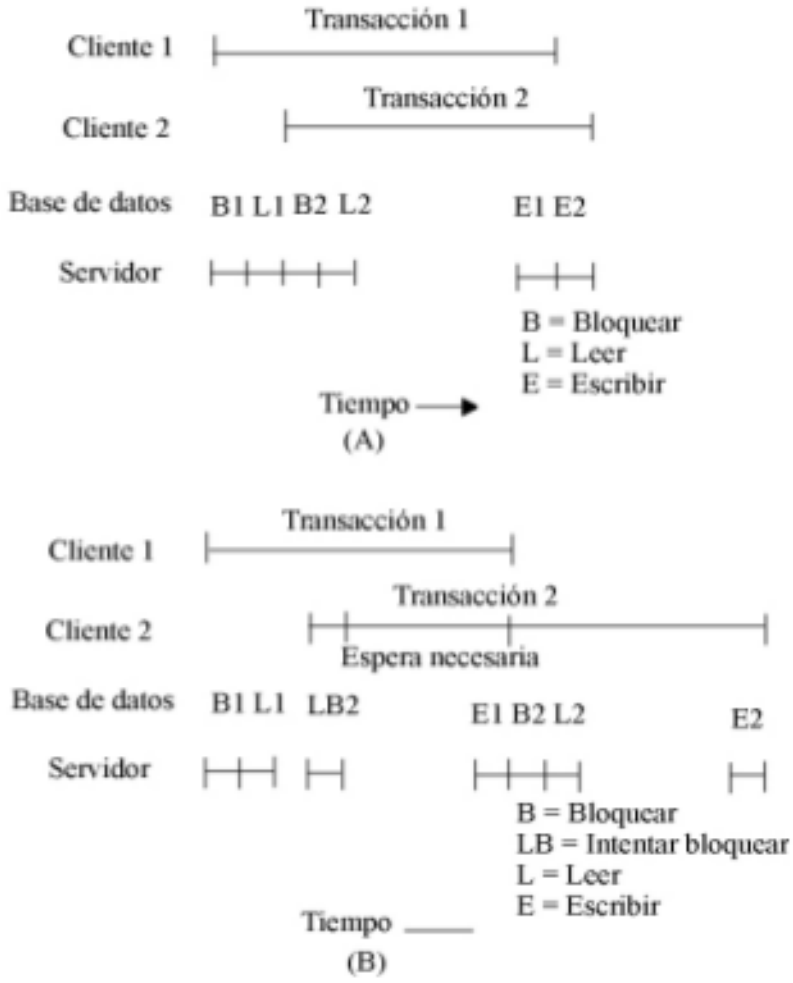


Figura 8: Bloqueos de un sistema cliente-servidor.

En la figura se muestran los tiempos requeridos para procesar una transacción. En la parte (a), las transacciones procesan diferentes datos, por lo que no existe conflicto de datos. El Cliente 2 espera mientras el servidor procesa la solicitud de datos del Cliente 1, pero una vez hecho esto, ambos clientes se procesan en paralelo hasta el final de la transacción.

En la parte (b) se muestran dos transacciones intentando procesar los mismos datos. En este caso, Cliente 2 espera datos hasta que Cliente 1 haya terminado de procesar su transacción. Esta espera puede resultar un problema. El Cliente 1 puede tener bloqueados los datos durante mucho tiempo (por ejemplo si tiene que introducir muchos datos en esa acción). Estos problemas empeoran si el SGBD no soporta bloqueos a nivel de tupla. Algunos SGBD bloquean una página (una sección de tuplas), en vez de una sola tupla, y algunos incluso bloquean toda la tabla. Si el bloqueo se lleva a cabo con estos grandes niveles de granularidad, los bloqueos implícitos pueden dar como resultado retrasos muy notables e innecesarios.

La situación mostrada en la siguiente figura supone un bloqueo a nivel de página. Dos transacciones están procesando dos tuplas distintas que por casualidad están en la misma página. En este caso, el Cliente 2 debe de esperar a que termine Cliente 1, aun cuando las dos transacciones están procesándose en diferentes tuplas. El retraso puede ser sustancial y, en este caso, también es innecesario.

Control mediante bloqueos optimistas

Una segunda estrategia, a veces conocida como bloqueo optimista, ya que no se prevé ningún conflicto, pero en caso de que hubiera alguno uno de los usuarios deberá volver a hacer su trabajo. Consiste en retrasar el bloqueo hasta el último momento posible, esperando que no exista conflicto. En este estilo, se procesa tanto de la transacción como sea posible antes de colocar cualquier bloqueo. A continuación se obtienen los bloqueos, conservándose de modo muy breve. Para utilizar esta estrategia, el SGBD no debe aplicar ningún bloqueo implícito, el programa de aplicación deberá colocar dichos bloqueos.

Con bloqueos retardados, los bloqueos se conservan durante periodos muy cortos, por lo cual la espera requerida se reduce. Si dos transacciones están procesando los mismos datos en paralelo, aquel que termina primera será el que realizará las modificaciones a la base de datos. La transacción más rápida no tendrá que esperar a que termine la más lenta.

La estrategia de bloqueos retardados es muy útil si el SGBD coloca los bloqueos a un nivel más alto que el de tupla. Dos usuarios procesando hileras distintas en la misma página o en la misma tabla, por ejemplo, pueden procesar en paralelo. En vista de que los bloqueos se conservan durante un periodo corto de tiempo, los usuarios minimizan sus interferencias. En la siguiente figura se muestra la misma situación que en la figura anterior. A diferencia del bloqueo implícito, prácticamente no existe ningún retardo.

El problema con los bloqueos retardados es que las transacciones pudieran necesitar procesarse dos y en ocasiones varias veces. Además la lógica es más complicada, y por lo tanto este método coloca una carga más pesada sobre el programador de la aplicación o la porción del SGBD, o sobre ambos.

0.19. Recuperación

La recuperación en sistemas cliente/servidor se puede llevar a cabo en la misma forma que en un sistema centralizado. La computadora servidor conserva un registro de las modificaciones en la base de datos, y la base de datos se almacena en algún dispositivo de almacenamiento. Cuando ocurre algún fallo, el SGBD del servidor puede deshacer la operación.

Instalación, Acceso, rutas

Para instalar PostgreSQL en un servidor Debian ejecutamos los siguientes comandos con privilegios de superusuario.

Empezamos actualizando nuestro índice de paquetes con APT:

```
apt update
```

Para instalar un servidor PostgreSQL con características adicionales a la base de datos ejecutamos:

```
apt install postgresql postgresql-contrib
```

Una vez que la instalación está completa, el servicio PostgreSQL empieza a trabajar. Para verificar la instalación, usamos la consola de Linux para mostrar la versión:

```
locate bin/postgres | xargs -i xargs -t '{}' -V
```

La salidad debe ser algo parecido a esto:

```
postgres (PostgreSQL) 9.6.15
```

0.20. psql

Para ingresar a la terminal psql, ejecutamos el siguiente comando:

```
psql -U nombreUsuario -W -h iphost nombreBaseDeDatos
```

Parámetros:

–U es el usuario de la base

–W mostrará el prompt de solicitud de password

–h IP del servidor de la base de datos en caso nos conectemos remotamente sino bastaría con poner localhost

Si hicimos la instalación por defecto, sólo tenemos el usuario postgres por lo tanto escribiríamos lo siguiente: psql –U postgres –W –h localhost presionamos enter y colocamos la contraseña que definimos al momento de la instalación.

psql es un programa de linea de comando o editor para interactuar con el servidor PostgreSQL.

Operaciones básicas en las Bases de datos y tablas.

Primero que nada veamos la siguiente lista de comandos básicos en postgresql.

Cuadro 1: Comandos para trabajar con psql.

Comando	Descripción
\l	muestra las bases de datos existentes.
\d	muestra las relaciones (tablas, secuencias, etc.) existentes en la base de datos.
\d [nombre_tabla]	Para ver la descripción (nombre de columnas, tipo de datos, etc.) de una tabla.
\c [nombre_bd]	Para conectarte a otra base de datos.
SHOW search_path;	Para ver la ruta de búsqueda actual.
SET search_path TO [nombre_esquema];	Para actualizar la ruta de búsqueda.
\q	Para salir de psql

0.21. Roles y métodos se autenticación

PostgreSQL gestiona los permisos de acceso a la base de datos utilizando el concepto de roles. Dependiendo de como configure el role, él representa un usuario o un grupo de la base de datos.

PostgreSQL soporta varios métodos de autenticación. los más frecuentes son:

Cuadro 2: Comandos para trabajar con psql.

Comando	Descripción
Trust	Un role puede conectarse sin password, citerio definido en pg_hba.conf.
Password	Un role puede conectarse con password. Éstas, son encriptadas como scram-sha-256, md5, y (clear-text)
Ident	Sólo soporta conexiones TCP/IP. Obtiene el nombre del usuario-cliente del sistema operativo, con la opción de mapear el nombre.
Peer	Igual que Ident, pero sólo soporta conexiones locales.

La autenticación de un cliente PostgreSQL es definida en el archivo de configuración pg_hba.conf. Para conexiones locales, PostgreSQL usa el método de autenticación peer.

El usuario ”postgres” es creado automáticamente durante la instalación PostgreSQL y es el superusuario.

Para ingresar en el servidor PostgreSQL como usuario ”postgres”, ejecutamos:

Figura 9: Pasos para ingresar en servidor PostgreSQL.

Para conocer el estatus del servidor PostgreSQL, escribimos en la consola:

```
postgres@jodocha:/home/jorge$ /etc/init.d/postgresql status
postgresql.service - PostgreSQL RDBMS
Loaded: loaded (/lib/systemd/system/postgresql.service; enabled; vendor
    preset: enabled)
Active: active (exited) since Sat 2019-11-02 03:08:08 -04; 4h 34min ago
Process: 4217 ExecStart=/bin/true (code=exited, status=0/SUCCESS)
Main PID: 4217 (code=exited, status=0/SUCCESS)
Tasks: 0 (limit: 4915)
CGroup: /system.slice/postgresql.service
```

Para detener el servidor, como superusuario (root) ejecutamos:

```
postgres@jodocha:/home/jorge$ /etc/init.d/postgresql stop
[ ok ] Stopping postgresql (via systemctl): postgresql.service.
```

Para iniciar o reiniciar el servidor, nuevamente como root ejecutamos:

```
postgres@jodocha:/home/jorge$ /etc/init.d/postgresql start
[ ok ] Starting postgresql (via systemctl): postgresql.service.
```

0.22. Creando roles y base de datos

El comando createuser crea nuevos roles desde la línea de comandos. Únicamente el superusuario y los roles con privilegio CREATEROLE crean nuevos roles.

En el siguiente ejemplo, creamos un nuevo rol llamado kylo, una base de datos llamada kylodb y otorgamos/grant privilegios sobre la base de datos al role.

Primero, creamos el rol con:

```
sudo su - postgres -c "createuser kylo"
```

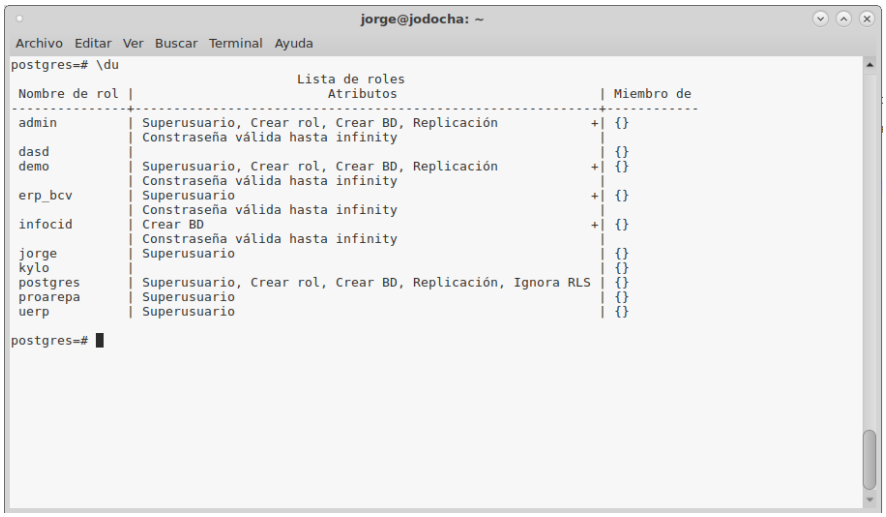


Figura 10: Listando los roles en servidor PostgreSQL.

Luego, creaamos la base de datos con el comando createdb:

```
sudo su - postgres -c "createdb kylodb"
```

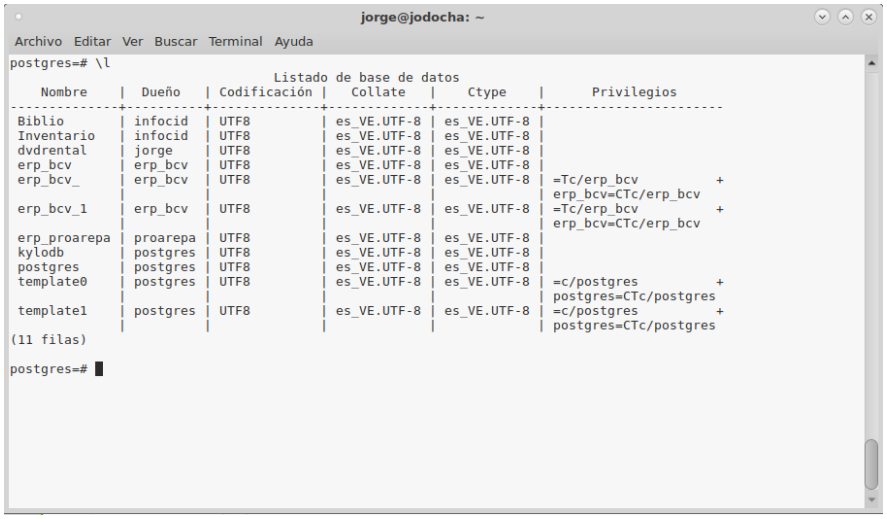


Figura 11: Listando las bases de datos en servidor PostgreSQL.

0.23. Otorgando/grant permisos

Ejecutamos la sentencia:

```
grant all privileges on database kylodb to kylo;
```

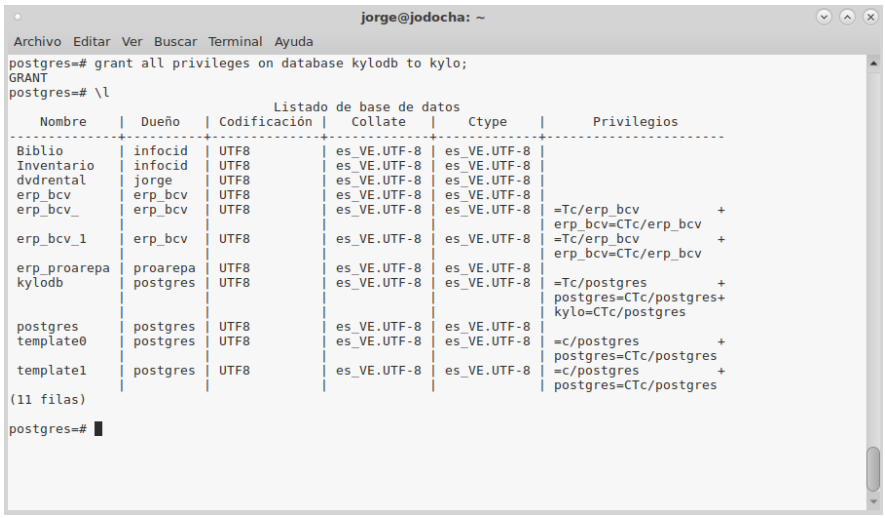


Figura 12: Listando las bases de datos en servidor PostgreSQL.

El usuario kylo tiene permisos sobre la base de datos kylodb, pero **NO** es el propietario.

0.24. Habilitando acceso remoto al servidor

Por defecto, el servidor PostgreSQL, escucha la interface local 127.0.0.1.

Si requerimos conectar PostgreSQL a una locación remota, necesitamos configurarlo para escuchar interfaces públicas, editando la configuración para aceptar conexiones remotas.

Abrimos el archivo de configuración postgresql.conf y, como superuser, agregamos listen_addresses = '*én la sección CONNECTIONS AND AUTHENTICATION.

0.25. Archivos de configuración

El comportamiento de PostgreSQL en nuestro sistema se puede controlar con tres archivos de configuración que se encuentran en el directorio de datos donde inicializamos nuestro cluster PostgreSQL (En nuestro caso /etc/postgresql/9.6/main/). Estos tres archivos son:

1. pg_hba.conf: Este archivo se utiliza para definir los diferentes tipos de accesos que un usuario tiene en el cluster.

2. pg_ident.conf: Este archivo se utiliza para definir la información necesaria en el caso que utilicemos un acceso del tipo ident en pg_hba.conf .

3. postgresql.conf: En este archivo podemos cambiar todos los parametros de configuracion que afectan al funcionamiento y al comportamiento de PostgreSQL en nuestra maquina.

```
1 nano /etc/postgresql/9.6/main/postgresql.conf
```

Vemos el siguiente contenido:

```
/etc/postgresql/11/main/postgresql.conf
#------------------------------------------------------------------------------

# CONNECTIONS AND AUTHENTICATION
#------------------------------------------------------------------------------


# - Connection Settings -

listen_addresses = '*'     # what IP address(es) to listen on;
```

Grabamos los cambios y reiniciamos el servicio PostgreSQL para que los cambios tengan efecto:

```
postgres@jodocha:/home/jorge$ /etc/init.d/postgresql restart
```

Verificamos los cambios con la utilidad ss:

```
ss -nlt | grep 5432

LISTEN   0         128                 0.0.0.0:5432              0.0.0.0:*
LISTEN   0         128                    [::]:5432                 [::]:*
```

Esta salida muestra que el servidor PostgreSQL escucha todas las intercaes (0.0.0.0).

El último paso es configurar el servidor para que acepte login remoto editando el archivo pg_hba.conf.

A continuación, algunos ejemplos:

```
/etc/postgresql/11/main/pg_hba.conf
# TYPE  DATABASE        USER            ADDRESS                 METHOD

# The user jane will be able to access all databases from all locations using an md5 password
host    all             jane            0.0.0.0/0               md5

# The user jane will be able to access only the janedb from all locations using an md5 password
host    janedb          jane            0.0.0.0/0               md5

# The user jane will be able to access all databases from a trusted location (192.168.1.134) without a password
host    all             jane            192.168.1.134           trust
```

¿Qué significa esto? Veamos columna a columna:

- Tipo: básicamente conexión local o conexión remota (host).
- Base de datos: base de datos a las que afecta la regla. Si queremos todas, usamos el comodín all.
- Usuario: usuarios a los que afecta la regla, si queremos que afecte a todos, usamos también all.

Nos paramos ahora en los dos apartados que más atención requieren.

0.26. Dirección/Address

En esta columna definimos, las direcciones IP (podemos también usar IPv6), desde las que podremos conectarnos a PostgreSQL. Usaremos la fórmula dirección/máscara:

```
Una sola dirección: 150.100.100.100/32
O un rango (ampliemos el mismo de antes): 150.100.100.0/24 (256
    direcciones)
```

0.27. Método

Aunque hay multitud de métodos para utilizar (incluyendo conexiones LDAP, Kerberos o PAM), explico los tres más básicos:

- ident: utiliza el usuario del sistema desde el que se está intentado conectar.
- trust: deja todos los accesos sin necesidad de autenticarse (sólo recomendable para conexiones desde el equipo local).
- password: identificación con usuario/contraseña, es la más típica y es la recomendable para conexiones desde clientes como EMS PostgreSQL Manager.

Una línea de ejemplo, para darle acceso a todos los usuarios, a todas las base de datos, desde el rango de IP explicado antes, usando autenticación con usuario y contraseña, sería la siguiente:

```
host    all          all          150.100.100.0/24          password
```

0.28. Habilitar conexiones al socket desde clientes que no sean el host local

Para versiones PostgreSQL 8.x en adelante el procedimiento es el siguiente. Buscamos el archivo postgresql.conf en:

- /etc/postgresql/9.6/main/postgresql.conf (en debian 9.11)
- /var/lib/pgsql/data/postgresql.conf

Y buscar dentro del mismo la siguiente línea:

```
listen_addresses='localhost'
```

Para sustituirla, por el comodín (para todas las IP, es una opción segura, tomar en cuenta que tenemos un filtro en el archivo

```
pg_hba.conf
listen_addresses='*'
```

O definir algunas direcciones IP en concreto:

```
listen_addresses='150.100.100.100 150.100.100.101'
```

0.29. Reiniciar servicio

Reiniciamos el servicio para que el servidor cargue los nuevos valores (como root):

```
service postgresql restart
```

0.30. Abrir entrada en iptables

Finalmente, nos aseguramos de que iptables no corta la entrada (el puerto de PostgreSQL es el 5432 por defecto, si lo cambia, cambia aquí también):

```
iptables -A INPUT -p tcp -s 0/0 --sport 1024:65535 -d ip_cliente \
 --dport 5432 -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -p tcp -s ip_cliente --sport 5432 -d 0/0 \
 --dport 1024:65535 -m state --state ESTABLISHED -j ACCEPT
```

0.31. Probar

Desde un cliente gráfico o desde línea de comandos, probamos la configuración (suponemos que desde el equipo al que estamos dando acceso):

```
psql -h ip_servidor -U miusuario
```

Y con esto debería ir todo sin problemas. Las posibilidades de PostgreSQL son enormes, empezando por este tipo de configuraciones.

Al termino de este proceso, podemos entrar según:

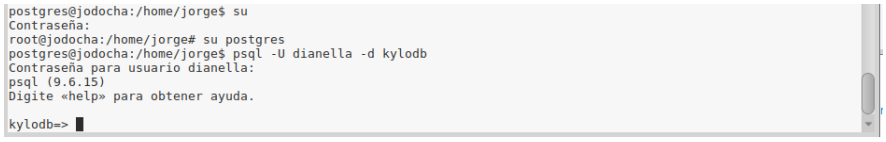


Figura 13: Listando las bases de datos en servidor PostgreSQL.

Administrador, usuarios y grupos

Cómo crear un usuario y asignarle permisos en PostgreSQL.

Capítulo dedicado a crear usuarios, crear bases de datos con un usuario específico como propietario y asignación de permisos a una base de datos.

Algo muy importante en cualquier motor de base de datos son los privilegios de usuarios, veamos a continuación algunas de las opciones posibles.

Veamos las etapas a seguir:

0.32. Acceder al sistema

Para acceder al servidor, luego de su instalación, abrimos la consola del computador o nos conectamos vía ssh.

```
1 \$ ssh miusuario@miservidorpgsql
```

0.33. Autenticarnos como usuario postgres

Una vez dentro del sistema, ingresamos como root, por seguridad y, luego, ingresar como usuario postgres:

```
1 su
2 clave de root
3 su postgres
```

0.34. Abrir un cliente de PostgreSQL

Iniciamos una sesión cliente en un servidor. Para esto, ejecutamos los siguientes comandos:

```
1 su
2 clave de root
3 su postgresql
4 psql -U postgres -h localhost -W
```

0.34.1. Crear usuario

Para crear un usuario, dentro del editor psql, ejecutamos el siguiente comando:

```
1 CREATE USER dianella PASSWORD 'password';
```

La figura muestra

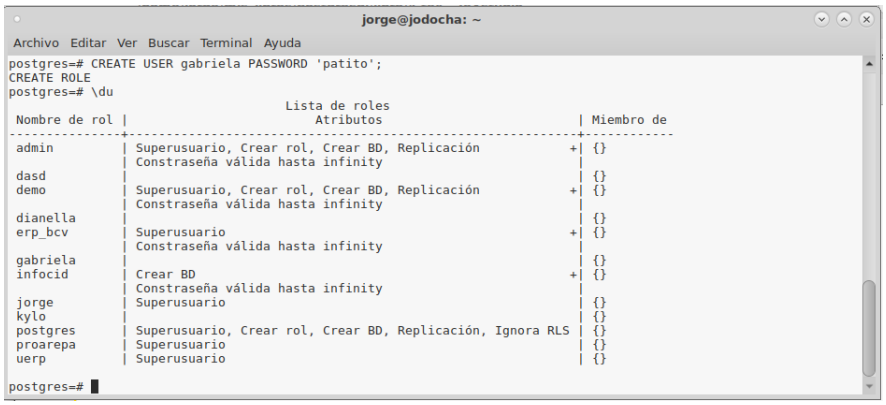


Figura 14: Lista de roles en servidor PostgreSQL.

Vemos que tenemos dos roles, dianella y kylo, en sus atributos carecen de clave y no tienen privilegios. Si creamos el rol gabriela con la misma clave de dianella, no pasa nada, porque para identificar al rol, PostgreSQL asocia nombre+clave.

0.34.2. Eliminar usuario

Para eliminar un usuario ejecutamos el siguiente comando:

```
1 DROP USER nanotutoriales;
```

Eliminamos usuario gabriela:

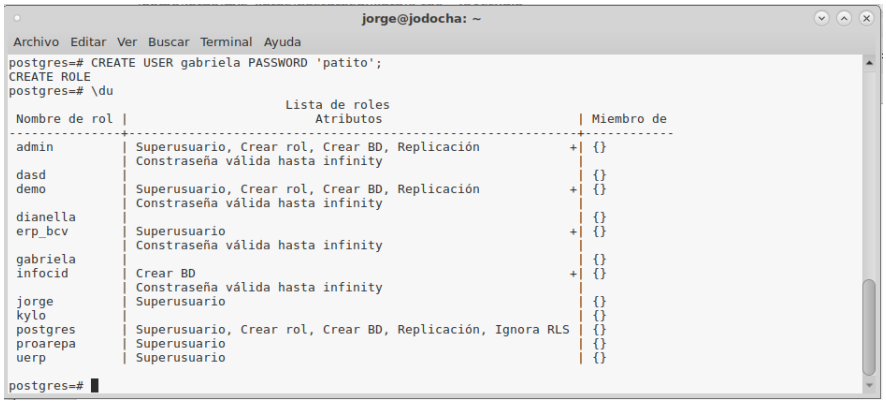


Figura 15: Lista de roles en servidor PostgreSQL.

```
1 ALTER ROLE
```

El manejo de roles en PostgreSQL permite diferentes configuraciones, entre ellas están:

Opción	Descripción
SUPERUSER/NOSUPERUSER	Super usuario, privilegios para crear bases de datos y usuarios.
CREATEDB / NOCREATEDB.	Permite crear bases de datos.
CREATEROLE / NOCREATEROLE.	Permite crear roles.
CREATEUSER / NOCREATEUSER.	Permite crear usuarios.
LOGIN / NOLOGIN.	Este atributo hace la diferencia entre un rol y usuario. Ya que el usuario tiene permisos para acceder a la base de datos a traves de un cliente.
ENCRYPTED.	graba la contraseña encriptada según método por defecto.
PASSWORD.	Permite alterar la contraseña.
VALID UNTIL.	Expiración de usuarios.

Para cambiar la configuracion de un usuario o rol debemos ejecutar el siguiente comando.

```
1 ALTE ROLE <nombre del rol> WITH <opciones>
```

Asignar permisos de super usuario a un usuario El permiso de super usuario es el mas alto. Con él se administran los objetos del motor de base de datos.

Para asignar este privilegio a un rol lo hacemos con el siguiente comando:

```
1 ALTER ROLE nanotutoriales WITH SUPERUSER;
```

Cambiar la contraseña de un usuario.

Para cambiar la contraseña de un usuario es necesario ejecutar el siguiente comando:

```
1 ALTER ROLE dianella WITH PASSWORD 'jorge';
```

Cambiar la contraseña de un usuario.

Cambiar propietario de una base de datos con un usuario específico.

Todas las bases de datos que creamos con un usuario que tenga los privilegios CREATEDB automáticamente asignan como propietario al usuario mismo. Si lo que queremos crear es un usuario limitado, la forma de crearlo con una base de datos específica será:

```
1 ALTER DATABASE kylodb OWNER TO dianella;
```

Verificamos que los cambias han tomado fecto con el comando \l.

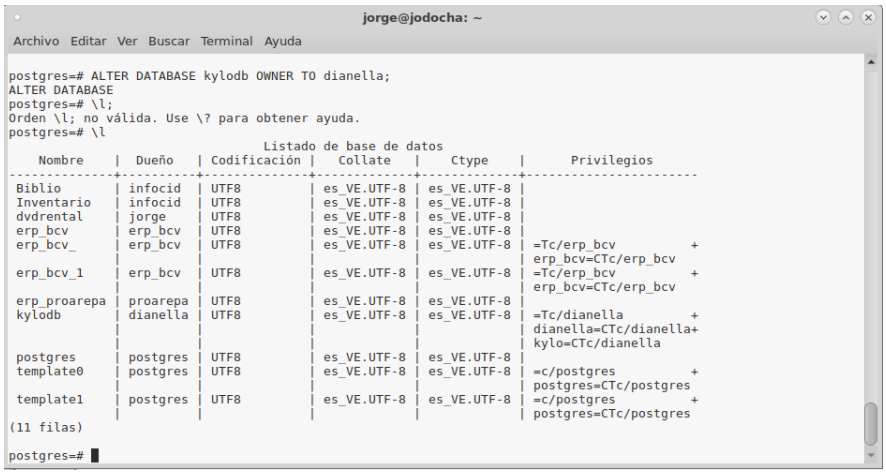


Figura 16: Lista de roles en servidor PostgreSQL.

Asignar todos los permisos a un usuario a una base de datos existente.

Cuando recien hemos creado un usuario y queremos darle permisos a una base de datos existente, podemos utilizar el siguiente comando:

```
1 GRANT ALL PRIVILEGES ON DATABASE kylodb TO dianella;
```

Asignar todos los permisos a un usuario a una base de datos existente

0.35. 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 ob- jeto 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, execu- te, usage, y all privileges. Presentamos algunas sentencias de trabajo con privilegios, que siguen al pie de la letra el estándar SQL:

```
1 grant all privileges on proveedores to marc;
2 grant select on precios to manuel;
3 grant update on precios to group migrupo;
4 revoke all privileges on precios to manuel;
5 grant select on ganacias from public;
```

0.35.1. FATAL: la autentificación Peer falló para el usuario dianella

Si requerimos entrar en kylodb como usuario dianella, ejecutamos:

```
1 postgres=# \c kylo dianella;
2 FATAL:  la autentificación Peer falló para el usuario dianella
3 Se ha mantenido la conexión anterior
```

De manera predeterminada PostgreSQL en GNU/Linux Debian, deja habilitado el mecanismo de autentificación PEER para conexiones locales, por lo tanto si intenta conectarse desde la sesión de un usuario del sistema operativo, que no es usuario de base de datos, obtendrá un error de autentificación. (psql: FATAL: la autentificación Peer falló para el usuario postgres), porque psql está obteniendo el nombre del usuario desde el propio Kernel para luego usarlo con nombre de usuario de base de datos al momento de intentar la autentificación.

0.35.1.1. Primer forma de arreglarlo

La primer solución no requiere cambiar nada. Solo consiste en utilizar la autentificación correctamente, iniciando sesión con postgres a nivel de sistema operativo y luego intentar autenticarse en psql.

```
usuario@laptop:~$ su postgres
Contraseña:
usuario@laptop:/home/usuario$ psql -d mi_basedatos
psql (9.4.0)
Digite help para obtener ayuda.

mi_basedatos=#
```

0.35.1.2. La segunda forma de arreglarlo

La otra posibilidad es cambiar la configuración de autentificación, cambiándola de PEER a MD5, en el archivo de configuración de PostgreSQL.

