

PostgreSQL

Equipo de Desarrollo PostgreSQL

**Editado por
Thomas Lockhart**

PostgreSQL

por Equipo de Desarrollo PostgreSQL

Editado por Thomas Lockhart

PostgreSQL

tiene Copyright © 1996-9 de Postgres Global Development Group.

Tabla de contenidos

Resumen	??
I. Guía de Usuario.....	??
1. Introduction	??
¿Qué es Postgres?.....	??
Breve historia de Postgres	??
Acerca de esta versión.....	??
Recursos	??
Terminología.....	??
Notación.....	??
Y2K Statement (Informe sobre el efecto 2000)	??
Copyrights y Marcas Registradas	??
2. Sintaxis SQL	??
Palabras Clave.....	??
Comentarios	??
Nombres.....	??
Constantes.....	??
Campos y Columnas	??
Operadores	??
Expresiones.....	??
3. Data Types	??
Numeric Types	??
Monetary Type	??
Character Types	??
Date/Time Types	??
Boolean Type	??
Geometric Types	??
IP Version 4 Networks and Host Addresses.....	??
4. Operadores.....	??
Lexical Precedence.....	??
Operadores generales.....	??
Operadores numéricos.....	??
Operadores geométricos.....	??
Operadores de intervalos de tiempo	??
Operadores IP V4 CIDR.....	??
Operadores IP V4 INET	??
5. Funciones.....	??
Funciones SQL	??
Funciones Matemáticas.....	??
String Functions	??
Funciones de Fecha/Hora	??
Funciones de Formato.....	??
Funciones Geométricas	??
Funciones PostgresIP V4	??
6. Conversión de tipos	??
Conceptos generales.....	??
Operadores	??
Funciones	??
Resultados de consultas.....	??
Consultas UNION	??
7. Indices and Keys.....	??
Keys	??
Partial Indices	??
8. Matrices	??
9. Herencia.....	??

10. Multi-Version Concurrency Control (Control de la Concurrency Multi	
Versión)	??
Introducción	??
Aislamiento transaccional	??
Nivel de lectura cursada	??
Nivel de aislamiento serializable	??
Bloqueos y tablas	??
Bloqueo e índices	??
Chequeos de consistencia de datos en el nivel de aplicación	??
11. Configurando su entorno	??
12. Administración de una Base de Datos	??
Creación de Bases de Datos	??
Ubicaciones Alternativas de las Bases de Datos	??
Acceso a una Base de Datos	??
Destrucción de una Base de Datos	??
13. Almacenamiento en disco	??
14. Instrucciones SQL	??
ABORT	??
MODIFICAR GRUPO	??
MODIFICAR TABLA	??
MODIFICAR USUARIO	??
BEGIN	??
CLOSE	??
CLUSTER	??
COMMIT	??
COPY	??
CREATE AGGREGATE	??
CREATE DATABASE	??
CREATE FUNCTION	??
CREATE GROUP	??
CREATE INDEX	??
CREATE LANGUAGE	??
CREATE OPERATOR	??
CREATE RULE	??
CREATE SEQUENCE	??
CREATE TABLE	??
CREATE TABLE AS	??
CREATE TRIGGER	??
CREATE TYPE	??
CREAR USUARIO	??
CREAR VISTA	??
DECLARE	??
DELETE	??
DROP AGGREGATE	??
DROP DATABASE	??
DROP FUNCTION	??
DROP GROUP	??
DROP INDEX	??
DROP LANGUAGE	??
DROP OPERATOR	??
DROP RULE	??
DROP SEQUENCE	??
DROP TABLE	??
DROP TRIGGER	??
DROP TYPE	??
DROP USER	??