Es muy recomendable hacer una copia del archivo de configuración antes de modificarlo, para hacer la copia introduzca el siguiente comando:

```
cp /etc/postgresql/9.4/main/pg_hba.conf /etc/postgresql/9.4/main/pg_hba
   .conf_bk
```

Con el siguiente comando se hace la modificación automáticamente, pero tiene que copiar y pegar el comando con todos los espacios en blanco, tal y cómo está a continuación:

```
$ sed -i -e 's/local   all             postgres
                                    peer/local   all             postgres
                                     md5/g' /etc/postgresql/9.4/main/
   pg_hba.conf
```

Para comprobar que el cambio se aplicó correctamente introduzca el siguiente comando:

```
$ diff /etc/postgresql/9.4/main/pg_hba.conf /etc/postgresql/9.4/main/
   pg_hba.conf_bk
85c85
< local   all             postgres                                md5
---
> local   all             postgres                                peer
```

Si el cambio no es correcto, debe editar el archivo manualmente cambiando "peer"por "md5.[en] la linea que contiene el siguiente texto:

```
# Database administrative login by Unix domain socket
local   all             postgres                                peer
```

Debe quedar así:

```
# Database administrative login by Unix domain socket
local   all             postgres                                md5
```

Para editar el archivo introduzca el siguiente comando:

```
$ nano /etc/postgresql/9.4/main/pg_hba.conf
```

Finalmente, ya sea que haya hecho el cambio automáticamente o editando el archivo de configuración manualmente, deberá reiniciar el servicio de postgres para que los cambios surtan efectos, para hacerlo introduzca el siguiente comando:

```
$ /etc/init.d/postgresql restart
[sudo] password for usuario:
[ ok ] Restarting postgresql (via systemctl): postgresql.service.
```

Ahora si podrá autentificarse en psql aunque el usuario del sistema operativo no sea usuario de base de datos. Tome en cuenta que siempre deberá indicar el usuario con el que intentará iniciar sesión, con el parámetro -U

```
usuario@laptop:~$ psql -U postgres -d mi_basedatos
Contraseña para usuario postgres:
psql (9.4.0)
Digite help para obtener ayuda.
mi_basedatos=#
```

confianza
Permitir la conexión incondicionalmente. Este método permite a cualquier persona que pueda conectarse al servidor de base de datos PostgreSQL iniciar sesión como cualquier usuario de PostgreSQL que desee, sin la necesidad de una contraseña o cualquier otra autenticación. Vea la Sección 19.3.1 para más detalles.
rechazar
Rechaza la conexión incondicionalmente. Esto es útil para "filtrarçiertos hosts de un grupo, por ejemplo, una línea de rechazo podría bloquear la conexión de un Host específico, mientras que una línea posterior permite que los hosts restantes en una red específica se conecten.
md5
Exigir al cliente que proporcione una contraseña doble hash MD5 para la autenticación. Vea la Sección 19.3.2 para más detalles.
contraseña
Exigir al cliente que proporcione una contraseña sin cifrar para la autenticación. Dado que la contraseña se envía en texto sin cifrar a través de la red, no debe utilizarse en redes que no sean de confianza. Vea la Sección 19.3.2 para más detalles.
gss
Utilice GSSAPI para autenticar al usuario. Esto solo está disponible para conexiones TCP/IP. Vea la Sección 19.3.3 para más detalles.
sspi
Utilice SSPI para autenticar al usuario. Esto solo está disponible en Windows. Vea la Sección 19.3.4 para más detalles.
ident
Obtenga el nombre de usuario del sistema operativo del cliente poniéndose en contacto con el servidor de identificación en el cliente y verifique si coincide con el nombre de usuario de la base de datos

solicitada. La autenticación de identidad solo se puede utilizar en conexiones TCP/IP. Cuando se especifique para las conexiones locales, se utilizará la autenticación de pares en su lugar. Vea la Sección 19.3.5 para más detalles.

par

Obtenga el nombre de usuario del sistema operativo del cliente del sistema operativo y verifique si coincide con el nombre de usuario de la base de datos solicitado. Esto solo está disponible para conexiones locales. Vea la Sección 19.3.6 para más detalles.

ldap

Autenticar utilizando un servidor LDAP. Vea la Sección 19.3.7 para más detalles.

radio

Autentíquese usando un RADIUS servidor. Vea la Sección 19.3.8 para más detalles.

cert

Autenticar utilizando certificados de cliente SSL. Vea la Sección 19.3.9 para más detalles.

pam

Realice la autenticación utilizando el servicio de módulos de autenticación conectables (PAM) proporcionado por el sistema operativo. Vea la Sección 19.3.10 para más detalles.

Seguridad

0.36. SCHEMAS

Un esquema es esencialmente un espacio de nombres: contiene el nombre de objetos (tablas, tipos de datos, funciones y operadores), cuyos nombres pueden duplicar los de otros objetos existentes en otros esquemas. Los objetos con nombre se accede bien por çalificación"de sus nombres con el nombre de esquema como prefijo, o mediante el establecimiento de una ruta de búsqueda que incluye el esquema deseado.

Un comando CREATE especifica un nombre de objeto no calificado, crea el objeto en el esquema actual.

Funcionalidad básica:

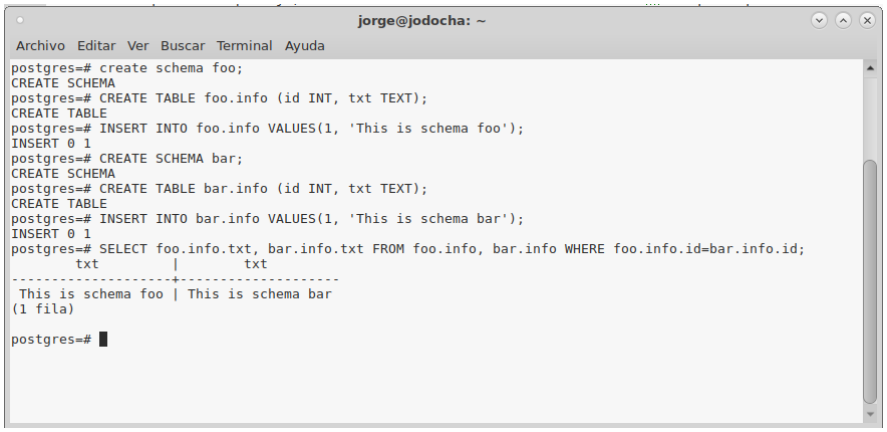


Figura 17: Trabajando con schemas.

Cambiar de schema:

```
1 test=# SET search_path TO foo;
2 SET
```

Para cambiar permanentemente el schema en cada conexión:

```
1 ALTER USER dianella SET search_path TO bar,foo;
```

El cambio surtirá efecto solamente después de conectarse de nuevo.

```
1 test=# SET search_path TO bar, foo;
2 SET
```

```
test=# SELECT txt FROM info;
txt
--------------------
This is schema bar
(1 row)
test=# SELECT * FROM info_view;
ERROR: Relation "info_view" does not exist
test=# SELECT * FROM public.info_view;
foo | bar
--------------------+--------------------
This is schema foo | This is schema bar
```

0.37. PERMISOS Y SEGURIDAD

Los esquemas solamente son creados por los superusuarios.

0.37.1. Crear esquemas

Para crear un esquema para otro usuario utilizamos:

```
CREATE SCHEMA tarzan AUTHORIZATION tarzan;
o
CREATE SCHEMA AUTHORIZATION tarzan;
```

El rol tarzan debe existir.

0.37.2. Eliminar esquemas

```
DROP SCHEMA tarzan;
```

Si existen datos en el esquema tarzan, utilizamos:

```
DROP SCHEMA tarzan CASCADE;
```

0.37.3. Funciones de schemas

```
select current_schema();
```

Retorna el nombre del esquema actual.

```
\dn;
```

Retorna todos los esquemas en search path

```
SET search_path TO
PUBLIC ,ASIS,CLIE,CONT,CRED,INVE,MASI,MISC,MUTU,PRES,RRHH,SEGU,SERV,
    SISE,TESO;
```

0.37.4. Limitaciones

no es posible transferir objetos entre esquemas, para ello:

```
CREATE TABLE new_schema.mytable AS SELECT * FROM old_schema.mytable
INSERT INTO new_schema.mytable SELECT * FROM old_schema.mytable;
```

0.38. Roles y permisos

En PostgreSQL, los roles son objetos globales que puede acceder a todas las bases de datos de cluster (contando con los privilegios adecuados). Los roles están completamente separados de los usuarios a nivel sistema operativo, aunque es conveniente mantener una correspondencia entre los mismos.

A fin de inicializar un sistema de bases de datos, cada instalación fresca siempre contiene un rol predefinido. Este rol es siempre un superusuario, y tiene el mismo nombre (a menos que sea cambiado cuando se corre initdb) que el del usuario (a nivel sistema operativo) que inicializó el cluster de bases de datos. Como habrán notado en artículos anteriores, habitualmente se utiliza el nombre de usuario "postgres"para dicho rol. Los roles determinan el conjunto de privilegios disponibles a un cliente conectado.

Cada conexión a un servidor de base de datos se realiza utilizando el nombre de un rol en particular. Este rol determina los privilegios de acceso iniciales para los comandos ejecutados en esa conexión. El nombre del rol para usar en una conexión en particular es indicado por el cliente que la inicia. Por ejemplo, el cliente psql dispone de la opción de línea de comandos -U para indicar el rol a utilizar. Otras aplicaciones asumen el nombre del usuario a actual (a nivel sistema operativo) como rol a utilizar. Esta es la razón por la cual es conveniente mantener una correspondencia entre nombres de usuario a nivel sistema operativo y roles.

Existe un pseudo-rol (más bien palabra clave) PUBLIC que puede pensarse como un grupo que almacena a todos los roles. Cuando se otorga un permiso sobre el rol PUBLIC, se otorga a todos los roles. PostgreSQL otorga privilegios por defecto a PUBLIC sobre algunos tipos de objetos. No se otorgan privilegios a PUBLIC por defecto sobre tablas, columnas, schemas o tablespaces (espacios de almacenamiento para tablas en el sistema de archivos). Los privilegios por defecto otorgados a PUBLIC son: CONNECT y CREATE TEMP TABLE para bases de datos; EXECUTE para funciones; y USAGE para lenguajes.

Una forma conveniente de trabajo consiste en otorgar permisos a un rol de grupo, y hacer a los usuarios miembros de dicho grupo. De esta forma se simplifica la gestión de permisos, pues es más simple lidiar con la granularidad de altas (GRANT) y bajas (REVOKE) de privilegios. Un rol de grupo es un rol común y corriente sin posibilidad de login (si se le otorga login se convierte en un rol de usuario).

Un cluster PostgreSQL contener múltiples bases de datos, que a su vez pueden contener varios schemas (colección de tablas). Los schemas son individuales a cada base de datos, y no tienen relación con otros schemas de otras bases en el mismo cluster. Por defecto, cada base de datos inicia con un schema "public". Este schema no tiene nada en particular, sólo que existe por defecto. Cuando se otorga un permiso sobre un schema, sólo aplica a éste schema en particular sobre la base de datos actual (al momento de ejecutar el GRANT). Una tabla puede ser referenciada por su nombre calificado (schema.tabla) o simplemente utilizando el nombre de la tabla (nombre sin calificar). Cuando se utilizan nombres sin calificar, el sistema necesita determinar a qué tabla se refiere examinando los schemas en una lista (search path).

Habiendo asimilado todos estos conceptos, veamos como obtener e interpretar esta información desde línea de comandos, utilizando el cliente psql.

0.39. Conectarse a un servidor PostgreSQL con psql

Algunas opciones típicas para conectarse a un servidor de bases de datos PostgreSQL con psql son el rol (usuario), host y puerto. Como:

```
$ psql -U postgres -h localhost -p 5432
```

0.40. Conectarse a una base de datos

El cliente psql posee una amplia variedad de subcomandos (es posible listarlos utilizando el subcomando \?). Para conectarse a una base de datos se utiliza el subcomando \c, por ejemplo:

```
postgres=# \c sghe4_db7
You are now connected to database "sghe4_db7" as user "postgres".
sghe4_db7=#
```

0.41. ¿Qué rol posee un usuario?

Es posible listar usuarios/roles utilizando el subcomando \du (o \dg):

```
sghe4_db7=# \du+
List of roles
Role name  |                         Attributes                         |
   Member of        | Description
--
   ----------+------------------------------------------------------------+--------------------

abot       |                                                            | {role_ro}
                 |
lenny      |                                                            | {
   role_devel}          |
homer      |                                                            | {role_ro}
                 |
carl       |                                                            | {
   role_devel}          |
postgres   | Superuser, Create role, Create DB, Replication | {}
                         |
role_prod  | Cannot login                                               | {}
                         |
role_devel | Cannot login                                               | {
   role_prod,role_bckp} |
role_ro    | Cannot login                                               | {}
                         |
role_bckp  | Cannot login                                               | {}
                         |
sghe_admin | Superuser, Create DB                                       | {}
                         |
willy      |                                                            | {
   role_devel}          |
```

Para diferenciar usuarios de roles, basta notar que los roles no tienen login (Çannot login"). Por ejemplo ”lenny” es un usuario y ”role_ro” un rol.

Notar además que esta tabla indica a qué rol pertenece cada usuario. Por ejemplo "lenny"pertenece al rol ”role_devel”.

0.42. ¿A qué schema pertenece una tabla?

La tabla ”tables” de la base de datos ”information_schema” posee información acerca de todas las tablas de todos los esquemas en un servidor de bases de datos. La siguiente consulta SQL permite saber a qué schema pertenece una tabla en particular (a modo de ejemplo, la tabla ”kevsigqpbh_fh234512”):

```
sghe4_db7=# select table_catalog,table_schema,table_name from
   information_schema.tables where table_name='kevsigqpbh_fh234512';
table_catalog | table_schema |     table_name
---------------+--------------+---------------------
sghe4_db7     | public       | kevsigqpbh_fh234512
(1 row)
```

Se observa que la tabla "kevsigqpbh_fh234512"pertenece al schema "public".

0.43. Listar la ACL de una base de datos

La ACL (lista de control de acceso) de una base de datos define los usuarios que pueden acceder a la misma, y con qué rol.

```
sghe4_db7=# select datname,datacl from pg_database where datname='
   sghe4_db7';
datname   |                            datacl
--
   ----------+-------------------------------------------------------------------


sghe4_db7  | {=Tc/postgres,postgres=CTc/postgres,role_ro=Tc/postgres}
(1 row)
```

Para interpretar una lista de control de acceso es necesario leer detenidamente la página del manual del comando GRANT.

Esta ACL en particular se interpreta de la siguiente forma:

```
=Tc/postgres: por defecto (cuando el cliente no especifica un usuario
   al momento de conectarse) se accede como el rol "postgres" y sólo se
    posee el permiso de conexión a la base de datos, es decir,
   cualquiera se puede conectar a la base de datos.
postgres=CTc/postgres: el usuario "postgres" posee permisos de conexión
    y creación de tablas.
role_ro=Tc/postgres: el rol "rol_ro" sólo tiene permiso de conexión a
   la base de datos.
También es posible obtener la lista de control de acceso de cada base
   de datos recurriendo al subcomando \l:
\begin{lstlisting}
sghe4_db7=# \l
List of databases
```

```
Name    |  Owner   | Encoding |   Collate   |    Ctype    |   Access
   privileges
--
   ----------+----------+----------+-------------+-------------+------------------
sghe4_db5  | postgres | UTF8     | es_AR.UTF-8 | es_AR.UTF-8 |
sghe4_db6  | postgres | UTF8     | es_AR.UTF-8 | es_AR.UTF-8 |
sghe4_db7  | postgres | UTF8     | es_AR.UTF-8 | es_AR.UTF-8 |
postgres   | postgres | UTF8     | es_AR.UTF-8 | es_AR.UTF-8 |
template0  | postgres | UTF8     | es_AR.UTF-8 | es_AR.UTF-8 | =c/
   postgres          +
|          |          |          |             | postgres=CTc/
   postgres
template1  | postgres | UTF8     | es_AR.UTF-8 | es_AR.UTF-8 | postgres
   =CTc/postgres+
|          |          |          |             | =c/postgres
(6 rows)
```

0.43.1. Listar los privilegios sobre los schemas de una base de datos

El subcomando \dn del cliente psql se utiliza para listar rápidamente todos los schemas de una base de datos. Cuando se agrega el modificador + adicionalmente lista los privilegios (ACL) sobre cada uno:

```
sghe4_db7=# \dn+
List of schemas
Name         |   Owner    |      Access privileges       |
   Description
--
   -----------------+------------+------------------------------+-----------------
audit              | postgres   | postgres=UC/postgres        +|
|            | sghe_admin=UC/postgres      +|
|            | role_ro=U/postgres          +|
|            | role_prod=U/postgres        +|
|            | role_devel=UC/postgres       |
content            | postgres   | postgres=UC/postgres        +|
|            | sghe_admin=UC/postgres      +|
|            | role_ro=U/postgres          +|
|            | role_prod=U/postgres        +|
|            | role_devel=UC/postgres       |
cache              | postgres   | postgres=UC/postgres        +|
|            | sghe_admin=UC/postgres      +|
|            | role_ro=U/postgres          +|
|            | role_prod=U/postgres        +|
|            | role_devel=UC/postgres       |
public             | postgres   | postgres=UC/postgres        +|
   standard public schema
|            | sghe_admin=UC/postgres      +|
|            | role_ro=U/postgres          +|
|            | role_prod=U/postgres        +|
|            | role_devel=UC/postgres       |
```

```
tmp_tables          | sghe_admin | sghe_admin=UC/sghe_admin    +|
|            | role_devel=UC/sghe_admin    +|
|            | role_bckp=U/sghe_admin       |
(5 rows)
```

0.43.2. Listar permisos sobre cada tabla/vista

Ya hemos visto cómo listar los permisos sobre cada base de datos y cada schema, sólo resta conocer los permisos sobre cada tabla. Para ello, se puede recurrir al subcomando \dp

```
sghe4_db7=# \dp+ imp*
Access privileges
Schema |       Name        | Type  |      Access privileges       |
   Column access privileges
--
   ------+-------------------+-------+------------------------------+-------------


public | imp_class         | table | role_prod=arwd/postgres     +|
|                   |       | sghe_admin=arwdDxt/postgres +|
|                   |       | role_devel=r/postgres       +|
|                   |       | role_devel=arwdDxt/postgres  |
public | imp_messages      | table | role_prod=arwd/postgres     +|
|                   |       | sghe_admin=arwdDxt/postgres +|
|                   |       | role_ro=r/postgres          +|
|                   |       | role_devel=arwdDxt/postgres  |
public | imp_mensaje_types | table | role_prod=arwd/postgres     +|
|                   |       | sghe_admin=arwdDxt/postgres +|
|                   |       | role_ro=r/postgres          +|
|                   |       | role_devel=arwdDxt/postgres  |
(3 rows)
```

0.43.3. Listar permisos efectivos sobre tablas

La tabla "table_privileges"de la base de datos ïnformation_schema.ªlmacena toda la información de privilegios sobre todas las tablas.

Es posible listar todos los permisos sobre una tabla específica utilizando la siguiente consulta SQL. Por ejemplo, para la tabla "kevsigqpbh_fh234512":

```
sghe4_db7=# select * from information_schema.table_privileges where
   table_name='kevsigqpbh_fh234512';
grantor  |  grantee   | table_catalog | table_schema |     table_name
        | privilege_type | is_grantable | with_hierarchy
--
   --------+------------+---------------+--------------+--------------------+-----


---------
postgres | role_prod  | sghe4_db7     | public       |
   kevsigqpbh_fh234512 | INSERT         | NO           | NO
postgres | role_prod  | sghe4_db7     | public       |
   kevsigqpbh_fh234512 | SELECT         | NO           | NO
postgres | role_prod  | sghe4_db7     | public       |
   kevsigqpbh_fh234512 | UPDATE         | NO           | NO
```

```
postgres | role_prod  | sghe4_db7    | public       |
   kevsigqpbh_fh234512 | DELETE         | NO           | NO
postgres | sghe_admin | sghe4_db7    | public       |
   kevsigqpbh_fh234512 | INSERT         | YES          | NO
postgres | sghe_admin | sghe4_db7    | public       |
   kevsigqpbh_fh234512 | SELECT         | YES          | NO
postgres | sghe_admin | sghe4_db7    | public       |
   kevsigqpbh_fh234512 | UPDATE         | YES          | NO
postgres | sghe_admin | sghe4_db7    | public       |
   kevsigqpbh_fh234512 | DELETE         | YES          | NO
postgres | sghe_admin | sghe4_db7    | public       |
   kevsigqpbh_fh234512 | TRUNCATE       | YES          | NO
postgres | sghe_admin | sghe4_db7    | public       |
   kevsigqpbh_fh234512 | REFERENCES     | YES          | NO
postgres | sghe_admin | sghe4_db7    | public       |
   kevsigqpbh_fh234512 | TRIGGER        | YES          | NO
postgres | role_ro    | sghe4_db7    | public       |
   kevsigqpbh_fh234512 | SELECT         | NO           | NO
postgres | role_devel | sghe4_db7    | public       |
   kevsigqpbh_fh234512 | INSERT         | NO           | NO
postgres | role_devel | sghe4_db7    | public       |
   kevsigqpbh_fh234512 | SELECT         | NO           | NO
postgres | role_devel | sghe4_db7    | public       |
   kevsigqpbh_fh234512 | UPDATE         | NO           | NO
postgres | role_devel | sghe4_db7    | public       |
   kevsigqpbh_fh234512 | DELETE         | NO           | NO
postgres | role_devel | sghe4_db7    | public       |
   kevsigqpbh_fh234512 | TRUNCATE       | NO           | NO
postgres | role_devel | sghe4_db7    | public       |
   kevsigqpbh_fh234512 | REFERENCES     | NO           | NO
postgres | role_devel | sghe4_db7    | public       |
   kevsigqpbh_fh234512 | TRIGGER        | NO           | NO
(19 rows)
```

Esta es una tabla muy completa que indica quién otorgó qué permiso a qué usuario sobre cada tabla, perteneciente a qué esquema de qué base de datos. También existe la vista role_table_grants", la cual presenta la misma información exeptuando todos los permisos otorgados al usuario actual a través de grants a PUBLIC.

0.44. 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 archivo las sentencias SQL necesarias para recrear las bases de datos.
- Haciendo copia a nivel de archivo 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 > archivo_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.

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

```
$ psql demo < archivo_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.

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 archivo, simplemente copiamos los archivos 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 archivos 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.

0.45. 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 archivos de registro.

0.45.1. 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
```

0.45.2. 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;
```

0.45.3. Archivos 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 archivos de registro; es decir, que cada cierto tiempo (12 horas, un día, una semana...), se haga una copia de estos archivos y se empiecen unos nuevos, lo que nos permitirá mantener un histórico de éstos (tantos como archivos 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.

Ajustes básicos

Luego de compilar e instalar un servidor de bases de datos Postgres, el siguiente paso luego de inicializar una instancia consiste en configurar el demonio o servicio. Este capítulo explica las opciones de configuración básicas de un servidor de bases de datos PostgreSQL.

Suponiendo que se ha creado una nueva instancia de PostgreSQL en el directorio /usr/local/postgres/pg_nuevo (directorio de datos de la instancia de Postgres), editar el archivo de configuración de la misma (postgresql.conf):

```
# nano /usr/local/postgres/pg_nuevo/postgresql.conf
```

El archivo postgresql.conf define los parámetros de configuración del servidor PostgreSQL (es una especie de análogo a my.cnf en servidores MySQL). Este archivo es creado por la herramienta initdb al momento de crear la instancia.

Los parámetros de configuración básicos para toda instancia de Postgres son los siguientes:

```
data_directory: directorio de datos de la instancia. Típicamente es el
   directorio donde se encuentra el archivo postgresql.conf, aunque es
   posible ubicarlo en cualquier otro directorio.
data_directory = '/usr/local/postgres/pg_nuevo'
external_pid_file: archivo donde guardar el ID de proceso de la
   instancia.
external_pid_file = '/usr/local/postgres/pg_nuevo/postmaster.pid'
listen_addresses: direcciones IP donde escuchar peticiones.
listen_addresses = 'localhost,192.168.63.42'
port: puerto donde escuchar peticiones (por defecto 5432).
port = 5432
max_connections: máxima cantidad de clientes de manera concurrente
   permitida
max_connections = 100
ssl_*: opciones de SSL. Si se utiliza SSL (recomendado), estas opciones
    definen las suites de cifrado permitidas y las rutas al certificado
    y su clave privada.
ssl = on
ssl_ciphers = 'ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH'
ssl_cert_file = 'server.crt'
ssl_key_file = 'server.key'
shared_buffers: tamaño de la memoria utilizada para caché. A partir de
   1 GB, el valor aconsejado es a partir de 1/4 de la memoria RAM (por
   ejemplo 128 MB). En sistemas con menor cantidad de memoria se
   recomienda utilizar un 15% de la misma.
shared_buffers = 24MB
log_*: opciones de logging (registro de eventos). Estas opciones
```

```
    determinan dónde se guardan los archivos de log y su esquema de
    rotación (en este ejemplo cada 180 días o 100 MB).
logging_collector = on
log_directory = 'pg_log'
log_filename = 'pg_nuevo%Y%m%d%H%M%S.log'
log_truncate_on_rotation = on
log_rotation_age = 180d
log_rotation_size = 100MB
log_duration = on
log_line_prefix = '%t %u %c %d %p %i'
log_statement = 'all'
```

La opción log_line_prefix determina el prefijo de cada entrada en el log: %t indica el timestamp, %u el usuario, %c el ID de sesión, %d el nombre de la base de datos, %p el ID de proceso e %i el tag asociado al comando ejecutado.

Opciones de localización (configuradas por initdb):

```
datestyle = 'iso, dmy'
lc_messages = 'es_AR.UTF-8'
lc_monetary = 'es_AR.UTF-8'
lc_numeric = 'es_AR.UTF-8'
lc_time = 'es_AR.UTF-8'
default_text_search_config = 'pg_catalog.spanish'
```

0.46. Consultar el tamaño de bases de datos, tablas y objetos

Consultar el tamaño de las bases de datos:

```
SELECT
pg_database.datname,
pg_size_pretty(pg_database_size(pg_database.datname)) AS size
FROM pg_database;
```

Consultar el Top10 de las tablas de una base de datos que más espacio consumen:

```
SELECT
relname as table,
pg_size_pretty(pg_total_relation_size(relid)) As size,
pg_size_pretty(pg_total_relation_size(relid) - pg_relation_size(relid))
    as external_size
FROM pg_catalog.pg_statio_user_tables
ORDER BY pg_total_relation_size(relid) DESC LIMIT 10;
```

Consultar el tamaño de todos los objetos:

```
SELECT
relname AS objectname, relkind AS objecttype,
reltuples AS "#entries", pg_size_pretty(relpages::bigint*8*1024) AS
   size
FROM pg_class
ORDER BY relpages DESC;
```

Esta última consulta muestra 4 campos: objectname: el nombre del objeto.

objecttype: r = tabla, i = índice, S = secuencia, v = vista, c = tipo compuesto, t = tabla TOAST.

#entries: el número de entradas en el objeto (p.ej. filas)
size: el tamaño del objeto.

Backups y Restauración

Este capítulo explica cómo exportar y restaurar (o importar) bases de datos PostgreSQL desde línea de comandos utilizando las herramientas pg_dump, pg_dumpall y psql.

PostgreSQL implementa la mayor parte del estándar SQL:2011, cumple con las propiedades ACID (Atomicity, Consistency, Isolation, Durability), es completamente transaccional (incluyendo todas las sentencias DDL, Data Definition Language), posee vistas extensibles actualizables, tipos de datos, operadores, índices, funciones, agregación, incluye un lenguaje procedural, y tiene un gran número de extensiones de terceros.

PostgreSQL funciona en la mayoría de los sistemas operativos incluyendo GNU/Linux, FreeBSD, Solaris, Microsoft Windows y MacOS X. La gran mayoría de distribuciones GNU/Linux disponen de PostgreSQL en sus repositorios.

Trabajamos con algo sencillo: cómo exportar y restaurar bases de datos Postgres desde línea de comandos.

0.47. Respaldo/Backup

Si requerimos respaldar una base de datos, ya sea para generar una copia (backup) o para importar en otro servidor, es posible utilizar la herramienta pg_dump, la cual vuelca una base de datos en una secuencia de instrucciones SQL en formato de texto plano. Si requiere exportar la base de datos "mibd.ªl archivo "pg_mibd.sqlütilizando el usuario "postgres", ejecute:

```
1 # pg_dump -U postgres -f pg\_mibd.sql mibd
```

O también:

```
1 # pg_dump -U postgres mibd > pg\_mibd.sql
```

Para obtener ayuda sobre pg_dump, ejecute:

```
1 # pg\_dump --help
2 pg_dump dumps a database as a text file or to other formats.
```

Uso:

```
1 pg_dump [OPTION]... [DBNAME]
```

Opciones Generales:

Cuadro 3: Lista de opciones generales de pg_dump

Opción	Descripción
-f, –file=FILENAME	output file or directory name
-F, –format=c|d|t|p	output file format (custom, directory, tar, plain text (default))
-j, –jobs=NUM	use this many parallel jobs to dump
-v, –verbose	verbose mode
-V, –version	output version information, then exit
-Z, –compress=0-9	compression level for compressed formats
–lock-wait-timeout=TIMEOUT	fail after waiting TIMEOUT for a table lock
- \? , –help	show this help, then exit

Opciones de control de salida:

Cuadro 4: Lista de opciones de salida de pg_dump

Opción	Descripción
-a, –data-only	dump only the data, not the schema
-b, –blobs	include large objects in dump
-c, –clean	clean (drop) database objects before recreating
-C, –create	include commands to create database in dump
-E, –encoding=ENCODING	dump the data in encoding ENCODING
-n, –schema=SCHEMA	dump the named schema(s) only
-N, –exclude-schema=SCHEMA	do NOT dump the named schema(s)
-o, –oids	include OIDs in dump
-O, –no-owner	skip restoration of object ownership in plain-text format
-s, –schema-only	dump only the schema, no data
-S, –superuser=NAME	superuser user name to use in plain-text format
-t, –table=TABLE	dump the named table(s) only
-T, –exclude-table=TABLE	do NOT dump the named table(s)
-x, –no-privileges	do not dump privileges (grant/revoke)
–binary-upgrade	for use by upgrade utilities only
–column-inserts	dump data as INSERT commands with column names
–disable-dollar-quoting	disable dollar quoting, use SQL standard quoting
–disable-triggers	disable triggers during data-only restore
–exclude-table-data=TABLE	do NOT dump data for the named table(s)
–inserts	dump data as INSERT commands, rather than COPY
–no-security-labels	do not dump security label assignments
–no-synchronized-snapshots	do not use synchronized snapshots in parallel jobs
–no-tablespaces	do not dump tablespace assignments
–no-unlogged-table-data	do not dump unlogged table data
–quote-all-identifiers	quote all identifiers, even if not key words
–section=SECTION	dump named section (pre-data, data, or post-data)
–serializable-deferrable	wait until the dump can run without anomalies
–use-set-session-authorization	use SET SESSION AUTHORIZATION commands instead of ALTER OWNER commands to set ownership

Opciones de Conexión:

Cuadro 5: Lista de opciones de conexión de pg_dump

Opción	Descripción
-d, – dbname=DBNAME	database to dump
-h, –host=HOSTNAME	database server host or socket directory
-p, –port=PORT	database server port number
-U, –username=NAME	connect as specified database user
-w, –no-password	never prompt for password
-W, –password	force password prompt (should happen automatically)
–role=ROLENAME	do SET ROLE before dump

Si no se suministra el nombre de la base de datos, entonces se utiliza la variable PGDATABASE.

Si requiere exportar todas las bases de datos de un servidor PostgreSQL, utilice pg_dumpall:

```
# pg\_dumpall -U postgres > pg\_todo.sql
```

Para obtener ayuda sobre pg_dumpall, ejecute:

```
# pg\_dumpall --help
pg_dumpall extracts a PostgreSQL database cluster into an SQL script
    file.
```

Uso:

```
pg_dumpall [OPTION]...
```

Veamos un caso completo:

```
pg_dump --host localhost --port 5432 --username "jorge" --role "jorge"
    --no-password  --format plain --create --clean --section pre-data --
    section data --section post-data --encoding UTF8 --verbose --file "/
    home/jorge/respaldo.sql" "dvdrental"
```

0.47.1. Explicación

Al realizar el backup de una base de datos y su restauración en una nueva base de datos, al verificar el peso de cada una son diferentes, al revisar la cantidad de registros y todo lo demás es igual, ¿Por qué ocurre esto?

Cuando trabaja con una base de datos y realiza actualizaciones (UPDATE) o borrados (DELETE) en realidad lo que hace PostgreSQL es mantener las versiones viejas/old de las filas de las tablas actualizadas o borradas. Es decir, al modificar una fila queda ”oculta” creando una nueva/new fila con los datos actualizados.

Del mismo modo, al borrar una fila permanece en la base de datos, y no está disponible para nuevas transacciones.

Esto genera que las tablas ”aumenten” con filas que no son necesarias (filas con datos desactualizados y filas borradas). Utilice un proceso de limpieza llamado VACUUM que libera el espacio usado por estas filas ”no necesarias” para un nuevo uso.

Al realizar un backup de la base de datos no tiene sentido ”llevarse” a la copia de seguridad esas filas ”no necesarias” (que ocupan espacio y no tendría sentido) por eso al recuperar el backup de la nueva base de datos ocupa menos.

0.48. pg_probackup

pg_probackup es una utilidad para gestionar el backup y la recuperación de clusters de bases de datos PostgreSQL. Está diseñado para realizar copias de seguridad periódicas de la instancia de PostgreSQL que le permiten restaurar el servidor en caso de fallo. pg_probackup soporta PostgreSQL 9.5 o superior.

0.48.1. Panorama general

En comparación con otras soluciones de copia de seguridad, pg_probackup ofrece los siguientes beneficios que pueden ayudarle a implementar diferentes estrategias de copia de seguridad y a manejar grandes cantidades de datos:

1. Copia de seguridad incremental: la copia de seguridad incremental a nivel de página le permite ahorrar espacio en disco, acelerar la copia de seguridad y la restauración. Con tres modos incrementales diferentes, puede planificar la estrategia de copia de seguridad de acuerdo con su flujo de datos

2. Validación: comprobaciones automáticas de coherencia de datos y validación de copias de seguridad a petición sin recuperación de datos reales.

3. Verificación: verificación bajo demanda de la instancia de PostgreSQL a través del comando dedicado checkdb

4. Retención: gestión de archivos y copias de seguridad de WAL de acuerdo con las políticas de retención - Basada en el tiempo y/o la redundancia, con dos métodos de retención: eliminar archivos caducados y fusionar archivos caducados. Además, puede diseñar su propia política de retención estableciendo'time to live'para las copias de seguridad.

5. Paralelización: ejecución de procesos de copia de seguridad, restauración, fusión, eliminación, verificación y validación en múltiples subprocesos paralelos.

6. Compresión: almacenar los datos de la copia de seguridad en un estado comprimido para ahorrar espacio en el disco.

7. Deduplicación: ahorro de espacio en disco al no copiar los archivos no modificados ('_vm', '_fsm', etc)

8. Operaciones remotas: copia de seguridad de la instancia PostgreSQL ubicada en la máquina remota o restauración de la copia de seguridad en la misma. Copia de seguridad desde la réplica: evite la carga adicional en el servidor maestro mediante la realización de copias de seguridad desde el modo de espera.

9. Directorios externos: añade al contenido de la copia de seguridad de los directorios ubicados fuera del directorio de datos PostgreSQL (PGDATA), tales como scripts, configs, logs y archivos pg_dump.

10. Catálogo de copias de seguridad: obtenga una lista de copias de seguridad y la información meta correspondiente en formato plano o json.

11. Catálogo de archivos: obtenga una lista de todas las líneas de tiempo de WAL y la meta información correspondiente en formato plano o json.

12. Restauración parcial: restaura sólo las bases de datos especificadas u omite las bases de datos especificadas.

Para gestionar los datos de la copia de seguridad, pg_probackup crea un catálogo de copias de seguridad. Este es un directorio que almacena todos los archivos de copia de seguridad con meta adicionales. Usando pg_probackup, puede realizar copias de seguridad completas o incrementales:

1. Las copias de seguridad FULL contienen todos los archivos de datos necesarios para restaurar el clúster de la base de datos.

2. Las copias de seguridad incrementales sólo almacenan los datos que han cambiado desde la copia de seguridad anterior. Disminuye el tamaño de la copia de seguridad y acelera las operaciones de copia de seguridad y restauración. pg_probackup soporta los siguientes modos de copias de seguridad incrementales:

    a) Respaldo DELTA. En este modo, pg_probackup lee todos los archivos de datos en el directorio de datos y copia sólo aquellas páginas que han cambiado desde la copia de seguridad anterior. Tenga en cuenta que este modo puede imponer una presión de E/S de sólo lectura igual a la de una copia de seguridad completa.

    b) Copia de seguridad de PÁGINA. En este modo, pg_probackup analiza todos los archivos WAL del archivo comprimido desde el momento en que se realizó la copia de seguridad completa o incremental anterior. Las copias de seguridad recién creadas contienen sólo las páginas que se mencionaron en los registros WAL. Esto requiere que todos los archivos WAL desde la copia de seguridad anterior estén presentes en el archivo WAL. Si el tamaño de estos archivos es comparable al tamaño total de los archivos del clúster de la base de datos, la velocidad es menor, pero la copia de seguridad ocupa menos espacio. Debe configurar el archivo WAL como se explica en la sección Configuración del archivo WAL continuo para realizar copias de seguridad de PAGE.

    c) Respaldo PTRACK. En este modo, PostgreSQL rastrea los cambios de página sobre la marcha. El archivo continuo no es necesario para que funcione. Cada vez que se actualiza una página de una relación, esta página se marca en un mapa de bits PTRACK especial para esta relación. Como una página sólo requiere un bit en la horquilla PTRACK, estos mapas de bits son bastante pequeños. El seguimiento implica una pequeña sobrecarga en el funcionamiento del servidor de bases de datos, pero acelera considerablemente las copias de seguridad incrementales.

pg_probackup sólo realiza copias de seguridad físicas en línea, y las copias de seguridad en línea requieren WAL para una recuperación consistente. Por lo tanto, independientemente del modo de copia de seguridad elegido (FULL, PAGE o DELTA), cualquier copia de seguridad realizada con pg_probackup debe utilizar uno de los siguientes modos de entrega de WAL:

1. ARCHIVO. Estas copias de seguridad se basan en el archivo continuo para garantizar una recuperación coherente. Este es el modo de entrega predeterminado de WAL.

2. STREAM. Tales copias de seguridad incluyen todos los archivos necesarios para restaurar el clúster a un estado consistente en el momento en que se realizó la copia de seguridad.

Independientemente de si se ha configurado un archivo continuo o no, los segmentos WAL necesarios para una recuperación coherente se transmiten a través del protocolo de replicación durante la copia de seguridad y se incluyen en los archivos de copia de seguridad. Por ello, las copias de seguridad de este modo WAL se denominan autónomas o autónomas.

0.48.2. Limitaciones

pg_probackup actualmente tiene las siguientes limitaciones:

1. Sólo se admiten PostgreSQL de las versiones 9.5 y posteriores.
2. Actualmente, el modo de operaciones remode no es compatible con los sistemas Windows.
3. En sistemas Unix, la copia de seguridad de las veriones PostgreSQL =<10 sólo es posible por el mismo usuario del sistema operativo con el que se está ejecutando el servidor PostgreSQL. Por ejemplo, si el servidor PostgreSQL está siendo ejecutado por postgres del usuario, entonces la copia de seguridad debe ser ejecutada por postgres del usuario. Si la copia de seguridad se está ejecutando en modo remoto usando ssh, entonces esta limitación se aplica de forma diferente: el valor de la opción –remote-user debería ser postgres.
4. Durante la copia de seguridad de PostgreSQL 9.5 las funciones pg_create_restore_point(text) y pg_switch_xlog() se ejecutarán sólo si la función de copia de seguridad es superusuario. Debido a que la copia de seguridad de un clúster con baja cantidad de tráfico WAL con función de no superusuario puede llevar más tiempo que la copia de seguridad del mismo clúster con función de superusuario.
5. El servidor PostgreSQL del que se tomó la copia de seguridad y el servidor restaurado deben ser compatibles con los parámetros block_size y wal_block_size y tener el mismo número de versión principal. También dependiendo de la configuración del cluster, PostgreSQL puede aplicar restricciones adicionales como la plataforma de arquitectura de CPU y las versiones libc/libicu.
6. La cadena incremental sólo puede extenderse dentro de una línea de tiempo. Por lo tanto, si tiene una cadena incremental de copia de seguridad tomada de una réplica y se promociona, se verá obligado a realizar otra copia de seguridad COMPLETA.

Instalación y configuración Una vez que haya instalado pg_probackup, complete la siguiente configuración:

1. Inicialice el catálogo de copias de seguridad.
2. Añada una nueva instancia de copia de seguridad al catálogo de copias de seguridad.
3. Configure el clúster de la base de datos para habilitar las copias de seguridad de pg_probackup.
4. Opcionalmente, configure SSH para ejecutar operaciones de pg_probackup en modo remoto.

0.48.3. Inicialización del catálogo de copias de seguridad

pg_probackup almacena todos los archivos WAL y de copia de seguridad en los subdirectorios correspondientes del catálogo de copias de seguridad. Para inicializar el catálogo de copias de seguridad, ejecute el siguiente comando:

```
pg\_probackup init -B directorio_de_copias_de_seguridad
```

Donde backup_dir es la ruta al catálogo de copias de seguridad. Si el directorio_de_copias_de_seguridad ya existe, debe estar vacío. De lo contrario, pg_probackup devuelve un error.

Para inicializar el catalogo dele backup catalog, ejecute el siguiente comando:

```
pg_probackup init -B backup_dir
```

Donde backup_dir es la ruta al catalogo al backup. Sí existe, debe estar vacío. Por el contrario, pg_probackup devuelve un error.

El usuario que inicia pg_probackup debe tener acceso completo al directorio backup_dir.

pg_probackup crea el catálogo de copias de seguridad de backup_dir, con los siguientes subdirectorios:

- wal/ - directorio para los archivos WAL.
- backups/ - directorio para los archivos de copia de seguridad.

Una vez inicializado el catálogo de copias de seguridad, puede agregar una nueva instancia de copia de seguridad.

0.48.4. Adición de una nueva instancia de copia de seguridad

pg_probackup puede almacenar copias de seguridad para múltiples clusters de bases de datos en un único catálogo de copias de seguridad. Para configurar los subdirectorios necesarios, debe agregar una instancia de copia de seguridad al catálogo de copias de seguridad para cada clúster de base de datos del que vaya a realizar la copia de seguridad.

Para añadir una nueva instancia de copia de seguridad, ejecute el siguiente comando:

```
pg\_probackup add-instance -B backup\_dir -D data\_dir --instance
    instance\_name[remote\_options]
```

Dónde:

- data_dir es el directorio de datos del clúster del que va a realizar la copia de seguridad. Para configurar y utilizar pg_probackup, se requiere acceso de escritura a este directorio.
- instance_name es el nombre de los subdirectorios que almacenarán WAL y los archivos de copia de seguridad para este cluster.
- Los parámetros opcionales remote_options deben utilizarse si data_dir se encuentra en una máquina remota.

pg_probackup crea los subdirectorios instance_name bajo los directorios 'backups/'y 'wal/'del catálogo de copias de seguridad. El directorio'backups/instance_name'contiene el archivo de configuración'pg_probackup.conf'que controla la configuración de pg_probackup para esta instancia de copia de seguridad. Si ejecuta este comando con las opciones remote_options, los parámetros utilizados

se añadirán a pg_probackup.conf.

Para obtener más información sobre cómo ajustar la configuración de pg_probackup, consulte la sección Configuración de pg_probackup.

```
pg_probackup help[comando]

pg_probackup init -B directorio_de_copias_de_seguridad

pg_probackup add-instance -B backup_dir -D data_dir --instance
    instance_name

pg_probackup del-instance -B directorio_de_copias_de_seguridad --
    nombre_de_instancia

pg_probackup set-config -B backup_dir --instance instance_name[option
    ....]

pg_probackup set-backup -B backup_dir --instancia instance_name -i
    backup_id[opción...]].

pg_probackup show-config -B backup_dir --instance instance_name[--
    format=format]

pg_probackup show -B directorio_de_copias_de_seguridad[opción...]

pg_probackup backup -B backup_dir --instance instance_name -b
    backup_mode[opción...]].

pg_probackup restore -B
    directorio_de_la_copia_de_la_copia_de_la_instancia[opción...]].

pg_probackup checkdb -B backup_dir --instance instance_name[-D data_dir
    ][opción...]].

pg_probackup validate -B backup_dir[opción...]

pg_probackup merge -B backup_dir --instance instance_name -i backup_id[
    option...]].

pg_probackup delete -B backup_dir --instance instance_name { -i
    backup_id | --delete-wal | --delete-expired | --merge-expired } [
    opción...]

pg_probackup archive-push -B backup_dir --instance instance_name --wal-
    file-path=wal_file_path --wal-file-name=wal_file_name[opción...]].

pg_probackup archive-get -B backup_dir --instance instance_name --wal-
    file-path=wal_file_path --wal-file-name=wal_file_name[opción...]].
```

0.49. Restore

Continuando con las operaciones habituales para el mantenimiento de la base de datos.

El comando a utilizar (y conocer) es pg_restore.

Si lo aplica a los casos vistos con pg_dump, el pg_restore seria el siguiente:

```
1 pg_restore --host localhost --port 5432 --username "jorge" --dbname "
    dvdrental" --role "jorge" --no-password  --section pre-data --
    section data --section post-data --verbose "/home/jorge/respaldo.sql
    "
2 pg_restore: [archiver] el archivo de entrada parece ser un volcado de
    texto. Por favor use psql.
```

Una vez que lo ejecute, solicta la contraseña del usuario a usar.

Para conocer los parámetros que se pasan al comando, leer la documentación oficial, ya que hay ciertos detalles que debe conocer (es buena práctica leer la documentación oficial).

Sobre los parámetros del caso, vea la tabla 6:

Cuadro 6: Parámetros de pg_restore

Opción	Descripción
-i	le indica que ignore la versión (entre el comando y la base de datos).
-h	localhost es el host de nuestro PostgreSql.
-p	5432 es la indicación del puerto donde corre el servicio.
-U	postgres especifica que se usará el usuario postgres para la operación.
-d	mibase es para que realize la restauración sobre una base de datos en particular, en este caso mibase.
-v	ejecuta el comando en modo verbose (así podremos ir viendo la salida de cada paso del proceso).

/home/jorge/respaldo.sql es el archivo que usa como backup y que queremos ingresar.

Hasta ahora, ha realizado backups, automatizarlos, y ahora, restaura las bases de datos desde la consola sin problemas.

0.50. Optimización y mejoría al desempeño con VACUUM, ANALYZE, y REINDEX

Si ejecuta una base de datos PostgreSQL, hay tres comandos para mejorar y optimizar el rendimiento.

Trabajamos con: VACUUM, ANALIZE y REINDEX.

Para evitar actualizaciones conflictivas de la base de datos, o daños, es ejecute estos comandos durante una sesión de mantenimiento cuando la aplicación está detenida.

En la configuración predeterminada de PostgreSQL, el servicio AUTOVACUUM está activado y todos los parámetros de configuración necesarios se ajustan según sea necesario. El servicio ejecuta VACUUM y ANALYZE a intervalos regulares. Si esta habilitado, estos comandos complementan

el trabajo del servicio. Para confirmar si el servicio de autovacuum se está ejecutando en UNIX, verificamos la lista de procesos.

```
ps aux|grep autovacuum|grep -v grep
```

Figura 18: Verificando que autovacuum está instalado

En UNIX o Windows, encontramos el estatus de autovacuum en la base de datos con la consulta de pg_settings:

```
select name, setting from pg_settings where name = 'autovacuum' ;
```

0.50.1. Vacuum

El comando VACUUM recupera el espacio utilizado por los datos que han sido actualizados. En PostgreSQL, las tuplas de valores clave actualizadas no se eliminan de las tablas cuando se cambian las filas, por lo que el comando VACUUM debe ejecutarse ocasionalmente para hacerlo.

El VACUUM funciona con o sin ANALYZE.

Cuando la lista de opciones está rodeada de paréntesis, las opciones se pueden escribir en cualquier orden. Sin paréntesis, las opciones deben especificarse exactamente en el orden que se muestra a continuación. La sintaxis entre paréntesis fue añadida a partir de PostgreSQL 9.0; por lo quel la sintaxis entre paréntesis es obsoleta.

0.50.1.1. Ejemplos

En los ejemplos siguientes,[nombre de tabla] es opcional. Sin una tabla especificada, VACUUM se ejecuta en las tablas disponibles en el esquema actual al que el usuario tenga acceso.

VACUUM Simple: Libera espacio para la reutilización

```
VACUUM[nombre de tabla]
VACUUM completo: bloquea la tabla de la base de datos y recupera más
    espacio que un VACUUM normal.
```

```
/* Antes de Postgres 9.0: *//
VACUUM FULL
/* Postgres 9.0+: *///
VACUUM(FULL)[nombre de tabla]
```

VACUUM FULL y ANALYZE: para realizar un VACUUM completo y recopilar nuevas estadísticas sobre las rutas de ejecución de las consultas utilizando ANALYZE.

```
/* Antes de Postgres 9.0: */
VACUUM FULL ANALYZE [tablename]
/* Postgres 9.0+: */
VACUUM(FULL, ANALYZE) [tablename]
```

Verbose Full VACUUM y ANALYZE: Igual que #3, pero con salida de progreso verbose.

```
/* Before Postgres 9.0: */
VACUUM FULL VERBOSE ANALYZE [tablename]
/* Postgres 9.0+: */
VACUUM(FULL, ANALYZE, VERBOSE) [tablename]
```

0.50.2. Analyze

ANALYZE recopila estadísticas para el planificador de consultas para una ruta de ejecución de consultas más eficiente. Según la documentación de PostgreSQL, unas estadísticas precisas ayudan al planificador a elegir el plan de consulta más adecuado y, por lo tanto, a mejorar la velocidad de procesamiento de la consulta.

En el siguiente ejemplo,[nombre de tabla] es opcional. Sin una tabla especificada, ANALYZE se ejecuta en las tablas disponibles en el esquema actual al que el usuario tenga acceso.

```
ANALYZE VERBOSE[nombre de tabla]
```

0.50.3. Reindex

El comando REINDEX reconstruye uno o más índices, reemplazando la versión anterior del índice. REINDEX se puede utilizar en muchos escenarios, incluyendo los siguientes (de la documentación de Postgres):

- Un índice se ha corrompido y ya no contiene datos válidos. Aunque en teoría esto nunca debería ocurrir, en la práctica los índices pueden corromperse debido a errores de software o fallos de hardware. REINDEX proporciona un método de recuperación.
- Un índice se ha vuelto "hinchado", es decir, contiene muchas páginas vacías o casi vacías. Esto puede ocurrir con los índices B-tree en PostgreSQL bajo ciertos patrones de acceso poco comunes. REINDEX proporciona una forma de reducir el consumo de espacio del índice escribiendo una nueva versión del índice sin las páginas muertas.
- Ha alterado un parámetro de almacenamiento (como por ejemplo, fillfactor) para un índice y desea asegurarse de que la modificación ha surtido pleno efecto.
- Una construcción de índice con la opción CONCURRENTLY falló, dejando un índice ïnválido". Estos índices son inútiles, pero puede ser conveniente utilizar REINDEX para reconstruirlos. Tenga en cuenta que REINDEX no realizará una compilación simultánea. Para construir el índice sin interferir con la producción, debe dejar caer el índice y volver a emitir el comando CREATE INDEX CONCURRENTLY.

Ejemplos Cualquiera de estos puede ser forzado añadiendo la palabra clave FORCE después del comando
Recrea un solo índice, mi índice:

```
REINDEX INDEX myindex
```

Recrea todos los índices en una tabla, mi tabla:

```
REINDEX TABLE mytable

Recrear todos los índices en el esquema público:

\begin{lstlisting}
REINDEX SCHEMA public

Recrear todos los índices en la base de datos postgres:

\begin{lstlisting}
REINDEX DATABASE postgres

Recrear todos los índices de los catálogos del sistema en postgres de la base de datos:

\begin{lstlisting}
REINDEX SYSTEM postgres

\subsection{Vacuum}
El vacuum es el proceso en el cual se eliminan definitivamente tuplas marcadas para borrar y hay una reorganización de datos a nivel fí sico.
Puede realizar vacuum utilizando el comando externo 'vacuumdb' y el cual puede recibir parámetros para realizar los diferentes tipos de vaciamiento:
FREEZE
FULL
ANALYZE
S/Parametros de tipo
Este comando es muy útil para automatizar vaciamientos a través de cualquier sincronizador de tareas (ya sea el cron de *nix u otro de Windows).
Asimismo, existe la opción del Autovacuum, cuya funcionalidad es ir realizando de manera paulatina la mantención de nuestra base. Previamente y antes de activar esta funcionalidad, es recomendable leer sobre las consideraciones a tener en cuenta, para no degradar la performance de nuestro servidor.
\section{Recuperación/Restore}
Para importar o restaurar un volcado de bases de datos Postgres desde l ínea de comandos, se debe utilizar la herramienta psql, la cual funciona como terminal interactiva contra servidores PostgreSQL. Por ejemplo, si se desea restaurar el volcado del archivo "pg\_mibd.sql " (utilizando el usuario "postgres") guardando un log en el archivo "pg\_mibd.log", ejecutar:
\begin{lstlisting}
# psql -U postgres < pg_mibd.sql > pg_mibd.log 2>&1
```

Luego es posible buscar errores en el log mediante:

```
# grep -i error pg\_mibd.log
```

Para obtener ayuda sobre psql, ejecutar:

```
# psql --help
psql is the PostgreSQL interactive terminal.

Uso:
psql [OPTION]... [DBNAME [USERNAME]]

General options:
-c, --command=COMMAND    run only single command (SQL or internal) and
   exit
-d, --dbname=DBNAME       database name to connect to (default: "root")
-f, --file=FILENAME       execute commands from file, then exit
-l, --list                list available databases, then exit
-v, --set=, --variable=NAME=VALUE
set psql variable NAME to VALUE
-V, --version             output version information, then exit
-X, --no-psqlrc           do not read startup file (~/.psqlrc)
-1 ("one"), --single-transaction
execute as a single transaction (if non-interactive)
-?, --help                show this help, then exit

Input and output options:
-a, --echo-all            echo all input from script
-e, --echo-queries        echo commands sent to server
-E, --echo-hidden         display queries that internal commands
   generate
-L, --log-file=FILENAME   send session log to file
-n, --no-readline         disable enhanced command line editing (
   readline)
-o, --output=FILENAME     send query results to file (or |pipe)
-q, --quiet               run quietly (no messages, only query output)
-s, --single-step         single-step mode (confirm each query)
-S, --single-line         single-line mode (end of line terminates SQL
   command)

Output format options:
-A, --no-align            unaligned table output mode
-F, --field-separator=STRING
set field separator (default: "|")
-H, --html                HTML table output mode
-P, --pset=VAR[=ARG]      set printing option VAR to ARG (see \pset
   command)
-R, --record-separator=STRING
set record separator (default: newline)
-t, --tuples-only         print rows only
-T, --table-attr=TEXT     set HTML table tag attributes (e.g., width,
   border)
-x, --expanded            turn on expanded table output
-z, --field-separator-zero
set field separator to zero byte
-0, --record-separator-zero
set record separator to zero byte
```

```
46
47 Connection options:
48 -h, --host=HOSTNAME      database server host or socket directory (
       default: "local socket")
49 -p, --port=PORT          database server port (default: "5432")
50 -U, --username=USERNAME  database user name (default: "root")
51 -w, --no-password        never prompt for password
52 -W, --password          force password prompt (should happen
       automatically)
53
54 For more information, type "\?" (for internal commands) or "\help" (for
       SQL
55 commands) from within psql, or consult the psql section in the
       PostgreSQL
56 documentation.
57
58 Report bugs to <pgsql-bugs@postgresql.org>.
```

0.51. El trabajo

El modelo MVCC de PostgreSQL proporciona un excelente soporte para ejecutar múltiples transacciones que operan en el mismo conjunto de datos. Una consecuencia natural de su diseño es la existencia de la llamada "hinchazón de la base de datos". A juzgar por la cantidad de preguntas que se plantean en Internet, se trata de un problema bastante común y no parece que mucha gente sepa cómo abordarlo adecuadamente. Yo mismo tuve los mismos problemas y aprendí una o dos cosas que podrían ser útiles, así que en este artículo me gustaría arrojar algo de luz sobre esta noción: por qué está ahí en primer lugar, cómo puede afectar al rendimiento y, finalmente, cómo controlarla y gestionarla.

0.51.1. Por qué está ahí

En PostgreSQL MVCC - que significa MultiVersion Concurrency Control - proporciona a cada transacción de base de datos una instantánea consistente de los datos. (Es importante recordar que si no se ha abierto ninguna transacción explícita, cada consulta es una transacción por sí misma.) Esto significa que los cambios realizados en una transacción serán visibles para otras transacciones sólo después de que su transacción de .ºrigen"se haya comprometido satisfactoriamente. En realidad, también depende del nivel de aislamiento de otras transacciones, pero para mantener este artículo simple no nos preocupemos por eso.

Lo que es realmente asombroso acerca de MVCC es que se las arregla para manejar una tarea bastante compleja como el control de concurrencia usando un diseño muy simple y limpio. Para entender por qué las bases de datos se hinchan, es importante familiarizarse un poco más con el MVCC, así que echemos un vistazo.

```
1 compose (session 1) => create table test as select id from
      generate_series(1,1000) as id;
2 SELECT 1000
3 compose (session 1) => begin;
4 BEGIN
5 compose (session 1) => select txid_current();
```

```
 txid_current
--------------
         1844
(1 row)

compose (session 1) => select xmin, xmax, id from test limit 5;
 xmin | xmax | id
------+------+----
 1843 |    0 |  1
 1843 |    0 |  2
 1843 |    0 |  3
 1843 |    0 |  4
 1843 |    0 |  5
(5 rows)
```

Como puede ver, en la sesión de base de datos 1 he creado una tabla llamada test con un solo id de columna entera y la he llenado con 1000 filas. A continuación, he abierto una transacción explícita en la que se puede ver su identificador y las primeras 5 filas de la tabla de pruebas.

Aparte de la columna de identificación de la tabla, he indicado a la consulta que imprima también dos columnas del sistema: xmin y xmax. El primero almacena el identificador de la operación que ha insertado la línea y el segundo - el identificador de la operación que la ha borrado. En caso de que la fila no haya sido borrada, xmax es igual a 0.

A la transacción explícita que he abierto se le ha asignado el identificador 1844. Es mayor o igual que 1843, por lo que esta transacción en particular puede ver las filas.

Ahora, por favor, observe lo que sucede en una segunda sesión de base de datos.

```
compose (session 2) => begin;
BEGIN
compose (session 2) => select txid_current();
 txid_current
--------------
         1846
(1 row)

compose (session 2) => select xmin, xmax, id from test limit 2;
 xmin | xmax | id
------+------+----
 1843 |    0 |  1
 1843 |    0 |  2
(2 rows
```

Nada sorprendente aquí, espero que esté claro.

Borremos una tupla en la primera sesión (recuerde que todavía tenemos una transacción abierta allí).

```
compose (session 1) => delete from test where id = 1;
DELETE 1
compose (session 1) => select xmin, xmax, id from test where id = 1;
 xmin | xmax | id
------+------+----
(0 rows)

```

```
Y justo después de eso en la sesión 2:
\begin{lstlisting}
compose (session 2) => select xmin, xmax, id from test where id = 1;
 xmin | xmax | id
------+------+----
 1843 | 1844 |  1
(1 row)
```

Aunque el identificador de esta sesión es mayor o igual que 1844, todavía puede ver la fila eliminada, ya que la transacción de eliminación aún no ha comprometido su trabajo.

Esto es realmente importante - aunque la fila ha sido borrada, no ha sido eliminada físicamente de la página de datos. No se puede eliminar, ya que algunas transacciones pueden seguir viéndolo en sus instantáneas como una fila activa estándar. Lo mismo ocurre con las actualizaciones, ya que una instrucción UPDATE no es otra cosa que un DELETE y un INSERT combinados.

Ahora es el momento de limpiar:

```
compose (session 1) => commit;
COMMIT
compose (session 2) => select xmin, xmax, id from test where id = 1;
 xmin | xmax | id
------+------+----
(0 rows)

Después de que se confirma la ejecución de la transacción en la sesión
   1, la transacción que se ejecuta en la sesión 2 deja de ver
   inmediatamente la línea borrada.
\begin{lstlisting}
compose (session 2) => vacuum verbose test;
INFO:  vacuuming "public.test"
INFO:  "test": removed 1 row versions in 2 pages
INFO:  "test": found 1 removable, 999 nonremovable row versions in 5
   out of 5 pages
DETAIL:  0 dead row versions cannot be removed yet.
```

Como puede ver, después de todo, VACUUM se encarga de la tupla eliminada; comprueba que ya no hay ninguna transacción en ejecución que pueda ver esta fila y la elimina (bueno, es un poco más complicado, pero sobre eso más tarde). Filas como esta se llaman "filas muertasz si no fuera por el VACUUM, permanecerían. Este sencillo ejercicio demuestra el proceso que conduce a la formación de lo que se conoce como "hinchazón de la base de datos". Bajo ciertas circunstancias, con el servicio de autovacuum no lo suficientemente agresivo, ya que la hinchazón de las mesas escritas a mano puede ser un problema que tiene que ser resuelto por el DBA.

La mecánica de MVCC hace obvio por qué existe VACUUM y la tasa de cambios en las bases de datos hoy en día es un buen argumento para la existencia de un servicio de autovacuum. Aunque es común que los nuevos usuarios de PostgreSQL se muestren reacios a activar el autovacuum (o deseosos de desactivarlo, ya que está activado de forma predeterminada), la gente de compose.io hizo un buen trabajo.

```
compose (session 2) => select name, setting from pg_settings where name
    = 'autovacuum';
    name     | setting
```

```
------------+---------
 autovacuum | on
(1 row)
```

0.51.2. ¿Qué causa el problema?

Ahora imagina que tu mesa tiene muchas tuplas muertas. Cada escaneo secuencial que pasa por la mesa tiene que hacer su camino a filas vivas esquivando a los muertos. Obviamente, lo hace mucho más lento. Por otro lado, cada escáner de índice tiene que seguir punteros de índice a filas muertas sólo para ver que están.... bien, muertas. Así que de nuevo - podría haber sido más rápido si no fuera por la hinchazón. (Sí, los índices también pueden hincharse, pero para mantener las cosas simples, este artículo sólo cubre la hinchazón de datos.)

Otro problema es que el VACUUM no elimina todas las tuplas muertas. Tiene algunas limitaciones. Lo que hace VACUUM es simplemente comprobar, si la tupla está muerta y si es así, marcar el espacio que utiliza como adecuado para ser reutilizado. Así que en la mayoría de los casos no devuelve el espacio del disco al sistema de archivos subyacente. La razón es simple - los datos en los archivos de datos de PostgreSQL están organizados en páginas, cada página tiene 8kB por defecto. VACUUM sólo puede eludir los archivos de datos si las filas que se consideran muertas se encuentran al final de la última página de datos de todo el archivo de datos (vale la pena mencionar aquí que el archivo de datos está realmente almacenado en fragmentos de 1 GB).

He visto casos en los que una mesa grande tenía sus filas actualizadas pocas veces y luego sólo se leía desde ella; aunque VACUUM ha marcado las filas muertas como aptas para ser reutilizadas, el archivo de datos tenía el doble (¡o más!) del tamaño de la mesa (o, para ser exactos, la suma de los tamaños de las filas vivas en la mesa en cuestión).

Lo que también es bastante problemático cuando se lee de tablas muy hinchadas es la E/S; cuantos más datos se necesitan más operaciones de entrada/salida se necesitan.

Como puedes ver, muchas filas muertas pueden causar muchos problemas, por lo que es necesario al menos llevar un registro del porcentaje de golpes de tus tablas.

Echemos un vistazo a un escenario simple:

```
compose (session 1) => truncate test;
TRUNCATE TABLE
compose (session 1) => insert into test (id) select * from
   generate_series(1,10000000);
INSERT 0 10000000
compose (session 1) => \d+
                    List of relations
 Schema | Name | Type  | Owner |  Size  | Description
--------+------+-------+-------+--------+-------------
 public | test | table | admin | 346 MB |
(1 row)

compose (session 1) => explain (buffers, analyse) select * from test
   where id = 9999999;
                                                  QUERY PLAN
--
   ------------------------------------------------------------------------------
```

```
 Seq Scan on test  (cost=0.00..176992.00 rows=1 width=4) (actual time
    =855.088..855.089 rows=1 loops=1)
   Filter: (id = 9999999)
   Rows Removed by Filter: 9999999
   Buffers: shared hit=16202 read=28046
 Planning time: 0.043 ms
 Execution time: 855.120 ms
(6 rows)
```

Hemos truncado nuestra tabla de pruebas y he insertado 10.000.000 de nuevas filas. Entonces, seleccionamos uno de ellos. Tomó algún tiempo ya que no hay índice y PostgreSQL tuvo que pasar por todas las páginas de datos para encontrar la fila solicitada. Lo más interesante en este ejercicio son los contadores de búferes de aciertos y lecturas; muestran cuántas filas se han encontrado en los búferes compartidos (aciertos) y cuántas se han leído desde el disco (lecturas).

Ahora, vamos a crear algo de hinchazón!