DROP VIEW	??
END	??
EXPLAIN	??
FETCH.....	??
GRANT	??
INSERT	??
LISTEN	??
LOAD	??
LOCK.....	??
MOVE.....	??
NOTIFY	??
RESET	??
REVOKE.....	??
ROLLBACK	??
SELECT	??
SELECT INTO	??
SET	??
SHOW.....	??
TRUNCATE	??
UNLISTEN.....	??
UPDATE.....	??
VACUUM.....	??
15. Aplicaciones	??
createdb.....	??
createlang.....	??
createuser.....	??
dropdb.....	??
droplang.....	??
dropuser	??
ecpg	??
pgaccess.....	??
pgadmin.....	??
pg_dump.....	??
pg_dumpall.....	??
psql	??
pgtclsh.....	??
pgtksh	??
vacuumdb.....	??
16. Aplicaciones del sistema	??
initdb.....	??
initlocation	??
ipcclean.....	??
pg_passwd.....	??
pg_upgrade.....	??
postgres.....	??
postmaster.....	??
II. Guía del Administrador	??
17. Portes.....	??
Plataformas actualmente soportadas.....	??
Plataformas no soportadas.....	??
18. Opciones de Configuración	??
Parámetros de configuración (configure).....	??
Parámetros de construcción (make).....	??
Soporte Local.....	??
Autenticación Kerberos	??
19. Distribución del Sistema	??

20. Instalación	??
Antes de comenzar	??
Procedimiento de Instalación	??
21. Instalacion en Win32	??
Construccion de librerias	??
Instalacion de las librerias	??
Usando las librerias	??
22. Entorno de tiempo de ejecución	??
Utilizando Postgres desde Unix	??
Iniciando postmaster	??
Usando pg_options	??
23. Seguridad	??
Autenticacion de Usuarios	??
Nombres de usuario y grupos	??
Control de Acceso	??
Funciones y Reglas	??
24. Agregar y Eliminar Usuarios	??
25. Gestión de Disco	??
Localizaciones Alternativas	??
26. Gestión de una base de datos	??
Creación de una base de datos	??
Acceso a la base de datos	??
Destrucción de una base de datos	??
Copia de seguridad y restauración	??
27. Tratamiento de problemas	??
Fallos de inicio de Postmaster	??
Problemas con la conexión del Cliente	??
Depuración de mensajes	??
28. Recuperación de bases de datos	??
29. Pruebas de regresión	??
Entorno de regresión	??
Estructura de directorios	??
Procedimiento para el test de regresión	??
Análisis de Regresión	??
Archivos de comparación específicos de la plataforma	??
30. Notas de versiones	??
Version 6.5.3	??
Version 6.5.2	??
Version 6.5.1	??
Version 6.5	??
Version 6.4.2	??
Version 6.4.1	??
Version 6.4	??
Version 6.3.2	??
Version 6.3.1	??
Version 6.3	??
Version 6.2.1	??
Version 6.2	??
Version 6.1.1	??
Version 6.1	??
Version v6.0	??
Version v1.09	??
Version v1.02	??
Version v1.01	??
Version v1.0	??
Postgres95 Beta 0.03	??