```
compose (session 1) => delete from test where id > 1000 and id <
   9000000;
DELETE 8998999

compose (session 1) => vacuum analyse verbose test;
INFO:  vacuuming "public.test"
INFO:  "test": removed 8998999 row versions in 39820 pages
INFO:  "test": found 8998999 removable, 1001001 nonremovable row
   versions in 44248 out of 44248 pages
DETAIL:  0 dead row versions cannot be removed yet.
There were 0 unused item pointers.
0 pages are entirely empty.
CPU 0.50s/1.05u sec elapsed 2.11 sec.
INFO:  analyzing "public.test"
INFO:  "test": scanned 30000 of 44248 pages, containing 671315 live
   rows and 0 dead rows; 30000 rows in sample, 993641 estimated total
   rows
VACUUM

compose (session 1) => \d+
                    List of relations
 Schema | Name | Type  | Owner |  Size  | Description
--------+------+-------+-------+--------+-------------
 public | test | table | admin | 346 MB |
(1 row)
```

Como vemos, hemos borrado la mayoría de las filas y luego ejecutado un VACUUM que informó que no hay páginas completamente vacías, por lo que no se han eliminado páginas, por lo que el tamaño de la tabla sigue siendo el mismo. El espacio de disco ocupado por las filas eliminadas se ha marcado como disponible para su reutilización, pero sigue formando parte del archivo de datos de esta tabla.

Veamos qué pasa si ejecuto el mismo SELECT ahora.

```
compose (session 1) => explain (buffers, analyse) select * from test
   where id = 9999999;
                                                  QUERY PLAN
--
   ------------------------------------------------------------------------------
```

```
 Seq Scan on test  (cost=0.00..56668.51 rows=1 width=4) (actual time
    =178.370..178.372 rows=1 loops=1)
   Filter: (id = 9999999)
   Rows Removed by Filter: 1001000
   Buffers: shared hit=16219 read=28029
 Planning time: 0.053 ms
 Execution time: 178.409 ms
(6 rows)
```

Gracias a la eliminación de la hinchazón que creamos, PostgreSQL tiene ahora muchos menos datos por los que pasar y es capaz de satisfacer nuestra consulta mucho más rápido.

Es más rápido que el anterior, pero notamos que Es más rápido que el anterior, pero tenga en cuenta que los contadores del búfer de hit and read son más o menos los mismos que durante la ejecución anterior. Lo que esto significa es que esta vez PostgreSQL ha leído los búferes no directamente desde el disco, sino desde la caché del sistema operativo. Básicamente, la cantidad de trabajo que PostgreSQL tenía que hacer ahora era prácticamente la misma que antes.

0.51.3. ¿Qué podemos hacer al respecto?

```
compose (session 1) => vacuum full analyse verbose test;
INFO:  vacuuming "public.test"
INFO:  "test": found 0 removable, 1001001 nonremovable row versions in
   44248 pages
DETAIL:  0 dead row versions cannot be removed yet.
CPU 0.16s/0.36u sec elapsed 0.58 sec.
INFO:  analyzing "public.test"
INFO:  "test": scanned 4430 of 4430 pages, containing 1001001 live rows
    and 0 dead rows; 30000 rows in sample, 1001001 estimated total rows
VACUUM
compose (session 1) => \d+
                    List of relations
 Schema | Name | Type  | Owner | Size  | Description
--------+------+-------+-------+-------+-------------
 public | test | table | admin | 35 MB |
(1 row)
```

VACUUM FULL es una de las formas de eliminar la hinchazón. Esencialmente reescribe toda la tabla (sosteniendo un AccessExclusiveLock mientras lo hace). Mucho se ha dicho acerca de por qué no usar VACUUM COMPLETO si hay otras maneras de tratar el hinchazón. Pero es perfecto para nuestro simple ejercicio.

Gracias a una reescritura completa, el tamaño de la tabla bajó de 350MB a 35MB.

Vamos a seleccionar la misma fila ahora.

```
compose (session 1) => explain (buffers, analyse) select * from test
   where id = 9999999;
                                                QUERY PLAN
--
   ------------------------------------------------------------------------------
```

```
 Seq Scan on test  (cost=0.00..16942.51 rows=1 width=4) (actual time
    =86.967..86.969 rows=1 loops=1)
   Filter: (id = 9999999)
   Rows Removed by Filter: 1001000
   Buffers: shared hit=32 read=4398
 Planning time: 0.125 ms
 Execution time: 87.005 ms
(6 rows)
```

Gracias a la eliminación de la hinchazón que creamos, PostgreSQL tiene ahora muchos menos datos por los que pasar y es capaz de satisfacer nuestra consulta mucho más rápido.

0.51.4. Cómo monitorearlo

Monitorear el hinchazón de la base de datos es bastante simple; el útil script check_postgres puede hacerlo por usted. Aparte de eso, hay pocas preguntas. También es posible utilizar el módulo pgstattuple.

Mi favorito personal es una consulta de ioguix.

Déjeme mostrarle su salida en mi base de datos de prueba.

```
compose (session 1) => truncate test;
TRUNCATE TABLE
compose (session 1) => insert into test (id) select * from
   generate_series(1,10000000);
INSERT 0 10000000
compose (session 1) => delete from test where id > 1000 and id <
   9000000;
DELETE 8998999
compose (session 1) => \e
 current_database | schemaname | tblname | real_size | extra_size |
    extra_ratio    | fillfactor | bloat_size |   bloat_ratio    | is_na
--
   ----------------+------------+---------+-----------+------------+---------------

 compose          | public     | test    | 362479616 |  324009984 |
    89.3870909419635 |        100 |  324009984 | 89.3870909419635 | f
(1 row)
```

0.51.5. Cómo lidiar con ello

Como ya se ha mencionado, la forma tradicional de tratar el hinchazón es utilizando VACUUM FULL. Esto no es aconsejable en entornos de producción, principalmente porque el bloqueo pesado - esta consulta de utilidad tiene que adquirir y mantener durante toda su duración el bloqueo más fuerte existente - AccessExclusiveLock. Básicamente, saca toda la mesa de cualquier operación hasta que termina.

Pero afortunadamente hay mejores formas de eliminar el hinchazón hoy en día. Básicamente, se puede hacer de una de dos maneras.

En primer lugar, podemos reescribir la tabla como lo hace VACUUM FULL, pero de una manera más amable. El enfoque que toma pg_repack es simple, pero muy efectivo - vamos a crear una tabla idéntica, pero vacía, colocar algunos disparadores en la original y copiar su contenido a la nueva. Los triggers se encargan de sincronizar todos los cambios (incluidas las nuevas filas) en la

tabla intermedia mientras el proceso se está ejecutando y, una vez hecho, pg_repack recrea todos los índices e intercambia las dos tablas. De esta manera, la mesa original no está totalmente bloqueada y sólo hay una breve interrupción cuando hay que cambiar las mesas. Una desventaja es que pg_repack requiere espacio extra en disco para la tabla intermedia.

En segundo lugar, podemos forzar la reorganización de la mesa para reutilizar el espacio marcado como disponible por las ejecuciones de VACUUM. El script pgcompact lo hace utilizando la idea presentada por depesz en dos entradas de blog. Básicamente, sabiendo qué páginas de nuestro archivo de datos tienen huecos, podemos intentar ejecutar las llamadas .ªctualizaciones vacías"forzando a PostgreSQL a crear nuevas versiones de filas y a colocarlas en los huecos que queremos rellenar. Dirígete a los puestos de depesz para una descripción detallada. Otra cosa que vale la pena mencionar es que pgcompact es lo suficientemente inteligente como para evitar que cualquier desencadenante que pueda tener se dispare al establecer session_replication_role en replica. ¡Qué bien!

Echemos un vistazo a la eliminación de blog (usando pgcompact) en mi base de datos de prueba alojada en compose.io.

```
\$ ./pgcompact -v info -h aws-eu-west-1-portal.2.dblayer.com -p 10392 -
   U admin -W my_secret_password -d compose -t test
Sat Mar 12 17:05:10 2016 ERROR A database error occurred, exiting:
DatabaseChooserError Can not find an adapter amongst supported:
DatabaseError Can not execute command:
SET lc_messages TO 'C'; SET session_replication_role TO replica; SET
   statement_timeout TO '0'; SET synchronous_commit TO off;
ERROR:  permission denied to set parameter "lc_messages"
DatabaseError Can not execute command:
SET lc_messages TO 'C'; SET session_replication_role TO replica; SET
   statement_timeout TO '0'; SET synchronous_commit TO off;
ERROR:  permission denied to set parameter "lc_messages"

DatabaseError Can not execute command:
 SET lc_messages TO 'C'; SET session_replication_role TO replica; SET
    statement_timeout TO '0'; SET synchronous_commit TO off; SELECT 1;
 ERROR:  permission denied to set parameter "lc_messages"
ERROR:  permission denied to set parameter "session_replication_role"
```

Parece que necesitamos privilegios de superusuario para establecer algunos de los parámetros.

0.51.6. Probemos con pg_repack

```
aws-eu-west-1-portal.2.dblayer.com -p 10392 -U admin -d componer -t
   test
ERROR: pg_repack falló con error: Debes ser un superusuario para usar
   pg_repack
```

Ouch! La cosa es que pg_repack modifica los catálogos del sistema directamente para intercambiar las dos tablas. Esto sólo lo puede hacer el superusuario.

Lenguaje SQL

El lenguaje estructurado de consultas (SQL) es un lenguaje de base de datos normalizado, utilizado por la gran mayoría de los servidores de bases de datos que manejan bases de datos relacionales u objeto-relacionales. Es un lenguaje declarativo en el que las órdenes especifican cual debe ser el resultado y no la manera de conseguirlo (como ocurre en los lenguajes procedimentales). Al ser declarativo es muy sistemático, sencillo y con una curva de aprendizaje muy agradable ya que sus palabras clave permiten escribir las ordenes como si fueran frases en las que se especifica (en inglés) que es lo que queremos obtener.

0.52. Trabajando con la base de datos

La figura 19 muestra la forma de trabajar con una base de datos PostgreSQL. Primero, nos identificamos como usuario root, escribimos la contraseña del usuario root, luego, nos identificamos como usuario PostgreSQLe ingresamos al editor psql. Como regla de trabajo creamos el usuario dasd, la sintaxis del comando es:

```
1 CREATE USER dasd;
```

Descripción: define un nuevo rol de la base de datos

Sintaxis:

```
1 CREATE USER nombre [ [ WITH ] opción [ ... ] ]
```

Figura 19: Abiendo el editor psql.

donde opción puede ser una combinación de:

```
SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| BYPASSRLS | NOBYPASSRLS
| CONNECTION LIMIT límite_conexiones
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'contraseña'
| VALID UNTIL 'fecha_hora'
| IN ROLE nombre_de_rol [, ...]
| IN GROUP nombre_de_rol [, ...]
| ROLE nombre_de_rol [, ...]
| ADMIN nombre_de_rol [, ...]
| USER nombre_de_rol [, ...]
| SYSID uid
```

escribimos:

```
create user dasd nosuperuser nocreatedb encrypted password 'jdch';
```

Figura 20: Creando un usuario

0.53. Las tablas

En PostgreSQL los datos son almacenados en tablas, veamos:

```
lact_code | lact_lib
-------------+-----------------------------------------------
15        | Industrias alimenticias
16        | Industria del tabaco
17        | Industria textil
18        | Industria de la ropa y pieles
19        | Industria del cuero y calzado
```

en esta tabla tenemos 5 filas o tuplas y dos columnas, lact_code y lact_lib. Al hablar de columnas podemos utilizar como sinónimo campos, cada campo tiene un tipo particular, en nuestro ejemplo, el primero es de tipo numérico entero y el segundo es una cadena de caracteres.

psql: Limpiar la pantalla.

Trabajando con postgresql en modo consola, psql, hay un momento en el que tenemos

la pantalla llena de consultas y necesitamos liberar un poco para verlo mas claro, para ello, utilizamos:

CRTL + L

el cual cual funciona igual cuando trabaja en Linux como en putty.

0.53.1. Valor NULL

Dentro de una base de datos, en especifico, en cada tabla existe un valor particular que se llama NULL o indeterminado.

NULL no significa vacío, es diferente, si tenemos una tabla con el campo número_de_hijos y asignamos un 0, significa que no tenemos hijos, pero si no asignamos un valor al campo tenemos NULL o indeterminado, igualmente pasa con el campo apellido, si contiene el valor NULL es indeterminado; pero si asignamos vacío, la persona no tiene apellido. Por lo que la cadena vacío y el valor NULL son valores completamente diferentes, indeterminado y determinado.

El valor NULL no es un tipo de dato y no realizan operaciones aritméticas porque es indeterminado. 1 + NULL genera NULL. La concatenación de una cadena a un NULL también produce un NULL.

```
1 Prueba1=> SELECT * FROM alumnos;
2 id | nombre | apellidos | edad | clase
3 ----+--------+---------------+-------+-------7
4 5  | MARTIN | Philippe |
5 6  | TUX
6 (2 rows)
```

0.54. CREATE TABLE, SELECT, INSERT, DELETE

Comandos para trabajar con tablas.

0.54.1. Creación de tablas

El comando SQL CREATE TABLE crea una tabla. Para cear la tabla alumnos, compuesta de 3 campos: nombre, apellido y edad, ejecutamos:

```
1 Prueba1=# CREATE TABLE alumnos (
2 Prueba1 (# nom varchar(20),
3 Prueba1 (# apellidos varchar(20),
4 Prueba1 (# edad integer);
5 CREATE
```

El comando puede estar en una sola línea, pero esto no es importante ya que el comando no se ejecuta hasta que se encuentra un ; (punto y coma) que indica el fin de sentencia. Después, escribimos el nombre de la tabla (alumnos) y entre paréntesis la lista de los campos. Para cada campo, especificamos su nombre y tipo. En nuestro caso tenemos 2 tipos de campo, integer que es un entero con signo y que ocupa 32 bits y varchar que es una cadena de caracteres de longitud variable. Debemos recordar que Linux y PostgreSQL son sensibles a las mayúsculas y minúsculas, por lo que Create Table alumnos, produce el mismo resultado, pero así debemos escribir el nombre la tabla para todas sus referencias siguientes.

0.54.2. Adición y consulta de datos

Para insertar datos en nuestra tabla alumnos utilizamos el comando INSERT INTO.

```
Prueba1=# INSERT INTO alumnos VALUES('Martin','DUPONT',25);
INSERT 53638 1
```

El valor 53638 o cualquier otro número devuelto cuando el comando es ejecutado corresponde al OID que se atribuye al nuevo registro (oid es un identificador numérico único que corresponde a cada objeto dentro de PostgreSQL) debemos estar atentos a que el número que aparezca en nuestra aplicación puede ser diferente. El número siguiente indica que el comando INSERT afecto a un sólo registro.

0.55. Caso de estudio UPDATE

```
 INSERT INTO target SET c1 = x, c2 = y+z, ... FROM tables-providing-x-y
    -z
```

(with the patch as-submitted corresponding to the case with an empty

```
 FROM clause, por lo tanto, no hay variables en las expresiones a
    asignar).
```

Por supuesto, esto no es funcionalmente distinto de:

```
 INSERT INTO target(c1,c2,...) SELECT x, y+z, ... FROM tables-providing
    -x-y-z
```

y es justo preguntarse si vale la pena apoyar un proyecto no estándar sintaxis sólo para permitir que los nombres de las columnas de destino se escriban más cerca de las expresiones a ser asignadas.

Por si sirve de algo, creo que esto sería lo suficientemente útil como para justificar su existencia. En los días de antaño, cuando los dragones vagaban por la tierra y Escribí aplicaciones basadas en bases de datos en lugar de hackear el sistema a menudo me preguntaba por qué tenía que escribir dos declaraciones SQL completamente diferentes, una para insertar los datos que un usuario había introducido en un formulario web en la base de datos, y otro para actualizar los datos introducidos previamente. Esta característica permitiría a aquellos de la misma manera, lo que me habría gustado, en otros tiempos.

Después podemos consultar los datos con el comando SELECT:

```
Martin

(1 row)
```

```
kylodb=# SELECT nom FROM alumnos;
nom
--------
Martin
Charles

(2 filas)
```

```
kylodb=# SELECT nom, apellidos FROM alumnos;
nom     | apellidos
---------+-----------
Martin  | DUPONT
Charles | AZNAVOUR

(2 filas)
```

El argumento * permite seleccionar todos los registros y todas sus filas, representa un ahorro en la codificación de una aplicación y recupera las columnas en el mismo orden en que fueron declaradas, aunque debe ser manejado con cuidado.

Un SELECT igualmente puede utilizarse para ejecutar operaciones, por ejemplo, el operador || efectuará una concatenación de dos cadenas de caracteres:

```
kylodb=# SELECT nom||' '||apellidos AS np, nom AS name FROM alumnos;
np                |  name
------------------+---------
Martin DUPONT     | Martin
Charles AZNAVOUR | Charles
(2 filas)
```

Aquí cambiamos el nombre de referencia de una columna con AS, recomendamos esta buena práctica en operaciones complejas y cuyo resultado necesitemos mostrar u operar con él posteriormente. Este técnica se conoce como asignación de ALIAS a una columna.

0.56. Operaciones con cursores

Como hemos visto, SQL manipular cursores. ¿Un ejemplo? Veamos las sentencias INSERT y SELECT; donde SELECT regresa un cursor con datos editados de una o más tablas, mientras que INSERT adiciona un cursor de datos en una tabla. Analicemos el siguiente caso:

```
kylodb=# SELECT * FROM alumnos;
nom     | apellidos | edad
---------+-----------+------
Martin  | DUPONT    |   25
Charles | AZNAVOUR  |   52
(2 filas)
```

```
kylodb=# INSERT INTO alumnos SELECT * FROM alumnos;
INSERT 0 2
```

Hemos copiado el contenido de la tabla alumnos en la misma tabla alumnos:

```
kylodb=# SELECT * FROM alumnos;
nom     | apellidos | edad
---------+-----------+------
Martin  | DUPONT    |   25
Charles | AZNAVOUR  |   52
Martin  | DUPONT    |   25
Charles | AZNAVOUR  |   52
(4 filas)
```

En el primer caso, obtenemos un cursor con dos registros, los que existen en la tabla. En el segundo, insertamos un cursor de dos registros en la tabla, en el tercer caso, obtenemos un cursor con cuatro

registros.

0.57. Haciendo limpieza

Para suprimir datos de una tabla, ejecutamos la sentencia DELETE:

```
kylodb=# DELETE FROM alumnos;
DELETE 4
```

Con ella suprimimos permanentemente todos los registros de la tabla alumnos. No hay forma de recuperarlos, a menos que nuevamente sean adicionados a nuestra tabla. También, suprimimos definitivamente la tabla alumnos, para recuperarla debemos crearla nuevamente con CREATE TABLE:

```
kylodb=# DROP TABLE alumnos;
DROP TABLE
```

0.58. WHERE y UPDATE

En el apartado anterior vimos la forma de recuperar registros en las tablas, pero las formas empleadas devolvían todos los registros de la mencionada tabla.

Ahora estudiaremos las posibilidades de filtrar los registros con el fin de recuperar o actualizar aquellos que cumplan con ciertas condiciones preestablecidas.

0.58.1. La cláusula WHERE

La cláusula WHERE se usa para determinar qué registros de las tablas enumeradas en la cláusula FROM aparecerán en los resultados de la instrucción SELECT. WHERE restringe o selecciona un grupo de registros que cumplan con cierta condición y siempre regresa un valor booleano. Analicemos el siguiente código:

```
kylodb=# CREATE TABLE alumnos(nom varchar,
kylodb(# apellidos varchar, edad integer, clase char(4));
CREATE TABLE
```

```
kylodb=# INSERT INTO alumnos VALUES('DUPONT', 'Martin', 6, kylodb(# 'CP
    ');
INSERT 0 1
```

```
kylodb=#  INSERT INTO alumnos VALUES('DURAND', 'Theo',15, '3A');
INSERT 0 1
```

```
kylodb=# INSERT INTO alumnos VALUES('DUPOND', 'Léa',12, '6B');
INSERT 0 1
```

```
kylodb=# SELECT * FROM alumnos WHERE edad > 10;
nom    | apellidos | edad | clase
--------+-----------+------+-------
DURAND | Theo      |   15 | 3A
DUPOND | Léa       |   12 | 6B
(2 filas)
```

Si observamos con atención el proceso de creación de la tabla, notaremos un nuevo tipo de dato incorporado: char(4). Lo que significa que el campo tiene una longitud fija de 4 caracteres, al asignarle '6Bél SGDB adiciona 2 espacios, al derecha, para completar la cadena, pero si sobrepasamos la cadena, sólo toma los cuatro primeros caracteres. Después insertamos tres registros en nuestra tabla y por último ejecutamos una sentencia SELECT con una cláusula $WHEREedad10$.

Esto significa que nuestra sentencia SQL muestra solamente aquellos registros cuyo campo edad es estrictamente mayor que 10.

la sentencia:

```
1 SELECT nombre FROM municipios WHERE poblacion > 5000 ORDER BY poblacion
    ;
```

Devuelve el nombre de aquellos municipios con una población mayor de 5000 habitantes y los presenta ordenados por número de habitantes. Sin embargo, los lenguajes declarativos carecen de la potencia de los procedimentales.

SQL se ha convertido, debido a su eficiencia, en un estandar para las bases de datos relacionales, de hecho el gran éxito del modelo de base de datos relacional se debe en parte a la utilización de un lenguaje como SQL. A pesar de su tesórico caracter estándar, se han desarrollado, sobre una base común, diversas versiones ampliadas como las de Oracle o la de Microsoft SQL server. Incluye diversos tipos de capacidades:

1. Comandos para la definición y creación de una base de datos (create table).
2. Comandos para inserción, borrado o modificación de datos (insert, delete, update).
3. Comandos para la consulta de datos seleccionados de acuerdo a criterios complejos que involucran diversas tablas relacionadas por un campo común (select).
4. Capacidades aritméticas: En SQL es posible incluir operaciones aritméticas así como comparaciones, por ejemplo $AB + 3$.
5. Asignación y comandos de impresión: es posible imprimir una tabla construida por una consulta o almacenarla como una nueva tabla.
6. Funciones de agregación: Operaciones tales como promedio (average), suma (sum), máximo (max), etc. se pueden aplicar a las columnas de una tabla para obtener una cantidad única y, a su vez, incluirla en consultas más complejas.

En una base de datos relacional, los resultados de la consulta son datos individuales, tuplas[2] o tablas generados a partir de consultas en las que se establecen una serie de condiciones basadas en valores numéricos.

Por ejemplo una típica consulta sobre una tabla en una base de datos relacional, utilizando SQL podría ser:

```
1 bd=# SELECT id, nombre, pob1991 FROM municipios WHERE pob1991 > 20000;
```

el resultado es una tabla con tres campos (id, nombre, poblacion) procedentes de la tabla municipios, las filas corresponderán sólo a aquellos casos en los que la poblacion en 1991 (columna pob1991) sea mayor que 20000. En el caso de que sólo uno de los municipios cumpliera la condición obtendríamos una sola fila y en caso de que la consulta fuera:

[2] equivalente a una fila de una tabla

```
bd=# SELECT pob1991 FROM municipios WHERE pob1991 > 20000;
```

obtendríamos un sólo número, la población del municipio más poblado.

0.58.2. Componentes del SQL

El lenguaje SQL está compuesto por comandos, cláusulas, operadores y funciones de agregado. Estos elementos se combinan en las instrucciones para crear, actualizar y manipular las bases de datos.

0.58.3. Comandos

Existen dos tipos de comandos SQL:

- Los que crean y definen nuevas bases de datos, campos e índices.
  - CREATE Utilizado para crear nuevas tablas, campos e índices
  - DROP Empleado para eliminar tablas e índices
  - ALTER Utilizado para modificar las tablas agregando campos o cambiando la definición de los campos.
- Los que generan consultas para ordenar, filtrar y extraer datos de la base de datos.
  - SELECT Utilizado para consultar registros de la base de datos que satisfagan un criterio determinado
  - INSERT Utilizado para cargar lotes de datos en la base de datos en una única operación.
  - UPDATE Utilizado para modificar los valores de los campos y registros especificados
  - DELETE Utilizado para eliminar registros de una tabla de una base de datos

0.58.4. Cláusulas

Las cláusulas son condiciones utilizadas para concretar que datos son los que se desea seleccionar o manipular.

Cuadro 7: Restricciones SQL

FROM	Utilizada para especificar la tabla de la cual se van a seleccionar los registros
WHERE	Utilizada para especificar las condiciones que deben reunir los registros que se van a seleccionar
GROUP BY	Utilizada para clasificar los registros seleccionados en grupos específicos
HAVING	Utilizada para expresar la condición que debe satisfacer cada grupo
ORDER BY	Utilizada para ordenar los registros seleccionados de acuerdo con un orden específico

0.58.5. Operadores Lógicos

Cuadro 8: Operadores lógicos SQL

AND	Evalua dos condiciones y devuelve un valor de verdad sólo si ambas son ciertas.
OR	Evalúa dos condiciones y devuelve un valor de verdar si alguna de las dos es cierta.
NOT	Devuelve el valor contrario de la expresión.

0.58.6. Operadores de Comparación

Cuadro 9: Operadores de comparación SQL

<	Menor que
	Mayor que
<	Distinto de
<=	Menor o Igual que
=	Mayor o Igual que
=	Igual que
BETWEEN	Utilizado para especificar un intervalo de valores.
LIKE	Para la comparación de una cadena de texto con una expresión regular

0.58.7. Funciones de Agregación

Las funciones de agregación se usan dentro de una cláusula SELECT en grupos de registros para devolver un único valor que se aplica a un grupo de registros.

Cuadro 10: Funciones SQL

AVG	Utilizada para calcular el promedio de los valores de un campo determinado
COUNT	Utilizada para devolver el número de registros de la selección
SUM	Utilizada para devolver la suma de todos los valores de un campo determinado
MAX	Utilizada para devolver el valor más alto de un campo especificado
MIN	Utilizada para devolver el valor más bajo de un campo especificado

0.59. Bases de datos relacionales

Es el modelo más utilizado hoy en día. Una base de datos relacional es básicamente un conjunto de tablas, similares a las tablas de una hoja de cálculo, formadas por filas (registros) y columnas (campos). Los registros representan cada uno de los objetos descritos en la tabla y los campos los atributos (variables de cualquier tipo) de los objetos. En el modelo relacional de base de datos, las tablas comparten algún campo entre ellas. Estos campos compartidos van a servir para establecer relaciones entre las tablas que permitan consultas complejas (ver figura 21. En esta figura aparecen

tres tablas con información municipal, en la primera aparecen los nombres de los municipios, en la segunda el porcentaje en cada municipio de los diferentes usos del suelo y en la tercera la población en cada municipio lo largo del siglo XX. Como campo común aparece ident, se trata de un identificador numérico, único para cada municipio[3].

La idea básica de las bases de datos relacionales es la existencia de entidades (filas en una tabla) caracterizadas por atributos (columnas en la tabla). Cada tabla almacena entidades del mismo tipo y entre entidades de distinto tipo se establecen relaciones[4]. Las tablas comparten algún campo entre ellas, estos campos compartidos van a servir para establecer relaciones entre las tablas. Los atributos pueden ser de unos pocos tipos simples:

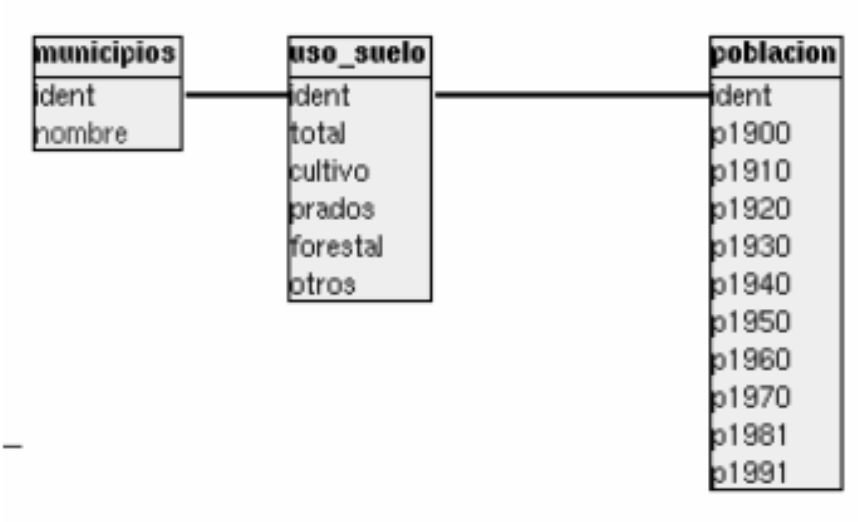


Figura 21: Esquema de base de datos relacional

1. Números enteros
2. Números reales
3. Cadena de caracteres de longitud variable

Estos tipos simples se denominan tipos atómicos y permiten una mayor eficacia en el manejo de la base de datos pero a costa de reducir la flexibilidad a la hora de manejar los elementos complejos del mundo real y dificultar la gestión de datos espaciales, en general suponen un problema para cualquier tipo de datos geométricos. Las relaciones que se establecen entre los diferentes elementos de dos tablas en una base de datos relacional pueden ser de tres tipos distintos:

1. Relaciones uno a uno, se establecen entre una entidad de una tabla y otra entidad de otra tabla. Un ejemplo aparece en la figura 21.

[3]Es preferible utilizar valores numéricos en lugar de una cadena de caracteres ya que se ahorra espacio y se evitan problemas con el uso de mayúsculas, acentos, etc.

[4]En la bibliografía inglesa sobre bases de datos se habla de relations (tablas) y relationships relaciones entre las tablas. El término base de datos relacional hace en realidad referencia a la organización de los datos en forma de tablas, no a las relaciones entre ellas.

2. Relaciones uno a varios, se establecen entre varias entidades de una tabla y una entidad de otra tabla. Un ejemplo es una tabla de pluviómetros en la que se indicara el municipio en el que se encuentra. La relación sería entre un municipio y varios pluviómetros

3. Relaciones varios a varios, se establecen entre varias entidades de cada una de las tablas. Un ejemplo sería una tabla con retenes de bomberos y otra con espacios naturales a los que cada uno debe acudir en caso de incendio.

0.60. Entrada en el cliente y exploración de la base de datos

La gestión de bases de datos se basa en la existencia de un programa servidor; que organiza los datos, recibe las consultas, las ejecuta y las devuelve; y un programa cliente que el usuario ejecuta y que lanza las consultas creadas por este al servidor. El programa cliente y el servidor no tienen siquiera porque ejecutarse en el mismo computador. Existen diferentes clientes para conectar al servidor de bases de datos de PostgreSQL. Vamos a utilizar en principio uno sencillo (psql). Si tecleamos:

```
psql -l
```

obtendremos un listado de todas la bases de datos disponibles para el servidor. Si queremos conectarnos a una de ellas se le especificará al teclear el comando:

```
psql clima
```

En este caso hemos especificado la base de datos a la que queremos conectarnos. El mensaje de bienvenida de psql será algo parecido a:

```
postgres@jodocha:/home/jorge$ psql kylodb
psql (9.6.15)
Digite help para obtener ayuda.
```

Disponemos de una serie de comandos formados por una barra y una letra que realizan operaciones sencillas:

1. \h para pedir ayuda sobre comandos SQL

2. \? para pedir ayuda sobre los comandos de barra y letra

3. \q para salir del programa

4. \d para obtener un listado de las tablas que forman la base de datos

Como ves tenemos 3 tipos de variables. El tipo int4 corresponde a números enteros, el tipo float8 corresponde a numeros reales y varchar a cadenas de caracteres. De todos estos atributos el más importante es ident ya que asigna a cada municipio un identificador único que coincide con el identificador del polígono correspondiente a dicho municipio en el mapa vectorial.

1. Tabla observatorios

```
Column    | Type                   | Modifiers --
   ---------+------------------------+----------
indentinm | character varying(6) |
nombre    | character varying(50)|
x         | integer              |
y         | integer              |
```

```
z          | smallint              |
ident      | integer               |
obs        | integer               |
```

2. Tabla menspluv

```
Column     | Type                  | Modifiers --
    ---------+-----------------------+----------
ide        | character varying(6)  |
mes        | smallint              |
ano        | smallint              |
pluv       | real                  |
ndias      | smallint              |
max        | real                  |
obs        | integer               |
```

3. Tabla menstem

```
Column     | Type                  | Modifiers --
    ---------+-----------------------+----------
ide        | character varying(6)  |
mes        | smallint              |
ano        | smallint              |
tmaxabs    | real                  |
tmaxmed    | real                  |
tmed       | real                  |
tminmed    | real                  |
tminabs    | real                  |
```

Tenemos 4 tipos de variables. El tipo integer (4 bytes) corresponde a números enteros, el tipo smallint (2 bytes) corresponde a números enteros lo suficientemente pequeños como para necesitar sólo 2 bytes, el tipo real (8 bytes) corresponde a numeros reales y character a cadenas de caracteres especificándose en cada caso el número de caracteres (bytes) que ocupa. De todos estos atributos el más importante es ide (en la tabla observatorios se llama indentinm ya que asigna a cada observatorio un identificador único que coincide en todas las tablas.

0.61. Consultas de Selección

Las consultas de selección se utilizan para indicar al servidor de base de datos que devuelva información de las bases de datos, tal como se ha visto esta información devuelta puede ser un valor, una tupla o una tabla. A partir de este momento todos los ejemplos se refieren a la base de datos clima.

0.61.1. Consultas básicas

La sintaxis básica de una consulta de selección es la siguiente:

```
clima=# SELECT campos FROM tabla;
```

En donde campos es la lista de campos que se deseen recuperar y tabla es el origen de los mismos, por ejemplo:

```
clima=# SELECT nombre,x,y,z FROM observatorios;
```

Esta consulta devuelve una tabla con el campo nombre y teléfono de la tabla clientes. La tabla devuelta no está almacenada en la base de datos, y por tanto no podrá ser objeto de posteriores consultas, salvo que la guardes de forma explícita con la orden SELECT INTO.

```
clima=# SELECT nombre,x,y,z INTO resumen FROM observatorios;
```

de esta manera se genera una nueva tabla que contiene sólo las cuatro columnas seleccionadas.

0.61.2. Ordenar los registros

Adicionalmente se puede especificar el orden en que se desean recuperar los registros de las tablas mediante la claúsula ORDER BY Lista de Campos. En donde Lista de campos representa los campos a ordenar. Ejemplo:

```
clima=# SELECT nombre,x,y,z
FROM observatorios
ORDER BY z;
```

Esta consulta devuelve los nombres de los observatorios junto a sus coordenadas pero ahora ordenados en función de su altitud.

Se pueden ordenar los registros por mas de un campo, como por ejemplo:

```
clima=# SELECT nombre,x,y,z
FROM observatorios
ORDER BY x,y;
```

Incluso se puede especificar el orden de los registros: ascendente mediante la claúsula (ASC -se toma este valor por defecto) o descendente (DESC)

```
clima=# SELECT nombre,x,y,z
FROM observatorios
ORDER BY x,y
DESC;
```

0.61.3. Consultas con Predicado

Una manera de limitar el número de filas que devuelve el servidor es utilizar predicados en la selección. El predicado se incluye entre la claúsula y el primer nombre del campo a recuperar, los posibles predicados son: * devuelve todos los campos de la tabla. En este caso el servidor de base de datos selecciona todos los registros que cumplen las condiciones de la instrucción SQL.

```
clima=# SELECT *
FROM observatorios;
```

No es conveniente abusar de este predicado ya que obligamos al servidor a analizar la estructura de la tabla para averiguar los campos que contiene, es mucho más rápido indicar el listado de campos deseados.

```
clima=# SELECT indentinm,x,y,z,nombre
FROM observatorios;
```

DISTINCT Omite los registros cuyos campos seleccionados coincidan totalmente. Con otras palabras el predicado DISTINCT devuelve aquellos registros cuyos campos indicados en la cláusula SELECT posean un contenido diferente.

```
clima=# SELECT DISTINCT indentinm,x,y,z,nombre
FROM observatorios;
```

DISTINC ON (campo) Omite registros que coincidan en el campo seleccionado. Por ejemplo la siguiente orden devuelve un sólo observatorio por valor de altitud:

```
clima=# SELECT DISTINCT ON (z) indentinm,x,y,z,nombre
FROM observatorios;
```

0.61.4. Alias

En determinadas circunstancias es necesario asignar un nuevo nombre a alguna de las columnas devueltas por el servidor. Para ello tenemos la palabra reservada AS que se encarga de asignar el nombre que deseamos a la columna deseada:

```
clima=# SELECT nombre,x AS longitud, y AS latitud, z AS altitud
FROM observatorios;
```

0.62. Criterios de Selección

En la sección anterior se vio la forma de recuperar los registros de las tablas, las formas empleadas devolvían todos los registros de la mencionada tabla, salvo que se untilizara el predicado DISTINCT. En esta sección se estudiarán las posibilidades de filtrar los registros con el fin de recuperar solamente aquellos que cumplan una condiciones preestablecidas.

0.62.1. La cláusula WHERE

La cláusula WHERE se usa para determinar qué registros de las tablas enumeradas en la cláusula FROM aparecerán en los resultados de la instrucción SELECT.

Por ejemplo, para obtener sólo los observatorios situados a más de 500 metros de altitud, la consulta adecuada sería:

```
clima=# SELECT nombre,x,y,z
FROM observatorios
WHERE z  500;
```

0.62.2. Operadores Lógicos

Los operadores lógicos soportados por SQL son: AND, OR, XOR, Eqv, Imp, Is y Not. A excepción de los dos últimos todos poseen la siguiente sintaxis:

```
<expresión1 operador <expresión2
```

En donde expresión1 y expresión2 son las condiciones a evaluar, el resultado de la operación varía en función del operador lógico. La tabla adjunta muestra los diferentes posibles resultados:

- Falso AND Verdad Falso
- Falso AND Falso Falso
- Verdad OR Falso Verdad

- Verdad OR Verdad Verdad
- Falso OR Verdad Verdad
- Falso OR Falso Falso

Si a cualquiera de las anteriores condiciones le anteponemos el operador NOT el resultado de la operación será el contrario al devuelto sin el operador NOT.

```
clima=# SELECT nombre,x,y,z
FROM observatorios
WHERE x  600000 AND x < 650000;
```

```
clima=# SELECT nombre,x,y,z
FROM observatorios
WHERE (x  600000 AND x < 650000) OR z<200;
```

La última consulta devolvera los observatorios situados entre los valores de X UTM de 600000 y 650000 UTM y aquellos con altitud inferior a 200 metros.

0.62.3. Intervalos de Valores

Para indicar que deseamos recuperar los registros según el intervalo de valores de un campo emplearemos el operador Between cuya sintaxis es:

```
campo [Not] Between valor1 And valor2
```

En este caso la consulta devolvería los registros que contengan en çampoün valor incluido en el intervalo valor1, valor2 (ambos inclusive). Si anteponemos la condición Not devolverá aquellos valores no incluidos en el intervalo:

```
clima=# SELECT nombre,x,y,z
FROM observatorios
WHERE x Between 600000 AND 650000;
```

esta orden es equivalente a

```
bd=# SELECT nombre,x,y,z
FROM observatorios WHERE x  600000 AND Edad < 650000;
```

0.62.4. El Operador

Se utiliza para comparar una expresión de cadena con un modelo en una expresión SQL. Su sintaxis es:

```
expresión modelo
```

En donde expresión es una variable y modelo un patrón de texto con el que se compara la expresión. Se puede utilizar este operador para encontrar valores en los campos que coincidan con el modelo especificado. Por modelo puede especificar un valor completo (Lorca), o se pueden utilizar caracteres comodín como los reconocidos por el sistema operativo para encontrar un rango de valores.

Cuadro 11: Posibilidades del operador Like

Tipo de coincidencia	Modelo Planteado	Coincide	No coincide
Varios caracteres	.ª*a"	.ªa", .ªBa", .ªBBBa"	.ªBC"
Carácter especial	.ª[*]a"	.ª*a"	.ªaa"
Varios caracteres	.ªb*"	.ªbcdefg", .ªbc"	çab", .ªab"
Un carácter	.ª?a"	.ªaa", .ª3a", .ªBa"	.ªBBBa"
Un dígito	.ª#a"	.ª0a", .ª1a", .ª2a"	.ªaa", .ª10a"
Rango de caracteres	"[a-z]"	"f", "p", "j"	"2", "&"
Fuera de un rango	"[!a-z]"	"9", "&", "%"	"b", .ª"
Distinto de un dígito	"[!0-9]"	.ª", .ª", "& ", " "	"0", "1", "9"
Combinada	.ª[!b-m]#"	.ªn9", .ªz0", .ª99"	.ªbc", .ªj0"

El operador se utiliza en una expresión para comparar un valor de un campo con una expresión de cadena. Por ejemplo, si introduces C* en una consulta SQL, la consulta devuelve todos los valores de campo que comiencen por la letra C. En una consulta con parámetros, puede hacer que el usuario escriba el modelo que se va a utilizar.

La tabla 11 muestra cómo utilizar el operador para comprobar expresiones con diferentes modelos. El siguiente ejemplo devolvería todos los observatorios en cuyo nombre apareciera incluida la palabra Lorca:

```
clima=# SELECT * from observatorios
WHERE nombre "Lorca";
```

0.62.5. El Operador In

Este operador devuelve aquellos registros cuyo campo indicado coincide con alguno de los en una lista. Su sintaxis es:

```
expresión [Not] In(valor1, valor2, . . .)
clima=# SELECT *
FROM observatorios
WHERE indentinm IN(7149,7069);
```

0.63. Agrupamiento de Registros

0.63.1. GROUP BY y HAVING

Combina los registros con valores idénticos, en la lista de campos especificados, en un único registro. Para cada registro se crea un valor sumario si se incluye una función SQL agregada, como por ejemplo Sum o Count, en la instrucción SELECT. Su sintaxis es:

```
SELECT campos FROM tabla WHERE criterio GROUP BY campos del grupo
```

GROUP BY es opcional. Los valores de resumen se omiten si no existe una función SQL agregada en la instrucción SELECT. Los valores Null en los campos GROUP BY se agrupan y no se omiten. No obstante, los valores Null no se evalúan en ninguna de las funciones SQL agregadas.

Se utiliza la cláusula WHERE para excluir aquellas filas que no se desea agrupar, y la cláusula HAVING para filtrar los registros una vez agrupados.

Como ejemplo vamos a hacer la primera consulta a una nueva tabla :

```
clima=# SELECT *
FROM menstem;
```

Esta tabla contiene los valores de temperatura mensual para algunos de los observatorios incluidos en la base de datos. Si a partir de esta tabla quisieramos conocer las temperaturas medias mensuales en el observatorio de Cajigal (identificador 7001E):

```
clima=# SELECT mes,AVG(tmed)
FROM menstem
WHERE ide="7001E"
GROUP BY mes;
```

si ademas añadimos lo siguiente:

```
clima=# SELECT mes,AVG(tmed),COUNT(tmed)
FROM menstem
WHERE ide="7001E"
GROUP BY mes;
```

Tenemos no sólo las medias sino también el número de años utilizado para calcular estas medias. Todos los campos de la lista de campos de SELECT deben o bien incluirse en la cláusula GROUP BY o como argumentos de una función SQL agregada.

Una vez que GROUP BY ha combinado los registros, HAVING muestra cualquier registro agrupado por la cláusula GROUP BY que satisfaga las condiciones de la cláusula HAVING.

HAVING es similar a WHERE, determina qué registros se seleccionan. Una vez que los registros se han agrupado utilizando GROUP BY, HAVING determina cuales de ellos se van a mostrar.

```
clima=# SELECT mes,AVG(tmed),COUNT(tmed)
FROM menstem
WHERE ide="7001E"
GROUP BY mes
HAVING AVG(tmed)20;
```

0.63.2. AVG

Calcula la media aritmética de un conjunto de valores contenidos en un campo especificado de una consulta. Su sintaxis es la siguiente

```
Avg(expr)
```

En donde expr representa el campo que contiene los datos numéricos para los que se desea calcular la media o una expresión que realiza un cálculo utilizando los datos de dicho campo. La media calculada por Avg es la media aritmética (la suma de los valores dividido por el número de valores). La función Avg no incluye ningún campo Null en el cálculo.

```
clima=# SELECT Avg(tmed) AS Promedio
FROM menstem
WHERE ide="7001E";
```

0.63.3. Count

Calcula el número de registros devueltos por una consulta. Su sintaxis es la siguiente

```
Count(expr)
```

En donde expr contiene el nombre del campo que desea contar. Los operandos de expr pueden incluir el nombre de un campo de una tabla, una constante o una función (la cual puede ser intrínseca o definida por el usuario pero no otras de las funciones agregadas de SQL). Puede contar cualquier tipo de datos incluso texto.

Aunque expr puede realizar un cálculo sobre un campo, Count simplemente cuenta el número de registros sin tener en cuenta qué valores se almacenan en los registros. La función Count no cuenta los registros que tienen campos null a menos que expr sea el carácter comodín asterisco (*). Si utiliza un asterisco, Count calcula el número total de registros, incluyendo aquellos que contienen campos null. Count(*) es considerablemente más rápida que Count(Campo). No se debe poner el asterisco entre dobles comillas ("*").

```
clima=# SELECT Count(*) AS Total
FROM observatorios;
```

Si expr identifica a múltiples campos, la función Count cuenta un registro sólo si al menos uno de los campos no es Null. Si todos los campos especificados son Null, no se cuenta el registro. Hay que separar los nombres de los campos con ampersand (&).

```
clima=# SELECT Count(x & y & z) AS Total
FROM observatorios;
```

0.63.4. Max, Min

Devuelven el mínimo o el máximo de un conjunto de valores contenidos en un campo especifico de una consulta. Su sintaxis es:

```
Min(expr)
Max(expr)
```

En donde expr es el campo sobre el que se desea realizar el cálculo. Expr pueden incluir el nombre de un campo de una tabla, una constante o una función (la cual puede ser intrínseca o definida por el usuario pero no otras de las funciones agregadas de SQL.

```
clima=# SELECT Min(Gastos) AS ElMin
FROM Pedidos
WHERE Pais = "España";
```

```
clima=# SELECT Max(Gastos) AS ElMax
FROM Pedidos
WHERE Pais = "España";
```

0.63.5. Stddev

Devuelve estimaciones de la desviación estándar para la población (el total de los registros de la tabla) o una muestra de la población representada (muestra aleatoria) . Su sintaxis es:

```
STDDEV(expr)
```

En donde expr representa el nombre del campo que contiene los datos que desean evaluarse o una expresión que realiza un cálculo utilizando los datos de dichos campos. Los operandos de expr pueden incluir el nombre de un campo de una tabla, una constante o una función (la cual puede ser intrínseca o definida por el usuario pero no otras de las funciones agregadas de SQL)

StDevP evalúa una población, y StDev evalúa una muestra de la población. Si la consulta contiene menos de dos registros devuelve un valor Null (el cual indica que la desviación estándar no puede calcularse). La siguiente consulta:

```
clima=# SELECT mes,Stddev(tmed) AS desviacion,count(tmed) AS datos
FROM menstem
WHERE ide="70001E"
GROUP BY mes;
```

devolverá la desviación típica y el tamaño muestral de la precipitación mensual en el observatorio 70001E (Aguilas Montagro).

0.63.6. Sum

Devuelve la suma del conjunto de valores contenido en un campo especifico de una consulta. Su sintaxis es:

```
Sum(expr)
```

En donde expr respresenta el nombre del campo que contiene los datos que desean sumarse o una expresión que realiza un cálculo utilizando los datos de dichos campos. Los operandos de expr pueden incluir el nombre de un campo de una tabla, una constante o una función (la cual puede ser intrínseca o definida por el usuario pero no otras de las funciones agregadas de SQL). En el siguiente ejemplo vamos a utilizar la tabla menspluv que contiene datos de precipitación mensual para obtener una tabla de precipitación anual.

```
clima=# SELECT ano,sum(pluv) AS precipitacion, count(pluv) as meses
   from menspluv
WHERE pluv=0 AND ide="7094" GROUP BY ano;
```

Date cuenta de que creamos una nueva variable denominada meses que almacena el número de meses utilizados para calcular la suma y sirve para verificar que el año este completo en la tabla.

0.63.7. Limitar el número de registros devueltos por el servidor

Finalmente, puede limitarse el número de registros devueltos por la orden SELECT utilizando el modificador LIMIT. Su sintaxis sería:

```
SELECT FROM tabla LIMIT número
```

lo que devolvería sólo número registros. Resulta útil para consultas del tipo ¿Cuales son los 5 observatorios situados a mayor altitd?

```
clima=# SELECT nombre,z
FROM observatorios
ORDER BY z DESC
LIMIT 5;
```

0.63.8. Consultas a varias tablas

Pueden combinarse varias tablas mediante operaciones de SQL más complejas. Cuando se combinan tablas suele introducirse un alias simplificado para las tablas (la primetra letra por ejemplo). En el siguiente ejemplo se va a utilizar, además de la tabla observatorios, una nueva tabla llamada menstem que contiene datos acerca de la temperatura mensual en los observatorios utilizados. Puedes ver los contenidos de esta tabla con la orden menstem y comprobar que existe una columna denominada ide que actúa como campo común con la columna indentinm de la tabla observatorios.

Vamos a obtener una tabla que nos podría servir para interpolar la temperatura media del mes de Julio de 1990. Los primero que vamos a hacer es ver con cuantos datos dispondríamos para la intrpolación:

```
bd=#SELECT *
WHERE ano=1990 and mes=7;
```

A continuación haremos una consulta que combine la tabla observatorios con la tabla menstem para obtener una tabla con la que poder hacer la interpolación:

```
clima=#SELECT x,y,z,tmed
FROM observatorios, menstem
WHERE ide=indentinm and mes=7 and ano=1990;
```

Como ves no hay grandes diferencias, es necesario declarar las dos tablas tras el modificador WHERE y especificar que el identificador de los observatorios, que actúa en este caso como campo común, debe ser igual para combinar los registros. Este es un caso especialmente simple porque no hay nombres de columna repetidos en ambas tablas, en caso de que esto hubiera sido así deberíamos haber utilizado la sintaxis:

```
clima=#SELECT o.x,o.y,o.z,t.tmed
FROM observatorios o, menstem t
WHERE t.ide=o.indentinm and mes=7 and ano=1990;
```

donde especificamos a que tabla pertenece cada nombre de columna (tabla.columna) utilizando un alias en lugar de los nombres de las tablas (o y t) para abreviar.

0.64. subconsultas

Una subconsulta es una instrucción SELECT escrita ente paréntesis y anidada dentro de una instrucción SELECT, SELECT...INTO, INSERT...INTO, DELETE, o UPDATE o dentro de otra subconsulta. Se puede utilizar una subconsulta en lugar de una expresión en la lista de campos de una instrucción SELECT o en una cláusula WHERE o HAVING.

En una subconsulta, se utiliza la instrucción SELECT para proporcionar un conjunto de uno o más valores especificados para evaluar en la expresión de la cláusula WHERE o HAVING de la consulta principal. Por ejemplo la siguiente consulta utiliza una subconsulta para calcular la altitud media de los observatorios y devolver aquellos observatorios situads a mayor altitud:

```
clima=# SELECT *
FROM observatorios
WHERE z  (
SELECT AVG(z)
FROM observatorios
);
```

En este caso la comparación se realiza con un operador simple (mayor que) ya que la subconsulta devuelve un sólo valor, en el caso de que la subconsulta devuelva varios valores es necesario utilizar predicados de comparación más sofisticados como ANY, ALL, IN o EXISTS.

0.64.1. ANY

Se puede utilizar el predicado ANY o SOME, los cuales son sinónimos, para recuperar registros de la consulta principal, que satisfagan la comparación con cualquier otro registro recuperado en la subconsulta. El ejemplo siguiente devuelve todos los productos cuyo precio unitario es mayor que el de cualquier producto vendido con un descuento igual o mayor al 25 por ciento.:

```
clima=# SELECT *
FROM observatorios
WHERE z  ANY (
SELECT *
FROM observatorios
WHERE nombre "Lorca"
);
```

Devolvera aquellos observatorios cuya altitud sea mayor que la de cualquiera de los observatorios que contienen “Lorca” en su nombre, es decir devolverá los situados a mayor altura que el más bajo de estos.

0.64.2. ALL

El predicado ALL se utiliza para recuperar únicamente aquellos registros de la consulta principal que satisfacen la comparación con todos los registros recuperados en la subconsulta. Si se cambia ANY por ALL en el ejemplo anterior, la consulta devolverá únicamente aquellos productos cuyo precio unitario sea mayor que el de todos los productos vendidos con un descuento igual o mayor al 25 por ciento. Esto es mucho más restrictivo.

```
clima=# SELECT *
FROM observatorios
WHERE z < ALL (
SELECT *
FROM observatorios
WHERE nombre "Lorca"
);
```

Devolverá aquellos observatorios cuya altitud sea mayor que la de todos los observatorios que contienen “Lorca” en su nombre, es decir devolverá los situados a mayor altura que el más alto de estos.

0.64.3. IN

El predicado IN se emplea para recuperar únicamente aquellos registros de la consulta principal para los que algunos registros de la subconsulta contienen un valor igual.

```
clima=# SELECT *
FROM observatorios
WHERE z IN (
SELECT *
FROM observatorios
WHERE nombre "Lorca"
);
```

devolvera aquellos observatorios cuya altitud sea igual a la altitud de algunos de los observatorios que incluyen la palabra "Lorca.[en] su numbre.

Inversamente se puede utilizar NOT IN para recuperar únicamente aquellos registros de la consulta principal para los que no hay ningún registro de la subconsulta que contenga un valor igual.

0.64.4. EXISTS

El predicado EXISTS (con la palabra reservada NOT opcional) se utiliza en comparaciones de verdad/falso para determinar si la subconsulta devuelve algún registro.

```
clima=# SELECT *
FROM observatorios
WHERE EXISTS (
SELECT z
FROM observatorios
WHERE nombre "Lorca");
```

Devolverá todos los registros porque en la base de datos existen observatorios cuyo nombre incluye la palabra "Lorca".

```
clima=# SELECT *
FROM observatorios
WHERE EXISTS (
SELECT z
FROM observatorios
WHERE nombre "Pamplona");
```

No devolverá ningún registro porque en la base de datos no existen observatorios cuyo nombre incluya la palabra "Pamplona".

Se puede utilizar también alias del nombre de la tabla en una subconsulta para referirse a tablas listadas en la cláusula FROM fuera de la subconsulta.

0.65. Consultas de Acción

Las consultas de acción son aquellas que no devuelven ningún registro, son las encargadas de acciones como añadir y borrar y modificar registros.

0.65.1. DELETE

Crea una consulta de eliminación que elimina los registros de una o más de las tablas listadas en la cláusula FROM que satisfagan la cláusula WHERE. Esta consulta elimina los registros completos, no es posible eliminar el contenido de algún campo en concreto. Su sintaxis es:

```
DELETE FROM Tabla WHERE criterio
```

Si desea eliminar todos los registros de una tabla, eliminar la propia tabla es más eficiente que ejecutar una consulta de borrado.

Una vez que se han eliminado los registros utilizando una consulta de borrado, no puede deshacerse la operación. Si quiere saber qué registros se eliminarán, primero examina los resultados de una consulta de selección que utilice el mismo criterio y después ejecuta la consulta de borrado. En todo caso es conveniente tener copias de seguridad de las tablas involucradas en una consulta de eliminación. Si se eliminan los registros equivocados podrás recuperarlos desde las copias de seguridad.

```
clima=# DELETE
FROM observatorios
WHERE x <1000;
```

0.65.2. INSERT INTO

Agrega un registro en una tabla. Se la conoce como una consulta de datos añadidos. Esta consulta puede ser de dos tipo: Insertar un único registro ó Insertar en una tabla los registros contenidos en otra tabla.

0.65.2.1. Para insertar un único Registro:

En este caso la sintaxis es la siguiente:

```
bd=# INSERT INTO Tabla (campo1, campo2, .., campoN)
VALUES (valor1, valor2, ..., valorN);
```

Esta consulta guarda en el campo1 el valor1, en el campo2 y valor2 y así sucesivamente. Hay que prestar especial atención a acotar entre comillas simples (') los valores literales (cadenas de caracteres).

0.65.2.2. Para insertar Registros de otra Tabla:

En este caso la sintaxis es:

```
bd=# INSERT INTO Tabla (campo1, campo2, ..., campoN)
SELECT TablaOrigen.campo1, TablaOrigen.campo2, ..., TablaOrigen.campoN
FROM TablaOrigen;
```

En este caso se seleccionarán los campos 1,2, ..., n dela tabla origen y se grabarán en los campos 1,2,.., n de la Tabla. La condición SELECT puede incluir la cláusula WHERE para filtrar los registros a copiar. Si Tabla y TablaOrigen poseen la misma estrucutra podemos simplificar la sintaxis a:

```
bd=#INSERT INTO Tabla
SELECT TablaOrigen.*
FROM TablaOrigen;
```

De esta forma los campos de TablaOrigen se grabarán en Tabla, para realizar esta operación es necesario que todos los campos de TablaOrigen estén contenidos con igual nombre en Tabla. Con otras palabras que Tabla posea todos los campos de TablaOrigen (igual nombre e igual tipo).

En este tipo de consulta hay que tener especial atención con los campos contadores o autonuméricos puesto que al insertar un valor en un campo de este tipo se escribe el valor que contenga su campo homólogo en la tabla origen, no incrementandose como le corresponde.

Se puede utilizar la instrucción INSERT INTO para agregar un registro único a una tabla, utilizando la sintaxis de la consulta de adición de registro único tal y como se mostró anteriormente. En este caso, su código específica el nombre y el valor de cada campo del registro. Debe especificar cada uno de los campos del registro al que se le va a asignar un valor así como el valor para dicho campo. Cuando no se especifica dicho campo, se inserta el valor predeterminado o Null. Los registros se agregan al final de la tabla.

También se puede utilizar INSERT INTO para agregar un conjunto de registros pertenecientes a otra tabla o consulta utilizando la cláusula SELECT ... FROM como se mostró anteriormente en la sintaxis de la consulta de adición de múltiples registros. En este caso la cláusula SELECT especifica los campos que se van a agregar en la tabla destino especificada.

La tabla destino u origen puede especificar una tabla o una consulta.

Si la tabla destino contiene una clave principal, hay que segurarse que es única, y con valores no-Null ; si no es así, no se agregarán los registros. Si se agregan registros a una tabla con un campo Contador , no se debe incluir el campo Contador en la consulta. Se puede emplear la cláusula IN para agregar registros a una tabla en otra base de datos.

Se pueden averiguar los registros que se agregarán en la consulta ejecutando primero una consulta de selección que utilice el mismo criterio de selección y ver el resultado. Una consulta de adición copia los registros de una o más tablas en otra. Las tablas que contienen los registros que se van a agregar no se verán afectadas por la consulta de adición. En lugar de agregar registros existentes en otra tabla, se puede especificar los valores de cada campo en un nuevo registro utilizando la cláusula VALUES. Si se omite la lista de campos, la cláusula VALUES debe incluir un valor para cada campo de la tabla, de otra forma fallará INSERT.

```
clima=#INSERT INTO observatorios (indentinm,nombre, x,y,z,ident,obs)
VALUES ("9999", "Murcia inventado",650000,4200000,\quad,450);
```

0.65.3. UPDATE

Crea una consulta de actualización que cambia los valores de los campos de una tabla especificada basándose en un criterio específico. Su sintaxis es:

```
bd=#UPDATE Tabla
SET Campo1=Valor1, Campo2=Valor2, ... CampoN=ValorN
WHERE Criterio;
```

UPDATE es especialmente útil cuando se desea cambiar un gran número de registros o cuando éstos se encuentran en múltiples tablas. Puede cambiar varios campos a la vez:

```
clima=#UPDATE observatorios
SET indentinm = 8888
WHERE obs = 450;
```

UPDATE no genera ningún resultado. Para saber qué registros se van a cambiar, hay que examinar primero el resultado de una consulta de selección que utilice el mismo criterio y después ejecutar la consulta de actualización.

```
clima=#SELECT *
WHERE obs=450; clima=#UPDATE observatorios
SET indentinm = 8888
WHERE obs = 450;
```

Si en una consulta de actualización suprimimos la cláusula WHERE todos los registros de la tabla señalada serán actualizados, por lo que hay que tener precaución con este tipo de consulta.

```
clima=#UPDATE observatorios
SET nombre = "te has cargado la tabla";
```

0.66. R como cliente de PostgreSQL

El programa psql es simplemente uno de los muchos clientes de bases de datos disponibles. Existe incluso la posibilidad de convertir cualquier programa en un cliente de base de datos añadiendole las funciones para lanza consultas al servidor e interpretar las respuestas de este. Se han desarrollado multitud de paquetes que amplian las capacidades del programa R. Uno de estos es RPgSQL que incorpora funciones para conectar con PostgreSQL. Una típica sesión de trabajo sería:

```
library(RPgSQL)
db.connect(dbname="clima",user="usuario")
orden="select indentinm,x,y,z from observatorios;"
db.execute(orden,clear=F)
datos $\leftarrow$db.fetch.result()
db.clear.result()
```

- 1. Carga el paquete RPgSQL dando acceso a sus funciones;
- 2. Conecta a la base de datos (en algunos casos será necesario pasar una contraseña);
- 3. Almacena el texto de una consulta en una variable;
- 4. Lanza la consulta al servidor;
- 5. Almacena los resultados en una data.frame llamado datos;
- 6. Borra de la memoria los resultados de la consulta (esta última no es necesaria pero sirve para liberar memoria del computador).

Una vez que se dispone de los resultados de la consulta en un data.frame, podemos utilizar cualquiera de las funciones de R para analizar los resultados:

```
plot(datos$x,datos$y)
```

Tipos de datos

Mostramos los tipos de datos (data types) que usa y admite el motor de base de datos gratuito y open source PostgreSQL. Mostramos todos los tipos de datos de propósito general, los tipos de datos de PostgreSQL numéricos, monetarios, de red, geométricos, carácter, fecha/hora, etc. Tipos de datos de propósito general en PostgreSQL A continuación, el listado de los tipos de datos (data types) del motor de base de datos gratuito PostgreSQL. Mostramos los tipos de datos de carácter o propósito general, los más habituales:

Cuadro 12: Tipos de datos de propósito general

Tipo de datos	Alias	Descripción
bigint	int8	Entero con signo de 8 bytes
bigserial	serial8	Autoincremento entero de 8 bytes
bit		Cadena de bit de longitud fija
bit varying(n)	varbit(n)	Cadena de bit de longitud variable
boolean	bool	Lógico (true/false)
box		Rectángulo en el plano
bytea		Datos binarios
character varying(n)	varchar(n)	Cadena de caracteres de longitud variable
character(n)	char(n)	Cadena de caracteres de longitud fija
cidr		Dirección IP de red (IPv4 o IPv6)
circle		Círculo en el plano
date		Fecha (año, mes, día)
double precision	float8	Número de punto flotante de precisión doble
inet		Dirección de un host de red (IPv4 or IPv6)
integer	int, int4	Enterio con signo, 4 bytes

(Continúa...)

Cuadro 12: Tipos de datos de propósito general

Tipo de datos	Alias	Descripción
interval(p)		Intervalo de tiempo
line		Línea infinita en el plano (no se aplica completamente)
lseg		Segmento de línea en el plano
macaddr		Dirección MAC de tarjeta o dispositivo de red
money		Moneda
numeric [(p, s)] decimal [(p, s)]		Numérico exacto con precisión modificable
path		Trazado geométrico abierto y cerrado en el plano
point		Punto geométrico en el plano
polygon		Polígono cerrado geométrico en el plano
real	float4	Número de punto flotante de precisión simple
smallint	int2	Entero con signo de 2 bytes
serial	serial4	Autoincremento, entero de 4 bytes
text		Cadena de caracteres de longitud variable
time [(p)] [sin zona horaria]		Hora del día
time [(p)] con zona horaria	timetz	Hora del día, incluyendo la zona horaria
timestamp [(p)] [sin zona horaria]	timestamp	Fecha y hora
timestamp [(p)] con zona horaria	timestamptz	Fecha y hora incluyendo la zona horaria

(Final)

Tipos numéricos en PostgreSQL

A continuación mostramos los tipos de datos numéricos de PostgreSQL:

Cuadro 13: Tipos numéricos

Nombre	Tamaño	Descripción	Rango
smallint	2 bytes	Entero de rango pequeño	De -32768 a +32767
integer	4 bytes	Selección habitual para tipos enteros	De -2147483648 a +2147483647
bigint	8 bytes	Entero de rango largo	De -9223372036854775808 a 9223372036854775807
decimal	variable	Precisión especificada por el usuario, exacto	Sin límite
numeric	variable	Precisión especificada por el usuario, exacto	Sin límite
real	4 bytes	Variable/precisión, inexacto	6 dígitos decimales de precisión
double precision	8 bytes	Variable/precisión, inexacto	15 dígitos decimales de precisión
serial	4 bytes	Autoincremento simple	De 1 a 2147483647
bigserial	8 bytes	Autoincremento largo	De 1 a 9223372036854775807

Tipos de datos monetarios (moneda) en PostgreSQL

El tipo de datos de PostgreSQL para valores de moneda es:

Cuadro 14: Tipos de datos monetarios (moneda)

Nombre	Tamaño	Descripción	Rango
money	4 bytes	Moneda	De -21474836.48 a +21474836.47

Tipos de datos carácter en PostgreSQL

Los tipos de datos del motor de base de datos gratuito y open source PostgreSQL de tipo carácter son:

Cuadro 15: Tipos de datos carácter

Nombre	Descripción
character varying(n), varchar(n)	De longitud variable, con límite
character(n), char(n)	De longitud fija
text	De longitud variable, ilimitado

Tipos de datos binarios en PostgreSQL

El tipo de datos binario de PostgreSQL es:

Cuadro 16: Tipos de datos binarios

Nombre	Tamaño	Descripción
bytea	4 bytes además de la cadena binaria actual	Cadena binaria de longitud variable

Tipos de datos Fecha/Hora en PostgreSQL

Los tipos de datos de fecha y hora del motor de base de datos PostgreSQL son:

Cuadro 17: Tipos de datos Fecha/Hora

Nombre	Tamaño	Descripción	Valor bajo	Valor alto	Resolución
timestamp [(p)] [sin zona horaria]	8 bytes	Fecha y hora	4713 BC	5874897 AD	1 microsegundo / 14 dígitos
timestamp [(p)] con zona horaria	8 bytes	Fecha y hora con zona horaria	4713 BC	5874897 AD	1 microsegundos / 14 dígitos
interval [(p)]	12 bytes	Intervalo de hora	-178000000 años	178000000 años	1 microsegundo date 4 bytes Sólo fecha 4713 BC 32767 AD 1 día
time [(p)] [sin zona horaria]	8 bytes	Sólo hora del día	00:00:00.00	23:59:59.99	1 microsegundo
time [(p)] con zona horaria	12 bytes	Horas del día con zona horaria	00:00:00.00+12	23:59:59.99-12	1 microsegundo

Tipos de datos geométricos en PostgreSQL

Los tipos de datos para valores geométricos del motor de base de datos PostgreSQL son:

Cuadro 18: Tipos de datos geométricos

Nombre	Tamaño	Representación	Descripción
point	16 bytes	Punto del plano	(x,y)
line	32 bytes	Línea infinita en el plano	((x1,y1),(x2,y2))
lseg	32 bytes	Segmento de línea en el plano	((x1,y1),(x2,y2))
box	32 bytes	Rectángulo en el plano	((x1,y1),(x2,y2))
path	16+16n bytes	Trazado geométrico cerrado en el plano	((x1,y1),...)
path	16+16n bytes	Trazado geométrico abierto en el plano	[(x1,y1),...]
polygon	40+16n bytes	Polígono (similar a trazado cerrado)	((x1,y1),...)
circle	24 bytes	Círculo	<(x,y),r>(centro y radio)

Tipos de datos de direcciones de red en PostgreSQL

Los tipos de datos para direcciones de red y mac de PostgreSQL son:

Cuadro 19: Tipos de datos binarios

Nombre	Tamaño	Descripción
cidr	12 o 24 bytes	Redes IPv4 o IPv6
inet	12 o 24 bytes	Hosts y redes IPv4 o IPv6
macaddr	6 bytes	Dirección MAC

0.67. Definidos por el usuario

En PostgreSQL es posible que el usuario defina sus propios tipos de datos. Estos tipos de datos son creados de una forma similar a los tipos de datos de C. Como en PostgreSQL es posible crear funciones, tipos de datos, operadores, etc... es posible también que el usuario pueda crear todo conjunto de utilerías para este nuevo tipo de datos y pueda interactuar con otros tipos de datos. Para hacer esto es necesario utilizar el comando DDL CREATE TYPE. Veamos un ejemplo de como se aplica este comando:

```
1 CREATE TYPE inventory_item AS (
2 name            text,
3 supplier_id     integer,
4 price           numeric
5 );
```

Lo cual generará un tipo de dato que puede ser utilizado en toda la base de datos. A continuación podremos ver como se crea una tabla a partir de este nuevo tipo de dato e insertaremos un valor dentro de este tipo de dato mediante INSERT.

```
CREATE TABLE on_hand (
item      inventory_item,
count     integer
);
INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

Para comprobar que si se ha creado nuestro tipo de dato podemos apreciar los tipo de datos agregados mediante el comando \dT:

```
test=# \dT
List of data types
Schema |      Name      | Description
--------+----------------+-------------
public | complex        |
public | inventory_item |
public | newtype        |
(3 rows)
Operadores
```

Funciones

La forma más fácil de implementar funciones es utilizar el lenguaje SQL. Una función SQL nos permite dar un nombre a uno o varios comandos sql.

Luego la sintaxis para implementar una función SQL:

```
create or replace function [nombre de la función]([parámetros]) returns
    [tipo de dato que retorna]
as [comandos sql]
language sql
```

Como primer problema implementaremos una función que reciba dos enteros y retorne la suma de los mismos:

```
create or replace function sumar(integer,integer) returns integer
AS
'select $1+$2;'
language sql;
Cada parámetro se lo accede luego mediante la posición que ocupa y se
   le antecede el caracter \$. El o los comandos SQL deben ir entre
   simples comillas (si tenemos que utilizar las simples comillas en el
    comando SQL debemos disponer dos simples comillas seguidas) y
   separados por punto y coma. Luego indicamos al final que se trata de
    una función SQL.

Para llamar luego a esta función lo hacemos por ejemplo en un select:

\begin{lstlisting}
select sumar(3,4);
Podemos acceder perfectamente a una o más tablas en la función.
   Confeccionaremos una función que acceda a la tabla usuarios:

\begin{lstlisting}
create table usuarios (
nombre varchar(30),
clave varchar(10)
);
y rescate la clave de un usuario que le pasamos como parámetro:

\begin{lstlisting}
create or replace function retornarclave(varchar) returns varchar
as
'select clave from usuarios where nombre=$1;'
language sql;
```

```
Luego para probar la función retornarclave debemos llamarla por ejemplo
    desde un select:

\begin{lstlisting}
select retornarclave('Susana');
Ingresemos el siguiente lote de comandos SQL en pgAdmin:
create or replace function sumar(integer,integer) returns integer
AS
'select $1+$2;'
language sql;

-- Llamamos la función que acabamos de crear:
\begin{lstlisting}
select sumar(3,4);

\begin{lstlisting}
drop table if exists usuarios;

-- Creamos la tabla usuarios:
\begin{lstlisting}
create table usuarios (
nombre varchar(30),
clave varchar(10)
);

insert into usuarios (nombre, clave) values ('Marcelo','Boca');
insert into usuarios (nombre, clave) values ('JuanPerez','Juancito');
insert into usuarios (nombre, clave) values ('Susana','River');
insert into usuarios (nombre, clave) values ('Luis','River');

-- Creamos una función que reciba una cadena con el nombre de usuario
-- y retorne la clave de dicho usuario:
\begin{lstlisting}
create or replace function retornarclave(varchar) returns varchar
as
'select clave from usuarios where nombre=$1;'
language sql;

-- Llamamos la función recuperando la clave del usuario llamada 'Susana
    ':
\begin{itemize}
select retornarclave('Susana');
\section{Extendiendo SQL: Funciones}
Parte de definir un tipo nuevo es la definición de funciones que
    describen su comportamiento. Como consecuencia, mientras que es
    posible definir una nueva función sin definir un tipo nuevo, lo
    contrario no es cierto. Por ello describimos como añadir nuevas
    funciones para Postgres antes de describir cómo añadir nuevos tipos
    .\salto
Postgres SQL proporciona tres tipos de funciones:
\begin{itemize}
\item funciones de lenguaje de consultas (funciones escritas en SQL)
```

```
\item funciones de lenguaje procedural (funciones escritas en, por
   ejemplo, PLTCL o PLSQL)
\item funciones de lenguaje de programación (funciones escritas en un
   lenguaje de programación compilado tales como C)
\end{itemize}
Cada clase de función puede tomar un tipo base, un tipo compuesto o
   alguna combinación como argumentos (parámetros). Además, cada clase
   de funcón puede devolver un tipo base o un tipo compuesto. Es más fá
   cil definir funciones SQL, así que empezaremos con ellas. Los
   ejemplos en esta sección se puede encontrar también en funcs.sql y
   funcs.c.
\section{Funciones de Lenguaje de Consultas (SQL)}
Las funciones SQL ejecutan una lista arbitraria de consultas SQL,
   devolviendo los resultados de la última consulta de la lista. Las
   funciones SQL en general devuelven conjuntos. Si su tipo de retorno
   no se especifica como un setof, entonces un elemento arbitrario del
   resultado de la última consulta será devuelto.\salto
El cuerpo de una función SQL que sigue a AS debería ser una lista de
   consultas separadas por caracteres espacio en blanco y entre paré
   ntesis dentro de comillas simples. Notar que las comillas simples
   usadas en las consultas se deben escribir como símbolos de escape,
   precediéndolas con dos barras invertidas.\salto
Los argumentos de la función SQL se pueden referenciar en las consultas
    usando una sintaxis $n: $1 se refiere al primer argumento, $2 al
   segundo, y así sucesivamente. Si un argumento es complejo, entonces
   una notación dot (por ejemplo "$1.emp") se puede usar para acceder a
    las propiedades o atributos del argumento o para llamar a funciones
   .

Ejemplos
Para ilustrar una función SQL sencilla, considere lo siguiente, que se
   podría usar para cargar en una cuenta bancaria:
\begin{lstlisting}
create function TP1 (int4, float8) returns int4
as 'update BANK set balance = BANK.balance - $2
where BANK.acctountno = $1
select(x = 1)'
language 'sql';
```

Un usuario podría ejecutar esta función para cargar $100.00 en la cuenta 17 de la siguiente forma:

```
select (x = TP1( 17,100.0));
```

El más interesante ejemplo siguiente toma una argumento sencillo de tipo EMP, y devuelve resultados múltiples:

```
select function hobbies (EMP) returns set of HOBBIES
as 'select (HOBBIES.all) from HOBBIES
where $1.name = HOBBIES.person'
language 'sql';
```

0.68. Funciones SQL sobre Tipos Base

La función SQL más simple posible no tiene argumentos y sencillamente devuelve un tipo base, tal como int4:

```
1  CREATE FUNCTION one() RETURNS int4
2  AS 'SELECT 1 as RESULT' LANGUAGE 'sql';
3
4  SELECT one() AS answer;
5
6  +-------+
7  |answer |
8  +-------+
9  |1      |
10 +-------+
```

Notar que definimos una lista objetivo para la función (con el nombre RESULT), pero la lista objetivo de la consulta que llamó a la función sobreescribió la lista objetivo de la función. Por esto, el resultado se etiqueta answer en vez de one.

Es casi tan fácil definir funciones SQL que tomen tipos base como argumentos. En el ejemplo de abajo, note cómo nos referimos a los argumentos dentro de la función como $1 y $2:

```
1  CREATE FUNCTION add_em(int4, int4) RETURNS int4
2  AS 'SELECT $1 + $2;' LANGUAGE 'sql';
3
4  SELECT add_em(1, 2) AS answer;
5
6  +-------+
7  |answer |
8  +-------+
9  |3      |
10 +-------+
```

0.69. Funciones SQL sobre Tipos Compuestos

Al especificar funciones con argumentos de tipos compuestos (tales como EMP), debemos no solo especificar qué argumento queremos (como hicimos más arriba con $1 y 2$) sino también los atributos de ese argumento. Por ejemplo, observe la función double_salary que procesa cual sería su salario si se doblase:

```
1  CREATE FUNCTION double_salary(EMP) RETURNS int4
2  AS 'SELECT $1.salary * 2 AS salary;' LANGUAGE 'sql';
3
4  SELECT name, double_salary(EMP) AS dream
5  FROM EMP
6  WHERE EMP.cubicle ~= '(2,1)'::point;
7
8
9  +-----+-------+
10 |name | dream |
11 +-----+-------+
```

```
|Sam   | 2400   |
+-----+-------+
```

Note el uso de la sintaxis $1.salary. Antes de adentrarnos en el tema de las funciones que devuelven tipos compuestos, debemos presentar primero la notación de la función para proyectar atributos. La forma sencilla de explicar esto es que podemos normalmente usar la notación atributo(clase) y clase.atributo indistintamente: – – esto es lo mismo que: – SELECT EMP.name AS youngster FROM EMP WHERE EMP.age <30 –

```
SELECT name(EMP) AS youngster
FROM EMP
WHERE age(EMP) < 30;

+----------+
|youngster |
+----------+
|Sam       |
+----------+
```

Como veremos, sin embargo, no siempre es este el caso. Esta notación de función es importante cuando queremos usar una función que devuelva una única instancia. Hacemos esto embebiendo la instancia completa dentro de la función, atributo por atributo. Esto es un ejemplo de una función que devuelve una única instancia EMP:

```
CREATE FUNCTION new_emp() RETURNS EMP
AS 'SELECT \'None\'::text AS name,
1000 AS salary,
25 AS age,
\'(2,2)\'::point AS cubicle'
LANGUAGE 'sql';
```

En este caso hemos especificado cada uno de los atributos con un valor constante, pero cualquier computación o expresión se podría haber sustituido por estas constantes. Definir una función como esta puede ser delicado. Algunos de las deficiencias más importantes son los siguientes:
La orden de la lista objetivo debe ser exactamente la misma que aquella en la que los atributos aparezcan en la orden CREATE TABLE (o cuando ejecute una consulta .*).

Se debe encasillar las expresiones (usando ::) muy cuidadosamente o verá el siguiente error:

```
WARN::function declared to return type EMP does not retrieve (EMP.*)
```

Al llamar a una función que devuelva una instancia, no podemos obtener la instancia completa. Debemos o bien proyectar un atributo fuera de la instancia o bien pasar la instancia completa a otra función.

```
SELECT name(new_emp()) AS nobody;

+-------+
|nobody |
+-------+
|None   |
+-------+
```

La razón por la que, en general, debemos usar la sintaxis de función para proyectar los atributos de los valores de retorno de la función es que el parser no comprende la otra sintaxis (dot) para la proyección cuando se combina con llamadas a funciones.

```
SELECT new_emp().name AS nobody;
WARN:parser: syntax error at or near "."
```

Cualquier colección de ordenes en el lenguaje de consulta SQL se pueden empaquetar juntas y se pueden definir como una función. Las ordenes pueden incluir updates (es decir, consultas INSERT, UPDATE, y DELETE) así como SELECT. Sin embargo, la orden final debe ser un SELECT que devuelva lo que se especifique como el tipo de retorno de la función.

```
CREATE FUNCTION clean_EMP () RETURNS int4
AS 'DELETE FROM EMP WHERE EMP.salary <= 0;
SELECT 1 AS ignore_this'
LANGUAGE 'sql';

SELECT clean_EMP();

+--+
|x |
+--+
|1 |
+--+
```

Vistas

Una vista es una alternativa para mostrar datos de varias tablas. Una vista es como una tabla virtual que almacena una consulta. Los datos accesibles a través de la vista no están almacenados en la base de datos como un objeto.

Entonces, una vista almacena una consulta como un objeto para utilizarse posteriormente. Las tablas consultadas en una vista se llaman tablas base. En general, se puede dar un nombre a cualquier consulta y almacenarla como una vista.

Una vista suele llamarse también tabla virtual porque los resultados que retorna y la manera de referenciarlas es la misma que para una tabla.

Las vistas permiten:

- ocultar información: permitiendo el acceso a algunos datos y manteniendo oculto el resto de la información que no se incluye en la vista. El usuario solo puede consultar la vista.
- simplificar la administración de los permisos de usuario: se pueden dar al usuario permisos para que solamente pueda acceder a los datos a través de vistas, en lugar de concederle permisos para acceder a ciertos campos, así se protegen las tablas base de cambios en su estructura.
- mejorar el rendimiento: se puede evitar tipear instrucciones repetidamente almacenando en una vista el resultado de una consulta compleja que incluya información de varias tablas.

Podemos crear vistas con: un subconjunto de registros y campos de una tabla; una unión de varias tablas; una combinación de varias tablas; un resumen estadístico de una tabla; un subconjunto de otra vista, combinación de vistas y tablas.

Una vista se define usando un "select".

La sintaxis básica parcial para crear una vista es la siguiente:

```
1 create view NOMBREVISTA as
2  SENTENCIAS SELECT
3   from TABLA;
```

El contenido de una vista se muestra con un "select":

```
1 select *from NOMBREVISTA;
```

En el siguiente ejemplo creamos la vista "vista_empleados", que es resultado de una combinación en la cual se muestran 4 campos:

```
1 create view vista_empleados as
2  select (apellido||' '||e.nombre) as nombre,sexo,
3   s.nombre as seccion, cantidadhijos
4   from empleados as e
```

```
  join secciones as s
  on codigo=seccion
```

Para ver la información contenida en la vista creada anteriormente tipeamos:

```
select *from vista_empleados;
```

Podemos realizar consultas a una vista como si se tratara de una tabla:

```
select seccion,count(*) as cantidad
 from vista_empleados;
```

Los nombres para vistas deben seguir las mismas reglas que cualquier identificador. Para distinguir una tabla de una vista podemos fijar una convención para darle nombres, por ejemplo, colocar el sufijo “vista” y luego el nombre de las tablas consultadas en ellas.

Los campos y expresiones de la consulta que define una vista DEBEN tener un nombre. Se debe colocar nombre de campo cuando es un campo calculado o si hay 2 campos con el mismo nombre. Note que en el ejemplo, al concatenar los campos .ªpellidoz "nombreçolocamos un alias; si no lo hubiésemos hecho aparecería un mensaje de error porque dicha expresión DEBE tener un encabezado, PostgreSQL no lo coloca por defecto.

Los nombres de los campos y expresiones de la consulta que define una vista DEBEN ser únicos (no puede haber dos campos o encabezados con igual nombre). Note que en la vista definida en el ejemplo, al campo "s.nombre"le colocamos un alias porque ya había un encabezado (el alias de la concatenación) llamado "nombrez no pueden repetirse, si sucediera, aparecería un mensaje de error.

Otra sintaxis es la siguiente:

```
create view NOMBREVISTA (NOMBRESDEENCABEZADOS)
 as
 SENTENCIASSELECT
  from TABLA;
```

Creamos otra vista de .empleados"denominada "vista_empleados_ingreso"que almacena la cantidad de empleados por año:

```
create view vista_empleados_ingreso (fecha,cantidad)
 as
 select extract(year from fechaingreso),count(*)
  from empleados
  group by extract(year from fechaingreso);
```

La diferencia es que se colocan entre paréntesis los encabezados de las columnas que aparecerán en la vista. Si no los colocamos y empleamos la sintaxis vista anteriormente, se emplean los nombres de los campos o alias (que en este caso habría que agregar) colocados en el "select"que define la vista. Los nombres que se colocan entre paréntesis deben ser tantos como los campos o expresiones que se definen en la vista.

Las vistas se crean en la base de datos activa.

Al crear una vista, PostgreSQL verifica que existan las tablas a las que se hacen referencia en ella.

Se aconseja probar la sentencia "selectçon la cual definiremos la vista antes de crearla para asegurarnos que el resultado que retorna es el imaginado.

Se pueden construir vistas sobre otras vistas.

Ingresemos el siguiente lote de comandos SQL en pgAdmin: – Borramos las tablas secciones y empleados si existen – y si hay objetos dependientes como las vistas también las elimina

```
drop table if exists secciones cascade;
drop table if exists empleados cascade;
```

```
create table secciones(
 codigo serial,
 nombre varchar(20),
 sueldo decimal(5,2),
 primary key (codigo)
);
```

```
create table empleados(
 legajo serial,
 documento char(8),
 sexo char(1),
 apellido varchar(20),
 nombre varchar(20),
 domicilio varchar(30),
 seccion smallint not null,
 cantidadhijos smallint,
 estadocivil char(10),
 fechaingreso date,
  primary key (legajo)
);
```

```
insert into secciones(nombre,sueldo) values('Administracion',300);
insert into secciones(nombre,sueldo) values('Contaduría',400);
insert into secciones(nombre,sueldo) values('Sistemas',500);

insert into empleados
(documento,sexo,apellido,nombre,domicilio,seccion,cantidadhijos,
   estadocivil,fechaingreso)
 values('22222222','f','Lopez','Ana','Colon 123',1,2,'casado','
    1990-10-10');
insert into empleados
(documento,sexo,apellido,nombre,domicilio,seccion,cantidadhijos,
   estadocivil,fechaingreso)
 values('23333333','m','Lopez','Luis','Sucre 235',1,0,'soltero','
    1990-02-10');
insert into empleados
(documento,sexo,apellido,nombre,domicilio,seccion,cantidadhijos,
   estadocivil,fechaingreso)
values('24444444','m','Garcia','Marcos','Sarmiento 1234',2,3,'
   divorciado','1998-07-12');
insert into empleados
(documento,sexo,apellido,nombre,domicilio,seccion,cantidadhijos,
   estadocivil,fechaingreso)
values('25555555','m','Gomez','Pablo','Bulnes 321',3,2,'casado','
   1998-10-09');
insert into empleados
(documento,sexo,apellido,nombre,domicilio,seccion,cantidadhijos,
   estadocivil,fechaingreso)
values('26666666','f','Perez','Laura','Peru 1254',3,3,'casado','
   2000-05-09');
```

– Creamos la vista "vista_empleados", que es resultado de una combinación en la cual – se muestran 5 campos:

```
create view vista_empleados as
 select (apellido||' '||e.nombre) as nombre,sexo,
  s.nombre as seccion, cantidadhijos
  from empleados as e
  join secciones as s
  on codigo=seccion;
```

– Vemos la información de la vista:

```
select * from vista_empleados;
```

– Realizamos una consulta a la vista como si se tratara de una tabla:

```
select seccion,count(*) as cantidad
 from vista_empleados
 group by seccion;
```

– Creamos otra vista de .empleados"denominada "vista_empleados_ingreso- que almacena la cantidad de empleados por año:

```
create view vista_empleados_ingreso (fecha,cantidad)
 as
 select extract(year from fechaingreso),count(*)
  from empleados
  group by extract(year from fechaingreso);
```

– Vemos la información:

```
select * from vista_empleados_ingreso;
```

La ejecución de este lote de comandos SQL genera una salida similar a:
PostgreSQL pgAdmin vistas
Las vistas y el sistema de reglas. Implementación de las vistas en Postgres Las vistas en Postgres se implementan utilizando el sistema de reglas. De hecho, no hay diferencia entre

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

y la secuencia:

```
CREATE TABLE myview
```

(la misma lista de atributos de mytab);

```
CREATE RULE "_RETmyview" AS ON SELECT TO myview DO INSTEAD
SELECT * FROM mytab;
```

Porque esto es exactamente lo que hace internamente el comando CREATE VIEW. Esto tiene algunos efectos colaterales. Uno de ellos es que la información sobre una vista en el sistema de catálogos de Postgres es exactamente el mismo que para una tabla. De este modo, para los traductores de queries, no hay diferencia entre una tabla y una vista, son lo mismo: relaciones. Esto es lo más importante por ahora. Cómo trabajan las reglas de SELECT.

Las reglas ON SELECT se aplican a todas las queries como el último paso, incluso si el comando dado es INSERT, UPDATE o DELETE. Y tienen diferentes semanticas de las otras en las que modifican el arbol de traducción en lugar de crear uno nuevo. Por ello, las reglas SELECT se describen las primeras.

Actualmente, debe haber sólo una acción y debe ser una acción SELECT que es una INSTEAD. Esta restricción se requería para hacer las reglas seguras contra la apertura por usuarios ordinarios, y restringe las reglas ON SELECT a reglas para vistas reales.

El ejemplo para este documento son dos vistas unidas que hacen algunos cálculos y algunas otras vistas utilizadas para ello. Una de estas dos primeras vistas se personaliza más tarde añadiendo reglas para operaciones de INSERT, UPDATE y DELETE de modo que el resultado final será una vista que se comporta como una tabla real con algunas funcionalidades mágicas. No es un ejemplo fácil para empezar, y quizá sea demasiado duro. Pero es mejor tener un ejemplo que cubra todos los puntos discutidos paso a paso que tener muchos ejemplos diferentes que tener que mezclar después.

La base de datos necesitada para ejecutar los ejemplos se llama al_bundy. Verá pronto el porqué de este nombre. Y necesita tener instalado el lenguaje procedural PL/pgSQL, ya que necesitaremos una pequeña función min() que devuelva el menor de dos valores enteros. Creamos esta función como:

```
1 CREATE FUNCTION min(integer, integer) RETURNS integer AS
2 'BEGIN
3 IF $1 < $2 THEN
4 RETURN $1;
5 END IF;
6 RETURN $2;
7 END;'
8 LANGUAGE 'plpgsql';
```

Las tablas reales que necesitaremos en las dos primeras descripciones del sistema de reglas son estas:

```
1 CREATE TABLE shoe_data (       -- datos de zapatos
2 shoename   char(10),       -- clave primaria (primary key)
3 sh_avail   integer,        -- número de pares utilizables
4 slcolor    char(10),       -- color de cordón preferido
5 slminlen   float,          -- longitud mínima de cordón
6 slmaxlen   float,          -- longitud máxima del cordón
7 slunit     char(8)         -- unidad de longitud
8 );
```

```
1 CREATE TABLE shoelace_data (  -- datos de cordones de zapatos
2 sl_name    char(10),       -- clave primaria (primary key)
3 sl_avail   integer,        -- número de pares utilizables
4 sl_color   char(10),       -- color del cordón
5 sl_len     float,          -- longitud del cordón
6 sl_unit    char(8)         -- unidad de longitud
7 );
```

```
1 CREATE TABLE unit (            -- unidades de longitud
2 un_name    char(8),        -- clave primaria (primary key)
3 un_fact    float           -- factor de transformación a cm
4 );
```

Pienso que la mayoría de nosotros lleva zapatos, y puede entender que este es un ejemplo de datos realmente utilizables. Bien es cierto que hay zapatos en el mundo que no necesitan cordones, pero nos hará más facil la vida ignorarlos.

Las vistas las crearemos como:

```
1 CREATE VIEW shoe AS
```

```
SELECT sh.shoename,
sh.sh_avail,
sh.slcolor,
sh.slminlen,
sh.slminlen * un.un_fact AS slminlen_cm,
sh.slmaxlen,
sh.slmaxlen * un.un_fact AS slmaxlen_cm,
sh.slunit
FROM shoe_data sh, unit un
WHERE sh.slunit = un.un_name;
```

```
CREATE VIEW shoelace AS
SELECT s.sl_name,
s.sl_avail,
s.sl_color,
s.sl_len,
s.sl_unit,
s.sl_len * u.un_fact AS sl_len_cm
FROM shoelace_data s, unit u
WHERE s.sl_unit = u.un_name;
```

```
CREATE VIEW shoe_ready AS
SELECT rsh.shoename,
rsh.sh_avail,
rsl.sl_name,
rsl.sl_avail,
min(rsh.sh_avail, rsl.sl_avail) AS total_avail
FROM shoe rsh, shoelace rsl
WHERE rsl.sl_color = rsh.slcolor
AND rsl.sl_len_cm >= rsh.slminlen_cm
AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

El comando CREATE VIEW para la vista shoelace (que es la más simple que tenemos) creará una relación shoelace y una entrada en pg_rewrite que dice que hay una regla de reescritura que debe ser aplicada siempre que la relación shoelace sea referida en la tabla de rango de una query. La regla no tiene cualificación de regla (discutidas en las reglas no SELECT, puesto que las reglas SELECT no pueden tenerlas) y es de tipo INSTEAD (en vez de). ¡Nótese que la cualificación de las reglas no son lo mismo que las cualificación de las queries! La acción de las reglas tiene una cualificación.

La acción de las reglas es un árbol de query que es una copia exacta de la instrucción SELECT en el comando de creación de la vista.

Nota Nota:

Las dos tablas de rango extra para NEW y OLD (llamadas *NEW* y *CURRENT* por razones históricas en el árbol de query escrito) que se pueden ver en la entrada pg_rewrite no son de interes para las reglas de SELECT.

Ahora publicamos unit, shoe_data y shoelace_data y Al (el propietario de al_bundy) teclea su primera SELECT en esta vida.

```
al_bundy=> INSERT INTO unit VALUES ('cm', 1.0);
al_bundy=> INSERT INTO unit VALUES ('m', 100.0);
al_bundy=> INSERT INTO unit VALUES ('inch', 2.54);
al_bundy=>
```

```
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh1', 2, 'black', 70.0, 90.0, 'cm');
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh2', 0, 'black', 30.0, 40.0, 'inch');
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh4', 3, 'brown', 40.0, 50.0, 'inch');
al_bundy=>
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl1', 5, 'black', 80.0, 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl2', 6, 'black', 100.0, 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl3', 0, 'black', 35.0 , 'inch');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl4', 8, 'black', 40.0 , 'inch');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl5', 4, 'brown', 1.0 , 'm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl6', 0, 'brown', 0.9 , 'm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl7', 7, 'brown', 60 , 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl8', 1, 'brown', 40 , 'inch');
al_bundy=>
al_bundy=> SELECT * FROM shoelace;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       7|brown     |    60|cm      |       60
sl3       |       0|black     |    35|inch    |     88.9
sl4       |       8|black     |    40|inch    |    101.6
sl8       |       1|brown     |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
sl6       |       0|brown     |   0.9|m       |       90
(8 rows)
```

Esta es la SELECT más sencilla que Al puede hacer en sus vistas, de modo que nosotros la tomaremos para explicar la base de las reglas de las vistas. 'SELECT * FROM shoelace'fue interpretado por el traductor y produjo un árbol de traducción.

```
SELECT shoelace.sl_name, shoelace.sl_avail,
shoelace.sl_color, shoelace.sl_len,
shoelace.sl_unit, shoelace.sl_len_cm
FROM shoelace shoelace;
```

y este se le dá al sistema de reglas. El sistema de reglas viaja a través de la tabla de rango, y comprueba si hay reglas en pg_rewrite para alguna relación. Cuando se procesa las entradas en la tabla de rango para shoelace (el único hasta ahora) encuentra la regla '_RETshoelace'con el árbol de traducción

```
SELECT s.sl_name, s.sl_avail,
```

```
2  s.sl_color, s.sl_len, s.sl_unit,
3  float8mul(s.sl_len, u.un_fact) AS sl_len_cm
4  FROM shoelace *OLD*, shoelace *NEW*,
5  shoelace_data s, unit u
6  WHERE bpchareq(s.sl_unit, u.un_name);
```

Nótese que el traductor cambió el calculo y la cualificación en llamadas a las funciones apropiadas. Pero de hecho esto no cambia nada. El primer paso en la reescritura es mezclar las dos tablas de rango. El árbol de traducción entonces lee:

```
1  SELECT shoelace.sl_name, shoelace.sl_avail,
2  shoelace.sl_color, shoelace.sl_len,
3  shoelace.sl_unit, shoelace.sl_len_cm
4  FROM shoelace shoelace, shoelace *OLD*,
5  shoelace *NEW*,
6  shoelace_data s,
7  unit u;
```

En el paso 2, añade la cualificación de la acción de las reglas al árbol de traducción resultante en

```
1  SELECT shoelace.sl_name, shoelace.sl_avail,
2  shoelace.sl_color, shoelace.sl_len,
3  shoelace.sl_unit, shoelace.sl_len_cm
4  FROM shoelace shoelace, shoelace *OLD*,
5  shoelace *NEW*, shoelace_data s,
6  unit u
7  WHERE bpchareq(s.sl_unit, u.un_name);
```

Y en el paso 3, reemplaza todas las variables en el arbol de traducción, que se refieren a entradas de la tabla de rango (la única que se está procesando en este momento para shoelace) por las correspondientes expresiones de la lista objetivo correspondiente a la acción de las reglas. El resultado es la query final:

```
1  SELECT s.sl_name, s.sl_avail,
2  s.sl_color, s.sl_len,
3  s.sl_unit,
4  float8mul(s.sl_len, u.un_fact) AS sl_len_cm
5  FROM shoelace shoelace, shoelace *OLD*,
6  shoelace *NEW*, shoelace_data s,
7  unit u
8  WHERE bpchareq(s.sl_unit, u.un_name);
```

Para realizar esta salida en una instrucción SQL real, un usuario humano debería teclear:

```
1  SELECT s.sl_name, s.sl_avail,
2  s.sl_color, s.sl_len,
3  s.sl_unit, s.sl_len * u.un_fact AS sl_len_cm
4  FROM shoelace_data s, unit u
5  WHERE s.sl_unit = u.un_name;
```

Esta ha sido la primera regla aplicada. Mientras se iba haciendo esto, la tabla de rango iba creciendo. De modo que el sistema de reglas continúa comprobando las entradas de la tabla de rango. Lo siguiente es el el número 2 (shoelace *OLD*). La Relación shoelace tiene una regla, pero su entrada en la tabla de rangos no está referenciada en ninguna de las variables del árbol de traducción, de modo que se ingnora. Puesto que todas las entradas restantes en la tabla de rango, o bien no tienen

reglas en pg_rewrite o bien no han sido referenciadas, se alcanza el final de la tabla de rango. La reescritura está completa y el resultado final dado se pasa al optimizador. El optimizador ignora las entradas extra en la tabla de rango que no están referenciadas por variables en el árbol de traducción, y el plan producido por el planificador/optimizador debería ser exactamente el mismo que si Al hubiese tecleado la SELECT anterior en lugar de la selección de la vista.

Ahora enfrentamos a Al al problema de que los Blues Brothers aparecen en su tienda y quieren comprarse zapatos nuevos, y como son los Blues Brothers, quieren llevar los mismos zapatos. Y los quieren llevar inmediatamente, de modo que necesitan también cordones.

Al necesita conocer los zapatos para los que tiene en el almacén cordones en este momento (en color y en tamaño), y además para los que tenga un número igual o superior a 2. Nosotros le enseñamos a realizar la consulta a su base de datos:

```
al_bundy=> SELECT * FROM shoe_ready WHERE total_avail >= 2;
shoename  |sh_avail|sl_name   |sl_avail|total_avail
----------+--------+----------+--------+-----------
sh1       |       2|sl1       |       5|          2
sh3       |       4|sl7       |       7|          4
(2 rows)
```

Al es un guru de los zapatos, y sabe que sólo los zapatos de tipo sh1 le sirven (los cordones sl7 son marrones, y los zapatos que necesitan cordones marrones no son los más adecuados para los Blues Brothers). La salida del traductor es esta vez el arbol de traducción.

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
shoe_ready.sl_name, shoe_ready.sl_avail,
shoe_ready.total_avail
FROM shoe_ready shoe_ready
WHERE int4ge(shoe_ready.total_avail, 2);
```

Esa será la primera regla aplicada para la relación shoe_ready y da como resultado el árbol de traducción

```
SELECT rsh.shoename,
rsh.sh_avail,
rsl.sl_name,
rsl.sl_avail,
min(rsh.sh_avail, rsl.sl_avail) AS
total_avail
FROM shoe_ready shoe_ready, shoe_ready *OLD*,
shoe_ready *NEW*,
shoe rsh,
shoelace rsl
WHERE int4ge(min(rsh.sh_avail, rsl.sl_avail), 2)
AND (bpchareq(rsl.sl_color, rsh.slcolor)
AND float8ge(rsl.sl_len_cm, rsh.slminlen_cm)
AND float8le(rsl.sl_len_cm, rsh.slmaxlen_cm)
);
```

En realidad, la clausula AND en la cualificación será un nodo de operadores de tipo AND, con una expresión a la izquierda y otra a la derecha. Pero eso la hace menos legible de lo que ya es, y hay más reglas para aplicar. De modo que sólo las mostramos entre paréntesis para agruparlos en unidades lógicas en el orden en que se añaden, y continuamos con las reglas para la relación shoe como está en la entrada de la tabla de rango a la que se refiere, y tiene una regla. El resultado de aplicarlo es

```
SELECT sh.shoename,
sh.sh_avail,
rsl.sl_name, rsl.sl_avail,
min(sh.sh_avail, rsl.sl_avail)
AS total_avail,
FROM shoe_ready shoe_ready, shoe_ready *OLD*,
shoe_ready *NEW*, shoe rsh,
shoelace rsl, shoe *OLD*,
shoe *NEW*,
shoe_data sh,
unit un
WHERE (int4ge(min(sh.sh_avail, rsl.sl_avail), 2)
AND (bpchareq(rsl.sl_color, sh.slcolor)
AND float8ge(rsl.sl_len_cm,
float8mul(sh.slminlen, un.un_fact))
AND float8le(rsl.sl_len_cm,
float8mul(sh.slmaxlen, un.un_fact))
)
)
AND bpchareq(sh.slunit, un.un_name);
```

Y finalmente aplicamos la regla para shoelace que ya conocemos bien (esta vez en un arbol de traducción que es un poco más complicado) y obtenemos

```
SELECT sh.shoename, sh.sh_avail,
s.sl_name, s.sl_avail,
min(sh.sh_avail, s.sl_avail) AS total_avail
FROM shoe_ready shoe_ready, shoe_ready *OLD*,
shoe_ready *NEW*, shoe rsh,
shoelace rsl, shoe *OLD*,
shoe *NEW*, shoe_data sh,
unit un, shoelace *OLD*,
shoelace *NEW*,
shoelace_data s,
unit u
WHERE (  (int4ge(min(sh.sh_avail, s.sl_avail), 2)
AND (bpchareq(s.sl_color, sh.slcolor)
AND float8ge(float8mul(s.sl_len, u.un_fact),
float8mul(sh.slminlen, un.un_fact))
AND float8le(float8mul(s.sl_len, u.un_fact),
float8mul(sh.slmaxlen, un.un_fact))
)
)
AND bpchareq(sh.slunit, un.un_name)
)
AND bpchareq(s.sl_unit, u.un_name);
```

Lo reducimos otra vez a una instrucción SQL real que sea equivalente en la salida final del sistema de reglas:

```
SELECT sh.shoename, sh.sh_avail,
s.sl_name, s.sl_avail,
min(sh.sh_avail, s.sl_avail) AS total_avail
FROM shoe_data sh, shoelace_data s, unit u, unit un
```

```
5  | WHERE min(sh.sh_avail, s.sl_avail) >= 2
6  | AND s.sl_color = sh.slcolor
7  | AND s.sl_len * u.un_fact >= sh.slminlen * un.un_fact
8  | AND s.sl_len * u.un_fact <= sh.slmaxlen * un.un_fact
9  | AND sh.sl_unit = un.un_name
10 | AND s.sl_unit = u.un_name;
```

El procesado recursivo del sistema de reglas reescribió una SELECT de una vista en un árbol de traducción que es equivalente a exactamente lo que Al hubiese tecleado de no tener vistas.

Nota Nota

Actualmente no hay mecanismos de parar la recursión para las reglas de las vistas en el sistema de reglas (sólo para las otras reglas). Esto no es muy grave, ya que la única forma de meterlo en un bucle sin fin (bloqueando al cliente hasta que lea el limite de memoria) es crear tablas y luego crearles reglas a mano con CREATE RULE de forma que una lea a la otra y la otra a la una. Esto no puede ocurrir con el comando CREATE VIEW, porque en la primera creación de una vista la segunda aún no existe, de modo que la primera vista no puede seleccionar desde la segunda.

Reglas de vistas en instrucciones diferentes a SELECT.

Dos detalles del arbol de traducción no se han tocado en la descripción de las reglas de vistas hasta ahora. Estos son el tipo de comando (commandtype) y la relación resultado (resultrelation). De hecho, las reglas de vistas no necesitan estas informaciones.

Hay sólo unas pocas diferencias entre un árbol de traducción para una SELECT y uno para cualquier otro comando. Obviamente, tienen otros tipos de comandos, y esta vez la relación resultado apunta a la entrada de la tabla de rango donde irá el resultado. Cualquier otra cosa es absolutamente igual. Por ello, teniendo dos tablas t1 y t2, con atributos a y b, los árboles de traducción para las dos instrucciones:

```
1 | SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

```
1 | UPDATE t1 SET b = t2.b WHERE t1.a = t2.a;
```

son prácticamente idénticos.

Las tablas de rango contienen entradas para las tablas t1 y t2.

Las listas objetivo continen una variable que apunta al atributo b de la entrada de la tabla rango para la tabla t2.

Las expresiones de cualificación comparan los atributos a de ambos rangos para la igualdad.

La consecuencia es que ambos árboles de traducción dan lugar a planes de ejecución similares. En ambas hay joins entre las dos tablas. Para la UPDATE, las columnas que no aparecen de la tabla t1 son añadidas a la lista objetivo por el optimizador, y el árbol de traducción final se lee como:

```
1 | UPDATE t1 SET a = t1.a, b = t2.b WHERE t1.a = t2.a;
```

Y por ello el ejecutor al correr sobre la join producirá exactamente el mismo juego de resultados que

```
1 | SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

Pero hay un pequeño problema con el UPDATE. El ejecutor no cuidará de que el resultado de la join sea coherente. El sólo produce un juego resultante de filas. La diferencia entre un comando SELECT y un comando UPDATE la manipula el llamador (caller) del ejecutor. El llamador sólo conoce (mirando en el árbol de traducción) que esto es una UPDATE, y sabe que su resultado deberá ir a la tabla t1. Pero ¿cuál de las 666 filas que hay debe ser reemplazada por la nueva fila? El plan

ejecutado es una join con una cualificación que potencialmente podría producir cualquier número de filas entre 0 y 666 en un número desconocido.

Para resolver este problema, se añade otra entrada a la lista objetivo en las instrucciones UPDATE y DELETE. Es el identificador de tupla actual (current tuple id, ctid). Este es un atributo de sistema con características especiales. Contiene el bloque y posición en el bloque para cada fila. Conociendo la tabla, el ctid puede utilizarse para encontrar una fila específica en una tabla de 1.5 GB que contiene millones de filas atacando un único bloque de datos. Tras la adición del ctid a la lista objetivo, el juego de resultados final se podría definir como:

```
1 | SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

Entra ahora en funcionamiento otro detalle de >Postgres. Las filas de la tabla no son reescritas en este momento, y el por ello por lo que ABORT TRANSACTION es muy rápido. En una Update, la nueva fila resultante se inserta en la tabla (tras retirarle el ctid) y en la cabecera de la tupla de la fila cuyo ctid apuntaba a las entradas cmax y zmax, se fija el contador de comando actual y el identificador de transaccion actual (ctid). De este modo, la fila anterior se oculta tras el commit de la transacción, y el limpiador vacuum puede realmente eliminarla.

Conociendo todo eso, podemos simplemente aplicar las reglas de las vistas exactamente en la misma forma en cualquier comando. No hay diferencia.

0.70. El poder de las vistas

Todo lo anterior demuestra como el sistema de reglas incorpora las definiciones de las vistas en el árbol de traducción original. En el segundo ejemplo, una simple SELECT de una vista creó un árbol de traducción final que es una join de cuatro tablas (cada una se utiliza dos veces con diferente nombre).

0.71. Beneficios

Los beneficios de implementar las vistas con el sistema de reglas están en que el optimizados tiene toda la información sobre qué tablas tienen que ser revisadas, más las relaciones entre estas tablas, más las cualificaciones restrictivas a partir de la definición de las vistas, más las cualificaciones de la query original, todo en un único árbol de traducción. Y esta es también la situación cuando la query original es ya una join entre vistas. Ahora el optimizador debe decidir cuál es la mejor ruta para ejecutar la query. Cuanta más información tenga el optimizador, mejor será la decisión. Y la forma en que se implementa el sistema de reglas en Postgres asegura que toda la información sobre la query está utilizable.

0.72. Puntos delicados a considerar

Hubo un tiempo en el que el sistema de reglas de Postgres se consideraba agotado. El uso de reglas no se recomendaba, y el único lugar en el que trabajaban era las reglas de las vistas. E incluso estas reglas de las vistas daban problemas porque el sistema de reglas no era capaz de aplicarse adecuadamente en más instrucciones que en SELECT (por ejemplo, no trabajaría en una UPDATE que utilice datos de una vista).

Durante ese tiempo, el desarrollo se dirigió hacia muchas características añadidas al traductor y al

optimizador. El sistema de reglas fué quedando cada vez más desactualizado en sus capacidades, y se volvió cada vez más dificil de actualizar. Y por ello, nadie lo hizo.

En 6.4, alguien cerró la puerta, respiró hondo, y se puso manos a la obra. El resultado fué el sistema de reglas cuyas capacidades se han descrito en este documento. Sin embargo, hay todavía algunas construcciones no manejadas, y algunas fallan debido a cosas que no son soportadas por el optimizador de queries de Postgres.

Las vistas con columnas agregadas tienen malos problemas. Las expresiones agregadas en las cualificaciones deben utilizarse en subselects. Actualmente no es posible hacer una join de dos vistas en las que cada una de ellas tenga una columna agregada, y comparar los dos valores agregados en a cualificación. Mientras tanto, es posible colocar estas expresiones agregadas en funciones con los argumentos apropiados y utilizarlas en la definición de las vistas.

Las vistas de uniones no son soportadas. Ciertamente es sencillo reescribir una SELECT simple en una unión, pero es un poco más dificil si la vista es parte de una join que hace una UPDATE.

Las clausulas ORDER BY en las definiciones de las vistas no están soportadas.

DISTINCT no está soportada en las definiciones de vistas.

No hay una buena razon por la que el optimizador no debiera manipular construcciones de árboles de traducción que el traductor nunca podría producir debido a las limitaciones de la sintaxis de SQL. El autor se alegrará de que estas limitaciones desaparezcan en el futuro.

0.73. Efectos colaterales de la implementación

La utilización del sistema de reglas descrito para implementar las vistas tiene algunos efectos colaterales divertidos. Lo siguiente no parece trabajar:

```
al_bundy=> INSERT INTO shoe (shoename, sh_avail, slcolor)
al_bundy->     VALUES ('sh5', 0, 'black');
INSERT 20128 1
al_bundy=> SELECT shoename, sh_avail, slcolor FROM shoe_data;
shoename  |sh_avail|slcolor
----------+--------+----------
sh1       |       2|black
sh3       |       4|brown
sh2       |       0|black
sh4       |       3|brown
(4 rows)
```

Lo interesante es que el código de retorno para la INSERT nos dió una identificación de objeto, y nos dijo que se ha insertado una fila. Sin embargo no aparece en shoe_data. Mirando en el directorio de la base de datos, podemos ver que el fichero de la base de datos para la relación de la vista shoe parece tener ahora un bloque de datos. Y efectivamente es así.

Podemos también intentar una DELETE, y si no tiene una cualificación, nos dirá que las filas se han borrado y la siguiente ejecución de vacuum limpiará el fichero hasta tamaño cero.

La razon para este comportamiento es que el árbol de la traducción para la INSERT no hace referencia a la relación shoe en ninguna variable. La lista objetivo contiene sólo valores constantes. Por ello no hay reglas que aplicar y se mantiene sin cambiar hasta la ejecución, insertandose la fila. Del mismo modo para la DELETE.

Para cambiar esto, podemos definir reglas que modifiquen el comportamiento de las queries no-SELECT. Este es el tema de la siguiente sección.

Consultas tipo JOIN

El manejo del lenguaje SQL es un conocimiento fundamental para todo programador moderno, ya que es la piedra angular sobre la que construiremos, sea cual sea el framework, el acceso y persistencia de nuestros datos (entre otras muchas cosas).

Y uno de los "misterios" que más cuesta en aprender es el funcionamiento de la sentencia JOIN, con sus calificadores situados antes y después del verbo.

Hoy quiero traer una de las cientos de imágenes que pululan en internet que muestran de forma gráfica el funcionamiento de todos los tipos de JOIN. Pero, además, lo voy a poner en práctica en una pequeñita base de datos para visualizar los resultados.

Lo primero de todo es localizar la imagen que vamos a utilizar para mostrar visualmente, y de un vistazo, lo que realmente significa un LEFT JOIN a diferencia de un INNER JOIN. Para eso he buscado en internet y he escogido la que más me ha gustado, que está publicada en stackoverflow.

A continuación tengo montado en mi equipo un SQL Express 2014 y he dado de alta la siguiente base de datos con la tablas y campos que veis a continuación... el sumun de la complejidad.

Y le he introducido información falsa en el interior con la siguiente característica, falta un dato en la tabla fichas para un registro en la tabla personas. Es decir, hay una persona sin ficha.

Y ahora voy a lanzar los JOIN que aparecen en la representación visual y a ver qué resultados devuelve, empezando por la central y más comúnmente utilizada.

Ahora voy a continuar por la columna de la izquierda, pasando por las tres combinaciones principales.

Y continuo por la columna de la derecha, empezando por arriba hasta haber obtenido los resultados de las tres siguientes.

Espero que con esta imagen y las diferentes ejecuciones de las sentencias SQL, te quede más claro el funcionamiento de la sentencia JOIN que, como se puede observar, tiene un comportamiento basado en la teoría de los conjuntos.

0.74. Combinaciones internas - INNER JOIN

Las combinaciones internas se realizan mediante la instrucción INNER JOIN. Devuelven únicamente aquellos registros/filas que tienen valores idénticos en los dos campos que se comparan para unir ambas tablas. Es decir aquellas que tienen elementos en las dos tablas, identificados éstos por el campo de relación.

La mejor forma de verlo es con un diagrama de Venn que ilustre en qué parte de la relación deben existir registros: En este caso se devuelven los registros que tienen nexo de unión en ambas tablas. Por ejemplo, en la relación entre las tablas de clientes y pedidos en Northwind, se devolverán los registros de todos los clientes que tengan al menos un pedido, relacionándolos por el ID de cliente.

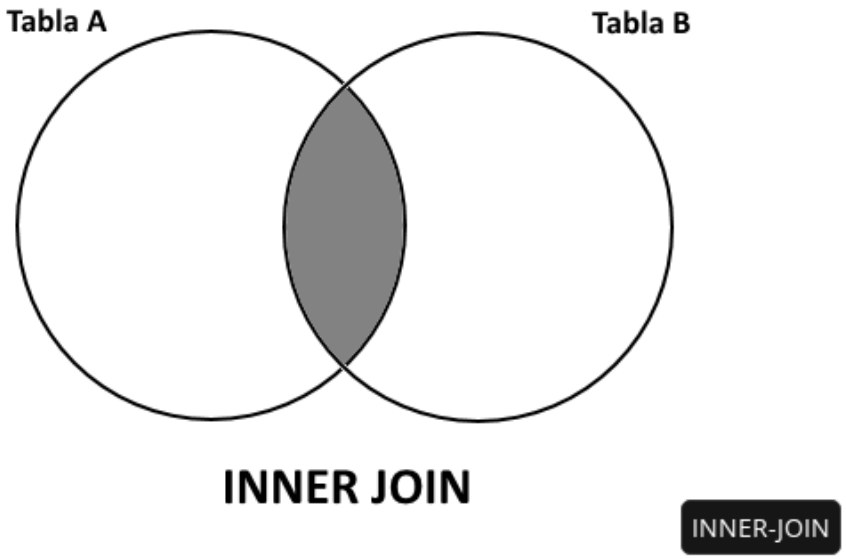


Figura 22:

Esto puede ocasionar la desaparición del resultado de filas de alguna de las dos tablas, por tener valores nulos, o por tener un valor que no exista en la otra tabla entre los campos/columnas que se están comparando.

Su sintaxis es:

```
FROM Tabla1 [INNER] JOIN Tabla2 ON Condiciones_Vinculos_Tablas
```

Así, para seleccionar los registros comunes entre la Tabla1 y la Tabla2 que tengan correspondencia entre ambas tablas por el campo Col1, escribiríamos:

```
SELECT T1.Col1, T1.Col2, T1.Col3, T2.Col7
FROM Tabla1 T1 INNER JOIN Tabla2 T2 ON T1.Col1 = T2.Col1
```

Por ejemplo, para obtener en Northwind los clientes que tengan algún pedido, bastaría con escribir:

```
SELECT OrderID, C.CustomerID, CompanyName, OrderDate
FROM Customers C INNER JOIN Orders O ON C.CustomerID = O.CustomerID
```

Que nos devolverá 830 registros. Hay dos pedidos en la tabla de Orders sin cliente asociado, como puedes comprobar, y éstos no se devuelven por no existir la relación entre ambas tablas.

En realidad esto ya lo conocíamos puesto que en las combinaciones internas, el uso de la palabra INNER es opcional (por eso lo hemos puesto entre corchetes). Si simplemente indicamos la palabra JOIN y la combinación de columnas (como ya hemos visto en el artículo anterior) el sistema sobreentiende que estamos haciendo una combinación interna. Lo hemos incluido por ampliar la explicación y por completitud.

0.75. Combinaciones externas (OUTER JOIN)

Las combinaciones externas se realizan mediante la instrucción OUTER JOIN. Como enseguida veremos, devuelven todos los valores de la tabla que hemos puesto a la derecha, los de la tabla que

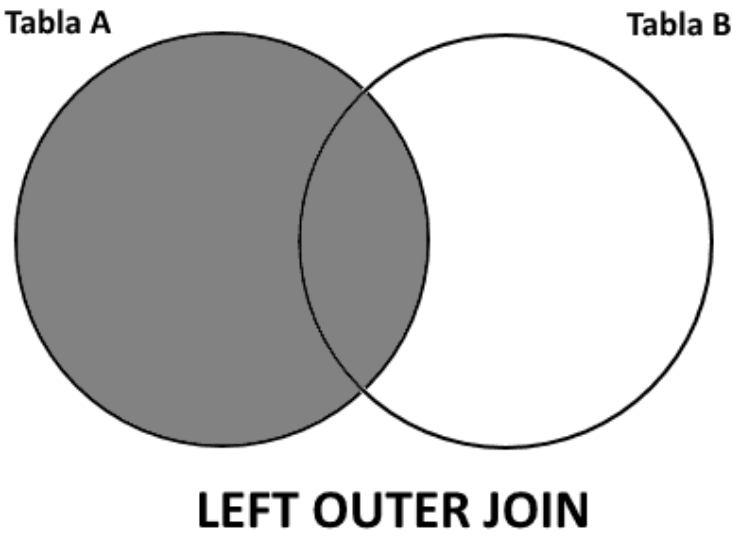


Figura 23:

hemos puesto a la izquierda o los de ambas tablas según el caso, devolviendo además valores nulos en las columnas de las tablas que no tengan el valor existente en la otra tabla.

Es decir, que nos permite seleccionar algunas filas de una tabla aunque éstas no tengan correspondencia con las filas de la otra tabla con la que se combina. Ahora lo veremos mejor en cada caso concreto, ilustrándolo con un diagrama para una mejor comprensión.

La sintaxis general de las combinaciones externas es:

```
FROM Tabla1 [LEFT/RIGHT/FULL] [OUTER] JOIN Tabla2 ON
   Condiciones_Vinculos_Tablas
```

Como vemos existen tres variantes de las combinaciones externas.

En todas estas combinaciones externas el uso de la palabra OUTER es opcional. Si utilizamos LEFT, RIGHT o FULL y la combinación de columnas, el sistema sobreentiende que estamos haciendo una combinación externa.

0.76. Variante LEFT JOIN

Se obtienen todas las filas de la tabla colocada a la izquierda, aunque no tengan correspondencia en la tabla de la derecha.

Así, para seleccionar todas las filas de la Tabla1, aunque no tengan correspondencia con las filas de la Tabla2, suponiendo que se combinan por la columna Col1 de ambas tablas escribiríamos:

```
SELECT T1.Col1, T1.Col2, T1.Col3, T2.Col7
FROM Tabla1 T1 LEFT [OUTER] JOIN Tabla2 T2 ON T1.Col1 = T2.Col1
```

Esto se ilustra gráficamente de la siguiente manera: De este modo, volviendo a Northwind, si escribimos la siguiente consulta:

```
SELECT OrderID, C.CustomerID, CompanyName, OrderDate
FROM Customers C LEFT JOIN Orders O ON C.CustomerID = O.CustomerID
```

Obtendremos 832 registros ya que se incluyen todos los clientes y sus pedidos, incluso aunque no tengan pedido alguno. Los que no tienen pedidos carecen de la relación apropiada entre las dos tablas a partir del campo CustomerID. Sin embargo se añaden al resultado final dejando la parte correspondiente a los datos de la tabla de pedidos con valores nulos, como se puede ver en esta captura de SQL Server:

0.77. Variante RIGHT JOIN

Análogamente, usando RIGHT JOIN se obtienen todas las filas de la tabla de la derecha, aunque no tengan correspondencia en la tabla de la izquierda.

Así, para seleccionar todas las filas de la Tabla2, aunque no tengan correspondencia con las filas de la Tabla1 podemos utilizar la cláusula RIGHT:

```
SELECT T1.Col1, T1.Col2, T1.Col3, T2.Col7
FROM Tabla1 T1 RIGHT [OUTER] JOIN Tabla2 T2 ON T1.Col1 = T2.Col1
```

El diagrama en este caso es complementario al anterior:

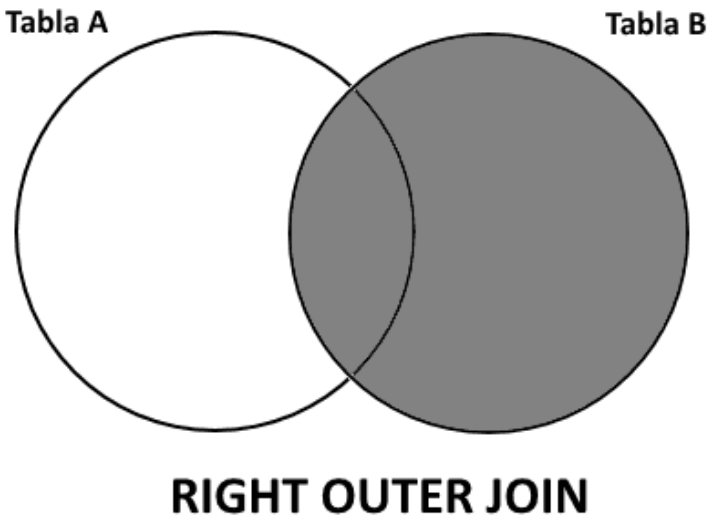


Figura 24:

Si en nuestra base de datos de ejemplo queremos obtener todos los pedidos aunque no tengan cliente asociado, junto a los datos de dichos clientes, escribiríamos:

```
SELECT OrderID, C.CustomerID, CompanyName, OrderDate
FROM Customers C RIGHT JOIN Orders O ON C.CustomerID = O.CustomerID
```

En este caso se devuelven 830 registros que son todos los pedidos. Si hubiese algún pedido con el CustomerID vacío (nulo) se devolvería también en esta consulta (es decir, órdenes sin clientes), aunque en la base de datos de ejemplo no se da el caso.

0.78. Variante FULL JOIN

Se obtienen todas las filas en ambas tablas, aunque no tengan correspondencia en la otra tabla. Es decir, todos los registros de A y de B aunque no haya correspondencia entre ellos, rellenando con nulos los campos que falten:

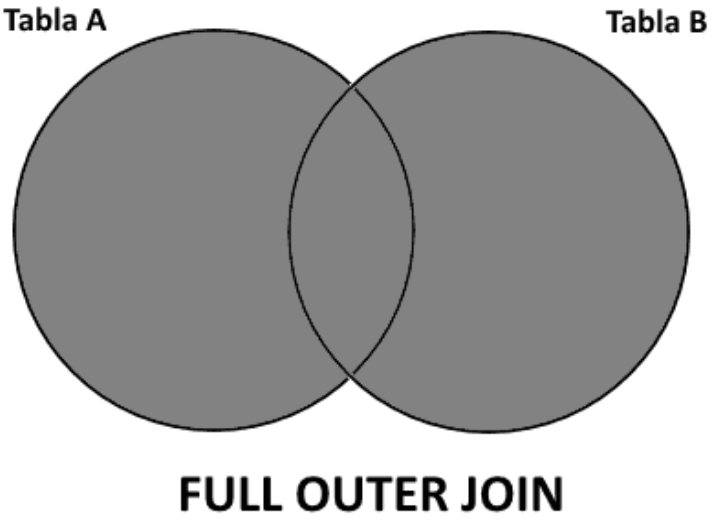


Figura 25:

Es equivalente a obtener los registros comunes (con un INNER) y luego añadirle los de la tabla A que no tienen correspondencia en la tabla B, con los campos de la tabla vacíos, y los registros de la tabla B que no tienen correspondencia en la tabla A, con los campos de la tabla A vacíos.

Su sintaxis es:

```
SELECT T1.Col1, T1.Col2, T1.Col3, T2.Col7
FROM Tabla1 T1 FULL [OUTER] JOIN Tabla2 T2 ON T1.Col1 = T2.Col1
```

Por ejemplo, en Northwind esta consulta:

```
SELECT OrderID, C.CustomerID, CompanyName, OrderDate
FROM Customers C FULL JOIN Orders O ON C.CustomerID = O.CustomerID
```

nos devuelve nuevamente 832 registros: los clientes y sus pedidos, los clientes sin pedido (hay 2) y los pedidos sin cliente (que en este caso son 0).

0.79. UNION

La sentencia SQL UNION es utilizada para acumular los resultados de dos sentencias SELECT.

Las dos sentencias SELECT tienen que tener el mismo número de columnas, con el mismo tipo de dato y en el mismo orden.

Sintaxis SQL UNION

```
SELECT columna1, columna2 FROM tabla1
UNION
SELECT columna1, columna2 FROM tabla2
```

Ejemplo SQL UNION

```
Tabla "personas_empresa1"

per nombre  apellido1 apellido2
1 ANTONIO PEREZ GOMEZ
2 ANTONIO GARCIA  RODRIGUEZ
3 PEDRO RUIZ  GONZALEZ
```

```
Tabla "personas_empresa2"

per nombre  apellido1 apellido2
1 JUAN  APARICIO  TENS
2 ANTONIO CHÁVEZ  RODRIGUEZ
3 LUIS  LOPEZ VAZQUEZ
```

```
SELECT nombre, apellido1 FROM personas_empresa1
UNION
SELECT nombre, apellido1 FROM personas_empresa2

nombre  apellido1
ANTONIO PEREZ
ANTONIO CHÁVEZ
PEDRO RUIZ
JUAN  APARICIO
LUIS  LOPEZ
```

La persona ÁNTONIO CHÁVEZ RODRIGUEZáparecerá sólo una vez en el resultado, porque no aparecerán las filas repetidas.

0.80. using

When joining tables with USING, the listed columns are merged and no longer belong to either the left or the right side. That means they can no longer be qualified which can often be an inconvenience.
SELECT a.x, b.y, z FROM a INNER JOIN b USING (z);
The SQL standard provides a workaround for this by allowing an alias on the join clause. (<join correlation name>in section 7.10)
SELECT j.x, j.y, j.z FROM a INNER JOIN b USING (z) AS j;

Transacciones

Una transacción es una operación de todo o nada, es decir, que se ejecuta todo el conjunto de operaciones a la vez o no debe ejecutarse ninguna. Esto es útil para mantener la integridad de los datos en la base. Por ejemplo, asumamos el caso en el que una persona transfiere dinero de una cuenta a otra, las operaciones a realizar serían las siguiente.

```
1 Sumar al monto de la cuenta A el monto transferido de la cuenta B.
2 Restar el monto de la cuenta B al monto inicial de la cuenta  B.
```

Estas operaciones deben realizarse las dos o no realizarse ninguna, ésto con el fin de mantener los saldos de las cuentas en valores correctos.

En postgreSQL se pueden realizar transacciones utilizando las siguientes instrucciones SQL: BEGIN, COMMIT,

0.81. ROLLBACK, ROLLBACK TO y SAVEPOINT

0.81.1. BEGIN

Se utiliza BEGIN para indicar explícitamente el inicio de una transacción. Todas las instrucciones siguientes a BEGIN deben completarse correctamente para que sean registradas permanentemente. En caso de que haya un error, la transacción completa habrá fallado y los cambios ya no serán aplicados. Es de hacer notar que postgreSQL utiliza un BEGIN implícito en cada instrucción en caso de que no se indique explícitamente.

0.81.2. COMMIT

Se utiliza COMMIT para registrar los cambios hechos en una transacción permanentemente. Después de ejecutar esta instrucción, los cambios realizados en la transacción son escritos permanentemente y se hacen visibles a otras transacciones concurrentes en la base. PostgreSQL utiliza un COMMIT implícito en cada instrucción en caso de que no se indique explícitamente. Siguiendo el ejemplo inicial, tendríamos: Loading ....

0.81.3. ROLLBACK

Se utilizar ROLLBACK para descartar todas las operaciones realizadas en una transacción. Cuando una transacción falla en alguna de las instrucciones, la única opción que tenemos es hacer un ROLLBACK. Al ejecutar esta instrucción, todos los cambios realizados son revertidos, como si nada

hubiera pasado. Asumamos, para nuestro ejemplo, que al cargar la cuenta A esta terminó con un saldo negativo (lo cual no es permitido):

0.81.4. SAVEPOINT

Se utiliza SAVEPOINT para definir puntos de control en una transacción. Algunas transacciones, por su complejidad, pueden tener una gran cantidad de instrucciones que en caso de que falle una, habría que volver a comenzar con la transacción. El uso de SAVEPOINT ayuda a dividir el bloque de instrucciones de la transacción en bloque más pequeños que queremos mantener en caso de que algo salga mal más adelante.

0.81.5. ROLLBACK TO

Se utiliza ROLLBACK TO para revertir los cambios hechos después de un SAVEPOINT. Después de definir un SAVEPOINT podemos volver a el utilizando la instrucción ROLLBACK TO <nombre_savepoint>. Todos los cambios realizados después del SAVEPOINT con nombre <nombre_savepoint>serán descartados (incluso savepoints posteriores) y nunca serán visibles en caso de ejecutar el COMMIT.

Agreguemos una tercera cuenta llamada C que también será cargada con un monto y este monto será abonado a la cuenta B. Utilizando SAVEPOINT y ROLLBACK TO.

Por supuesto que las transacciones pueden ser más complejas que este ejemplo pero esta el la forma en que se manejan las transacciones en PostgreSQL.

Una transacción tiene cuatro características esenciales conocidas como el acrónimo ACID:

- Atomicity(Atomicidad): Una transacción es una unidad atómica o se ejecutan las operaciones múltiples por completo o no se ejecuta absolutamente nada, cualquier cambio parcial es revertido para asegurar la consistencia en la base de datos.
- Consistency (Consistencia): Cuando finaliza una transacción debe dejar todos los datos sin ningún tipo de inconsistencia, por lo que todas las reglas de integridad deben ser aplicadas a todos los cambios realizados por la transacción, o sea todas las estructuras de datos internas deben de estar en un estado consistente.
- Isolation (Aislamiento o independencia): Esto significa que los cambios de cada transacción son independientes de los cambios de otras transacciones que se ejecuten en ese instante, o sea que los datos afectados de una transacción no están disponibles para otras transacciones sino hasta que la transacción que los ocupa finalice por completo.
- Durability (Permanencia): Después de que las transacciones hayan terminado, todos los cambios realizados son permanentes en la base de datos incluso si después hay una caída del DBMS.

Las transacciones en PostgreSQL utilizan las siguientes palabras reservadas:

Cuadro 20: Comandos básicos para transacciones.

Comando	Descripción
BEGIN:	Empieza la transacción
SAVEPOINT [name]:	Le dice al DBMS la localización de un punto de retorno en la transacción si una parte de la transacción es cancelada. El DBMS guarda el estado de la transacción hasta este punto.
COMMIT:	Todos los cambios realizados por las transacciones deben ser permanentes y accesibles a las demás operaciones del DBMS.
ROLLBACK [savepoint]:	Aborta la actual transacción todos los cambios realizados deben ser revertidos.

Para prácticar estos comandos utilizaremos dos tablas relacionadas Invoices y InvoiceDetails.

Agregamos un par de registros, cada uno dentro de una transacción en el primer registro el commit (confirmación) se realiza de forma automática al terminar la transacción con el comando END.

En el segundo registro utilizamos el comando COMMIT de forma explicita para hacer los cambios permanentes.

Ahora insertamos un nuevo registro y eliminamos un par pero en vez de confirmar la transacción con COMMIT deshacemos los cambios y regresamos los registros a su estado original, utilizando ROLLBACK.

Aquí otro ejemplo del uso de ROLLBACK.

Trabajaremos con la base de datos Northwind en nuestros ejemplos. Vamos a realizar una transacción que modifica el precio de dos productos de la base de datos.

```
1  USE NorthWind
2  DECLARE @Error int
3  --Declaramos una variable que utilizaremos para almacenar un posible có
       digo de error
4
5  BEGIN TRAN
6  --Iniciamos la transacción
7  UPDATE Products SET UnitPrice=20 WHERE ProductName ='Chai'
8  --Ejecutamos la primera sentencia
9  SET @Error=@@ERROR
10 --Si ocurre un error almacenamos su código en @Error
11 --y saltamos al trozo de código que deshara la transacción. Si, eso de
       ahí es un
12 --GOTO, el demonio de los programadores, pero no pasa nada por usarlo
13 --cuando es necesario
14 IF (@Error<>0) GOTO TratarError
15
16 --Si la primera sentencia se ejecuta con éxito, pasamos a la segunda
17 UPDATE Products SET UnitPrice=20 WHERE ProductName='Chang'
18 SET @Error=@@ERROR
19 --Y si hay un error hacemos como antes
20 IF (@Error<>0) GOTO TratarError
```

```
21
22 --Si llegamos hasta aquí es que los dos UPDATE se han completado con
23 --éxito y podemos "guardar" la transacción en la base de datos
24 COMMIT TRAN
25
26 TratarError:
27 --Si ha ocurrido algún error llegamos hasta aquí
28 If @@Error<>0 THEN
29 BEGIN
30 PRINT 'Ha ecorrido un error. Abortamos la transacción'
31 --Se lo comunicamos al usuario y deshacemos la transacción
32 --todo volverá a estar como si nada hubiera ocurrido
33 ROLLBACK TRAN
34 END
```

Como se puede ver para cada sentencia que se ejecuta miramos si se ha producido o no un error, y si detectamos un error ejecutamos el bloque de código que deshace la transacción.

Hay una interpretación incorrecta en cuanto al funcionamiento de las transacciones que esta bastante extendida. Mucha gente cree que si tenemos varias sentencias dentro de una transacción y una de ellas falla, la transacción se aborta en su totalidad.

¡Nada más lejos de la realidad! Si tenemos dos sentencias dentro de una transacción.

```
1 USE NorthWind
2 BEGIN TRAN
3 UPDATE Products SET UnitPrice=20 WHERE ProductName='Chang'
4 UPDATE Products SET UnitPrice=20 WHERE ProductName='Chang'
5 COMMIT TRAN
```

Estas dos sentencias se ejecutarán como una sola. Si por ejemplo en medio de la transacción (después del primer update y antes del segundo) hay un corte de electricidad, cuando el SQL Server se recupere se encontrará en medio de una transacción y, o bien la termina o bien la deshace, pero no se quedará a medias.

El error está en pensar que si la ejecución de la primera sentencia da un error se cancelará la transacción. El SQL Server sólo se preocupa de ejecutar las sentencias, no de averiguar si lo hacen correctamente o si la lógica de la transacción es correcta. Eso es cosa nuestra.

Por eso en el ejemplo que tenemos más arriba para cada sentencia de nuestro conjunto averiguamos si se ha producido un error y si es así actuamos en consecuencia cancelando toda la operación.

0.82. Caso

En el siguiente bloque de PL/SQL anónimo vamos a utilizar los comandos anteriores además del comando SAVEPOINT el cuál permite deshacer parcialmente los cambios hechos dentro de una transacción y no toda la transacción por completo.

Persistimos entonces sólo los cambios antes del SAVEPOINT, los cambios realizados después serán revertidos por el comando ROLLBACK.

En este capítulo hablaremos de uno de los conceptos más importantes a la hora de hablar de gestores de Bases de Datos; hablaremos de las transacciones en PostgreSQL.

Es fundamental conocer el concepto de transacción en los sistemas gestores de bases de datos, como PostgreSQL. Una transacción empaqueta varios pasos en una operación, de forma que se completen todos o ninguno. Los estados intermedios entre los pasos no son visible para otras transacciones ocurridas en el mismo momento.

En el caso de que ocurra algún fallo que impida que se complete la transacción, ninguno de los pasos se ejecutan y no afectan a los objetos de la base de datos.

Los pasos dentro de una transacción son varias sentencias SQL que deben de completarse todas para que queden registradas. Para comenzar una transacción utilizamos el comando BEGIN. Para indicar al sistema que han terminado correctamente todas las sentencias SQL, utilizamos el comando COMMIT. Hay ocasiones en las que tenemos que desechar algunos de los pasos que se están realizando. Para cancelar la transacción comenzada utilizamos el comando ROLLBACK.

Veamos estos comando con ejemplos.

0.83. Comando BEGIN

Cuando utilizamos este comando el sistema permite que se ejecuten todas las sentencias SQL que necesitemos y las registra en un fichero. A continuación os dejo un ejemplo donde se comienza una transacción donde deben de completarse satisfactoriamente todas las sentencias.

```
1 BEGIN;
2
3 UPDATE cuentas SET balance = balance - 100.00 WHERE n_cuenta = 0127365;
4
5 UPDATE cuentas SET balance = balance + 100.00 WHERE n_cuenta = 0795417;
```

0.84. Comando COMMIT

Cuando ejecutamos este comando estamos confirmando que todas las sentencias son correctas. Así pues, hasta que no se ejecute el comando COMMIT, las sentencias no quedarán registradas. Por ejemplo, si cerramos la conexión antes de ejecutar este comando no se verá afectada ninguna de las relaciones de la base de datos. A continuación os dejo un ejemplo en el cual se comienza una transacción, se ejecutan una serie de pasos y confirmamos que todas las sentencias están correctas.

```
1  BEGIN;
2
3  INSERT INTO cuentas (n_cuenta, nombre, balance) VALUES (0679259, 'Pepe'
       , 200);
4
5  UPDATE cuentas SET balance = balance - 137.00 WHERE nombre = 'Pepe';
6
7  UPDATE cuentas SET balance = balance + 137.00 WHERE nombre = 'Juan';
8
9  SELECT nombre, balance FROM cuentas WHERE nombre = 'Pepe' AND nombre =
       'Juan';
10
11 COMMIT;
```

0.85. Comando ROLLBACK

Con este comando podemos desechar las transacciones que se hayan ejecutado. Por lo tanto, después de haber realizado y confirmado una transacción, PostgreSQL nos permite anular dicha transacción de forma que no se modifique los datos de nuestra base de datos. A continuación os dejo un ejemplo donde vamos a anular la transacción confirmada anteriormente.

```
1 BEGIN;
2 "SENTENCIAS SQL"
3 COMMIT;
4
5 ROLLBACK;
```

Para utilizar estos comando mencionados (BEGIN, COMMIT y ROLLBACK) debemos de desactivar el AUTOCOMMIT. Ésta opción es a nivel de cliente y por defecto está activada. De forma que toda sentencia ejecutada queda confirmada y registrada en la base de datos.

0.86. Casos

Las transacciones son un concepto fundamental en todos los sistemas de base de datos, el punto esencial de una transaccion es que engloba multiples operaciones en un solo paso. Por ejemplo, considere la base de datos de un banco que contiene balances para varias cuentas de clientes, supongamos que queremos registrar el pago de Bs.S.100 desde la cuenta de Alice hacia la cuenta de Bob, las sentencias SQL a ejecutar para esta operacion serian como la siguiente:

```
1 UPDATE cuentas SET balance = balance - 100 WHERE nombre = 'Alice';
2 UPDATE branches SET balance = balance - 100 WHERE nombre = (SELECT
     branch_name FROM cuentas WHERE nombre = 'Alice');
3 UPDATE cuentas SET balance = balance + 100 WHERE nombre = 'Bob';
4 UPDATE branches SET balance = balance + 100 WHERE nombre = (SELECT
     branch_name FROM cuentas WHERE nombre = 'Bob');
```

Como se puede observar hay varias actualizaciones involucradas para terminar la operacion, los operadores del bando deben estar seguros de que todas esas actualizaciones se ejecuten, o en caso de falla que ninguna se ejecute, ya que se podria dar el caso de que Bob reciba Bs.S.100 sin que sean debitados de la cuenta de Alice, Agrupando las actualizaciones en una sola transaccion se puede garantizar que en caso de un fallo ninguna actualizacion se ejecute.

En PostgreSQL las transacciones se configuran simplemente encerrando en un bloque las operaciones que se desean incluir en la misma, el bloque debe comenzar y terminar con los comandos BEGIN y COMMIT, por ejemplo:

```
1 BEGIN;
2 UPDATE cuentas SET balance = balance - 100 WHERE nombre = 'Alice';
3 \dots
4 COMMIT;
```

Al momento que se le pasa a postgresql la clausula COMMIT es cuando se escribe en base de datos las actualizacione u operacion que se desea hacer en la misma, si en algun momento no queremos hacer COMMIT de alguna operacion (quizas se nota que la cuenta de Alice da un balance negativo) entonces se puede utilizar la clausula ROLLBACK y todas las actualizaciones dentro de la transaccion se cancelaran.

Si no se desea hacer un rollback completo de la transaccion, entonces se pueden definir marcadores (savepoints) hasta los cuales se desea que se regrese en la transaccion, ejemplo:

```
1 BEGIN;
2 UPDATE cuentas SET balance = balance - 100 WHERE nombre = 'Alice';
3 SAVEPOINT marcador1;
4 UPDATE cuentas SET balance = balance + 100 WHERE nombre = 'Bob';
5 -- se desea descartar la actualizacion para Bob y en vez hacerla para
     Wally --
6 ROLLBACK TO marcador1;
7 UPDATE cuentas SET balance = balance + 100 WHERE nombre = 'Wally';
8 COMMIT;
```

En el ejemplo anterior se vio el uso de Marcadores y Rollbacks, en este caso lo que paso es que se realizo una actualizacion sobre la cuenta de Bob, pero a ultima instacia se decide que el dinero no se le va a abonar a Bob sino a Wally, entonces se realiza un rollback hasta el marcador llamado marcador1 y se pasa a hacer la actualizacion en la cuenta de Wally.

0.87. Transacciones dentro de una transacción

¿Qué comportamiento mostraría PostgreSQL si, por ejemplo, se llamara el script a continuación?

```
1 BEGIN;
2 SELECT * FROM foo;
3 INSERT INTO foo(name) VALUES ('bar');
4 BEGIN; <- The point of interest
5 END;
```

¿PostgreSQL descartaría el segundo BEGIN o se decidiría implícitamente una confirmación y luego ejecutaría el bloque BEGIN END al final como una transacción separada?

Lo que necesitaría es una llamada "transacción autónoma" (una característica proporcionada por Oracle). En este punto, esto aún no es posible en PostgreSQL.

Sin embargo, puede usar SAVEPOINT:

```
1 BEGIN;
2 INSERT ...
3 SAVEPOINT a;
4 some error;
5 ROLLBACK TO SAVEPOINT a;
6 COMMIT;
```

No es del todo una transacción autónoma, pero le permite obtener "cada transacción" correctamente. Puede usarlo para lograr lo que espera de las transacciones autónomas.

Triggers

Los triggers son un tipo de constricción, un elemento fundamental en la estructura de las bases de datos. Donde una constricción define una regla algebraica, el trigger define una regla algoritmica. Muy útil desde el punto de vista funcional.

0.88. ¿Qué es un trigger?

El trigger es una función que afecta a una tabla y controla ciertas operaciones sobre ella.

Veamos dos ejemplos que demeustrasn la utilidad de los triggers:

- Constricción lógica. Con REFERENCES definimos las constricciones sobre claves foráneas.

  También, podemos definir constricciones de tipo "de este tipo o de este otro tipo". Como tener dos campos que son claves foráneas, pero para un registro no es posible tener los dos campos asignados. Por tanto, nos es muy útil tener una función de control automática que a la vez que escribe un valor en un campo, verifique que se cumpla cierta condición para el nuevo valor.

- Utilidad funcional. Una de nuestras tablas contiene un campo texto y requerimos grabar una traza de las sucesivas versiones del texto.

  Más que manejarla dentro de nuestra aplicación (con el riesgo del mantenimiento evolutivo, al que un desarrollador se compromete para conservar la integridad de dicha gestión en una nueva porción de código que permita modificar el texto), es más onfiable y simple definir un trigger, controlador sobre UPDATE, que irá archivando la versión anterior del texto antes de efectuar la actualización.

Un trigger es un controlador que se verifica antes de una instrucción de tipo INSERT, DELETE o UPDATE. De hecho, puede ser llamado para cada uno de los registros sujetos a la instrucción o una sola vez.

0.89. ¿Cómo hacer un trigger?

Lo primero es programar un procedimiento almacenado.

Este procedimiento (o función) constituye el trigger propiamente. Esta función no admite argumentos y regresa el tipo OPAQUE (tipo indeterminado).

Cuando una función es llamada por un trigger, hereda automáticamente, en tiempo de ejecución, un cierto número de variables particularmente útiles, como los citados a continuación:

- NEW: el registro que se genera después de la ejecución de la instrucción que invoca el trigger.

- OLD: el estado del regsitro antes de la ejecución de la instrucción que invoca eltrigger.

Veamos un pequeño ejemplo. Crearemos dos tablas, una para artículos escritos y la otra para las versiones sucesivas de los artículos.

```
Prueba1=# create table article(id integer not null primary key,
date_modif
date not null default now(), texte text, status varchar);
NOTICE:~ CREATE TABLE/PRIMARY KEY will create implicit
index 'article_pkey' for table 'article'
CREATE

Prueba1=# create table archive(id integer, date_modif date not null,
texte text);
CREATE

Prueba1=#
```

programamos una función que será invocada durante la modificación de un artículo con el objetivo de conservar la versión anterior de dicho artículo:

```
DROP FUNCTION arch_art();
CREATE FUNCTION arch_art() RETURNS OPAQUE AS '
BEGIN
IF NEW.texte != OLD.texte THEN
INSERT INTO archive(id, date_modif, texte)
VALUES (OLD.id,OLD.date_modif,OLD.texte);
END IF;
RETURN NEW;
END;
' LANGUAGE 'plpgsql';
```

La última línea (RETURN NEW), retorna el registro que deberá ser escrito en la base de datos.

En nuestro caso, permanece sin cambios con relación al comando UPDATE original.

Resaltamos que el trigger será llamado por cada comando UPDATE, debido a que es necesario verificar que el texto del artículo ha sido modificado antes de grabarlo.

Luego de programar el trigger, le indicamos a la base de datos la tabla, las operaciones y el procedimiento almacenado respectivos:

```
DROP TRIGGER trg_arch_art ON article;

CREATE TRIGGER trg_arch_art BEFORE DELETE OR UPDATE ON
article
FORE ACH ROW EXECUTE PROCEDURE arch_art();
Verificamos que el trigger trabaja correctamente:
Prueba1=# insert into article(id,texte) values(1,'Ceci est le premier
article original');
INSERT 40942 1

Prueba1=# select * from article;
id | date_modif | texte                                    | status
----+------------+------------------------------------------+-------
1  | 2001-11-19 | Ceci est le premier article original |
```

```
(1 row)

Prueba1=# select * from archive;
id | date_modif | texte
----+------------+------
(0 rows)

Prueba1=# update article set texte='Ceci est le premier article version
2', date_modif=now();
UPDATE 1

Prueba1=# select * from article;
id | date_modif | texte                                      | status
----+------------+-------------------------------------------+-------
1  | 2001-11-19 | Ceci est le premier article version 2 |
(1 row)

Prueba1=# select * from archive;
id | date_modif | texte
----+------------+--------------------------------------

1 | 2001-11-19 | Ceci est le premier article original
(1 row)

Prueba1=#
```

ahora modificaremos nuestro trigger con el propósito de rechazar las modificaciones si el campo status indica 'publicado':

```
DROP FUNCTION arch_art();
CREATE FUNCTION arch_art() RETURNS OPAQUE AS '
BEGIN
IF NEW.texte != OLD.texte THEN
IF OLD.status = ''publicado'' THEN
RAISE EXCEPTION ''Article % déjà publié'',NEW.id;
END IF;
INSERT INTO archive(id, date_modif, texte)
VALUES (OLD.id,OLD.date_modif,OLD.texte);
END IF;
RETURN NEW;
END;
' LANGUAGE 'plpgsql';
DROP TRIGGER trg_arch_art ON article;

CREATE TRIGGER trg_arch_art BEFORE DELETE OR UPDATE ON
article
FOR EACH ROW EXECUTE PROCEDURE arch_art();
```

Veamos el resultado:

```
Prueba1=# select * from article;
id | date_modif | texte                                      | status
----+------------+-------------------------------------------+-------
1  | 2001-11-19 | Ceci est le premier article version 2 |
```

```
(1 row)

Prueba1=# select * from archive;
id | date_modif | texte
----+------------+-------------------------------------
1  | 2001-11-19 | Ceci est le premier article original
(1 row)

Prueba1=# update article set status='publié' where id=1;
UPDATE 1

Prueba1=# select * from article;
id | date_modif | texte                                      | status
----+------------+--------------------------------------------+-------
1  | 2001-11-19 | Ceci est le premier article version 2 | publié
(1 row)

Prueba1=# select * from archive;
id | date_modif | texte
----+------------+-------------------------------------
1  | 2001-11-19 | Ceci est le premier article original
(1 row)

Prueba1=# update article set texte='Ceci est une tentative de
modification', date_modif=now();
ERROR: Article 1 déjà publié

Prueba1=#
```

Como podemos ver, los triggers son muy practicos para agilizar la aplicación cliente de la gestión de las reglas de flujo de trabajo, por ejemplo. También son muy aconsejables para generar reglas de seguridad.

Suficiente información para empezar, manos a la obra, de aquí en adelante se indican cuales son los pasos para implementar esta solución:

A. Crear un lenguaje de consulta PL/PgSQL a dicha Base de Datos, para ello escribimos el siguiente comando desde la consola del sistema:

```
createlang -h localhost -U postgres plpgsql Peliculas
```

NOTA: EN caso de que se requiera eliminar el lenguaje de la base de datos utilizamos el siguiente comando:

```
droplang -h localhost -U postgres plpgsql Peliculas
```

B. Crear una tabla llamada “tbl_audit”, donde se guardan cada una de las transacciones realizadas:

```
CREATE TABLE tbl_audit (
pk_audit serial NOT NULL,
"TableName" character(45) NOT NULL,
"Operation" char(1) NOT NULL,
"OldValue" text,
"NewValue" text,
```

```
"UpdateDate" timestamp without time zone NOT NULL,
"UserName" character(45) NOT NULL,
CONSTRAINT pk_audit PRIMARY KEY (pk_audit))
WITH (OIDS=FALSE);
ALTER TABLE tbl_audit OWNER TO postgres;
```

C. Creamos una función en PL/PgSQL que nos permite insertar los datos de aquellos registros que son afectados cada vez que se realiza una acción de tipo INSERT, UPDATE y DELETE en una tabla determinada, la cual es definida en la creación del Trigger.

```
CREATE OR REPLACE FUNCTION fn_log_audit() RETURNS trigger AS
$$
BEGIN
IF (TG_OP = 'DELETE') THEN
INSERT INTO tbl_audit ("TableName", "Operation", "OldValue", "NewValue"
    , "UpdateDate", "UserName")
VALUES (TG_TABLE_NAME, 'D', OLD, NULL, now(), USER);
RETURN OLD;
ELSIF (TG_OP = 'UPDATE') THEN
INSERT INTO tbl_audit ("TableName", "Operation", "OldValue", "NewValue"
    , "UpdateDate", "UserName")
VALUES (TG_TABLE_NAME, 'U', OLD, NEW, now(), USER);
RETURN NEW;
ELSIF (TG_OP = 'INSERT') THEN
INSERT INTO tbl_audit ("TableName", "Operation", "OldValue", "NewValue"
    , "UpdateDate", "UserName")
VALUES (TG_TABLE_NAME, 'I', NULL, NEW, now(), USER);
RETURN NEW;
END IF;
RETURN NULL;
END;
$$
LANGUAGE 'plpgsql' VOLATILE COST 100;
ALTER FUNCTION fn_log_audit() OWNER TO postgres;
```

D. Crear el Trigger en todas las tablas menos en “tbl_audit”, para este caso de ejemplo usamos la tabla tbl_atributos, indicando que será ejecutado el trigger antes de la ejecución de una instrucción INSERT, UPDATE y DELETE para cada registro y le asignamos la función anterior.

```
CREATE TRIGGER tbl_atributos_tg_audit AFTER INSERT OR UPDATE OR DELETE
ON tbl_atributos FOR EACH ROW EXECUTE PROCEDURE fn_log_audit();
```

E. Por ultimo realizaremos una serie de consultas para poner en practica el registro de transacciones:

```
INSERT INTO tbl_atributos (fk_tipo, nombre) VALUES (1, 'Femenido');
UPDATE tbl_atributos SET nombre = 'Masculino' WHERE pk_atributo = 2;
DELETE FROM tbl_atributos WHERE pk_atributo = 2;
```

Al hacer una selección de los datos de la tabla tbl_audit podemos observar en cada uno de los registros la acción que se le realizó a una determinada tabla, cuales son los datos antiguos y nuevos como la fecha de cuando fueron registrados. Por otro lado se registra el usuario que realizo la acción sobre los datos, en este caso si el usuario que se conecto al manejador de base de datos es “postgres”, será el responsable de los registros alterados, para corregir este “posible defecto” de utilizar otro “valor” que represente a un “usuario real del sistema” puede utilizar las variables de sesión que se mencionaron en un post anterior y ajusta la función “fn_log_audit”.

Bibliografía

[1] , *The Practical SQL Handbook , Bowman et al, 1993 , Using Structured Query Language , 3, Judity Bowman, Sandra Emerson, y Marcy Damovsky, 0-201-44787-8, 1996, Addison-Wesley, 1997.*

[2] , *A Guide to the SQL Standard , Date and Darwen, 1997 , A user's guide to the standard database language SQL , 4, C. J. Date y Hugh Darwen, 0-201-96426-0, 1997, Addison-Wesley, 1997.*

[3] , *An Introduction to Database Systems , Date, 1994 , 6, C. J. Date, 1, 1994, Addison-Wesley, 1994.*

[4] , *C. J. Date and Hugh Darwen, A Guide to the SQL Standard: A user's guide to the standard database language SQL, Fourth Edition, Addison-Wesley, ISBN 0-201-96426-0, 1997.*

[5] , *C. J. Date, An Introduction to Database Systems, Eighth Edition, Addison-Wesley, ISBN 0-321-19784-4, 2003.*

[6] , *Ramez Elmasri and Shamkant Navathe, Fundamentals of Database Systems, Fourth Edition, Addison-Wesley, ISBN 0-321-12226-7, 2003.*

[7] , *Zelaine Fong, The design and implementation of the POSTGRES query optimizer 2 , University of California, Berkeley, Computer Science Department.*

[8] , *Discusses SQL history and syntax, and describes the addition of INTERSECT and EXCEPT constructs into PostgreSQL. Prepared as a Master's Thesis with the support of O. Univ. Prof. Dr. Georg Gottlob and Univ. Ass. Mag. Katrin Seyr at Vienna University of Technology.*

[9] , *Jim Melton and Alan R. Simon, Understanding the New SQL: A complete guide, Morgan Kaufmann, ISBN 1-55860-245-3, 1993.*

[10] , *Understanding the New SQL , Melton and Simon, 1993 , A complete guide, Jim Melton y Alan R. Simon, 1-55860-245-3, 1993, Morgan Kaufmann, 1993.*

[11] , *Nels Olson, Partial indexing in POSTGRES: research project, University of California, UCB Engin T7.49.1993 O676, 1993.*

[12] , *L. Ong and J. Goh, "A Unified Framework for Version Modeling Using Production Rules in a Database System", ERL Technical Memorandum M90/33, University of California, April, 1990.*

[13] , *Documentación de PostgreSQL de la distribución: http://www.postgresql.org/docs/*

[14] , *The PostgreSQL Administrator's Guide , The Administrator's Guide , Editado por Thomas Lockhart, 1998-10-01, The PostgreSQL Global Development Group.*

[15] , *The PostgreSQL Developer's Guide , The Developer's Guide , Editado por Thomas Lockhart, 1998-10-01, The PostgreSQL Global Development Group.*

[16] , *The PostgreSQL Programmer's Guide , The Programmer's Guide , Editado por Thomas Lockhart, 1998-10-01, The PostgreSQL Global Development Group.*

[17] , *The PostgreSQL Tutorial Introduction , The Tutorial , Editado por Thomas Lockhart, 1998-10-01, The PostgreSQL Global Development Group.*

[18] , *The PostgreSQL User's Guide , The User's Guide , Editado por Thomas Lockhart, 1998-10-01, The PostgreSQL Global Development Group.*

[19] , *P. Seshadri and A. Swami, "Generalized Partial Indexes (cached version) 4 ", Proc. Eleventh International Conference on Data Engineering, 6-10 March 1995, IEEE Computer Society Press, Cat. No.95CH35724, 1995, 420-7.*

[20] , *L. Rowe and M. Stonebraker, " The POSTGRES data model 3 ", Proc. VLDB Conference, Sept. 1987.*

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

[22] , *Enhancement of the ANSI SQL Implementation of PostgreSQL , Simkovics, 1998 , Stefan Simkovics, O.Univ.Prof.Dr.. Georg Gottlob, November 29, 1998, Department of Information Systems, Vienna University of Technology.*

[23] , *Stefan Simkovics, Enhancement of the ANSI SQL Implementation of PostgreSQL, Department of Information Systems, Vienna University of Technology, November 29, 1998.*

[24] , *M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, " On Rules, Procedures, Caching and Views in Database Systems 10 ", Proc. ACM-SIGMOD Conference on Management of Data, June 1990.*

[25] , *M. Stonebraker and L. Rowe, " The design of POSTGRES 5 ", Proc. ACM-SIGMOD Conference on Management of Data, May 1986.*

[26] , *M. Stonebraker, E. Hanson, and C. H. Hong, "The design of the POSTGRES rules system", Proc. IEEE Conference on Data Engineering, Feb. 1987.*

[27] , *M. Stonebraker, L. A. Rowe, and M. Hirohama, " The implementation of POSTGRES 9 ", Transactions on Knowledge and Data Engineering 2(1), IEEE, March 1990.*

[28] , *M. Stonebraker, M. Hearst, and S. Potamianos, " A commentary on the POSTGRES rules system 7 ", SIGMOD Record 18(3), Sept. 1989.*

[29] , *M. Stonebraker, " The case for partial indexes 8 ", SIGMOD Record 18(4), Dec. 1989, 4-11.*

[30] , *M. Stonebraker, " The design of the POSTGRES storage system 6 ", Proc. VLDB Conference, Sept. 1987.*

[31] , *Jeffrey D. Ullman, Principles of Database and Knowledge: Base Systems, Volume 1, Computer Science Press, 1988.*

[32] , *Principles of Database and Knowledge : Base Systems , Ullman, 1988 , Jeffrey D. Ullman, 1, Computer Science Press , 1988 .*

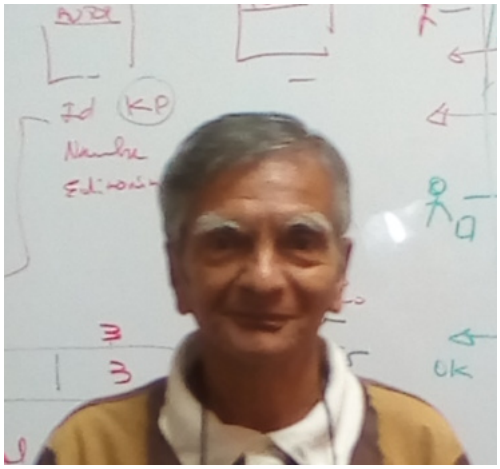
[33] , *Worsley, John C.; Drake, Joshua D. (2002). Practical PostgreSQL. O'Reilly.*

[34] , *A. Yu and J. Chen, The POSTGRES Group, The Postgres95 User Manual, University of California, Sept. 5, 1995.*

[35] , *The Postgres95 User Manual , Yu and Chen, 1995 , A. Yu y J. Chen, The POSTGRES Group , Sept. 5, 1995, University of California, Berkeley CA.*

Acerca del autor

Graduado en Física, Facultad de Ciencias, Universidad Nacional Autónoma de México (UNAM), Doctor en Ciencias de la Computación, Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas (IIMAS) UNAM. Especialista en Economía Matemática, Centro de Investigación y Docencia Económica, CIDE, México. Cursante del Doctorado en Minería de Datos, Modelos y Sistemas Expertos por la Universidad de Illinois en Urbana-Champaings (USA).



Ha sido profesor en la Facultad de Química y en la Facultad de Ingeniería, UNAM. Fue profesor en el Departamento de Ciencias Básicas, Universidad de Las Américas en Cholula, (UDLAP), Puebla, México y, durante seis años, profesor visitante en la Universidade Federal de Rio Grande do Sul (Brasil), profesor y jefe de sistemas postgrados de agronomía y veterinaria, Universidad Central de Venezuela (UCV), actualmente es profesor del Departamento de informática y del Departamento de Postgrado, Universidad Politécnica Territorial de Aragua (UPT Aragua), Venezuela.

Expositor y conferencista a nivel nacional e internacional.

Es asesor en mejora de procesos, gestión de proyectos, desarrollo de software corporativo en los sectores de servicios, banca, industria y gobierno.

El Dr. Domínguez es un especialista reconocido en base de datos, desarrollo de software y servidores en el área del software libre, así como un experto en LINUX DEBIAN.

En la actualidad orienta su trabajo a la creación y desarrollo de equipos de software de alto desempeño. Autor de múltiples artículos y libros sobre la materia.