Postgres95 Beta 0.02	??
Postgres95 Beta 0.01	??
Tiempos Resultantes.....	??
III. Guía del Programador.....	??
31. Arquitectura	??
Conceptos de Arquitectura de Postgres	??
32. Extensor SQL: Preludio	??
Como hacer extensible el trabajo.....	??
El Tipo de Sistema de Postgres	??
Acerca de los Sistema de Catalogo de Postgres	??
33. Extendiendo SQL: Funciones	??
Funciones de Lenguaje de Consultas (SQL)	??
Funciones de Lenguaje Procedural	??
Funciones Internas.....	??
Funciones de Lenguaje Compilado (C)	??
Sobrecarga de funciones	??
34. Extendiendo SQL: Tipos.....	??
Tipos Definidos por el Usuario.....	??
35. Extendiendo SQL: Operadores.....	??
Información de optimización de operador	??
36. Extensiones de SQL: Agregados	??
37. El Sistema de reglas de Postgres	??
¿Qué es un árbol de query?	??
Las vistas y el sistema de reglas.	??
Reglas sobre INSERT, UPDATE y DELETE	??
Reglas y permisos	??
Reglas frente triggers	??
38. Utilización de las Extensiones en los Índices	??
39. GiST Indices	??
40. Enlazando funciones de carga dinámica	??
ULTRIX.....	??
DEC OSF/1.....	??
SunOS 4.x, Solaris 2.x y HP-UX	??
41. Triggers (disparadores).....	??
Creación de Triggers.....	??
Interacción con el Trigger Manager	??
Visibilidad de Cambios en Datos	??
Ejemplos.....	??
42. Server Programming Interface	??
Interface Functions	??
Interface Support Functions.....	??
Memory Management.....	??
Visibility of Data Changes	??
Examples.....	??
43. Lenguajes Procedurales.....	??
Instalación de lenguajes procedurales	??
PL/pgSQL.....	??
PL/Tcl.....	??
IV. Interfaces.....	??
44. Funciones.....	??
45. Objetos Grandes	??
Nota Histórica	??
Características de la Implementación.....	??
Interfaces	??
Funciones registradas Incorporadas	??

Accediendo a Objetos Grandes desde LIBPQ	??
Programa de Ejemplo.....	??
I. CCVS API Functions	??
46. libpq.....	??
Funciones de Conexión a la Base de Datos.....	??
Funciones de Ejecución de Consultas.....	??
Procesamiento Asíncrono de Consultas.....	??
Ruta Rápida	??
Notificación Asíncrona	??
Funciones Asociadas con el Comando COPY	??
Funciones de Trazado de libpq	??
Funciones de control de libpq.....	??
Variables de Entorno	??
Programas de Ejemplo	??
47. libpq C++ Binding.....	??
Control e Inicialización	??
Clases de libpq++	??
Funciones de Conexión a la Base de Datos.....	??
Funciones de Ejecución de las Consultas.....	??
Notificación Asíncrona	??
Funciones Asociadas con el Comando COPY.....	??
48. pgctl.....	??
Comandos	??
Ejemplos.....	??
Información de referencia de comandos pgctl	??
49. Interfaz ODBC	??
Trasfondo	??
Aplicaciones Windows	??
Instalación Unix	??
Ficheros de Configuración.....	??
ApplixWare.....	??
50. JDBC Interface	??
Building the JDBC Interface	??
Preparing the Database for JDBC	??
Using the Driver	??
Importing JDBC	??
Loading the Driver	??
Connecting to the Database	??
Issuing a Query and Processing the Result.....	??
Performing Updates.....	??
Closing the Connection.....	??
Using Large Objects.....	??
Postgres Extensions to the JDBC API	??
Further Reading	??
51. Interfaz de Programación Lisp	??
V. Guía del Desarrollador.....	??
52. Código Fuente Postgres.....	??
Formateo	??
53. Revisión de las características internas de PostgreSQL	??
El camino de una consulta.....	??
Cómo se establecen las conexiones	??
La etapa de traducción.....	??
El sistema de reglas de Postgres	??
Planificador/optimizador	??
Ejecutor.....	??

54. pg_options	??
55. Optimización Genética de Consulta en Sistemas de Base de Datos	??
Planificador de consulta para un Problema Complejo de Optimización	??
Algoritmo Genéticos (AG)	??
Optimización Genética de Consultas (GEQO) en Postgres	??
Futuras Tareas de Implementación para el OGEC de Postgres	??
56. Protocolo Frontend/Backend	??
Introducción	??
Protocolo	??
Tipos de Datos de Mensajes	??
Formatos de Mensajes	??
57. Señales de Postgres	??
58. gcc Default Optimizations	??
59. Interfaces de Backend	??
Formato de fichero BKI	??
Comandos Generales	??
Macro Commands	??
Comandos de Depuración	??
Ejemplo	??
60. Ficheros de páginas	??
Estructura de la página	??
Ficheros	??
Bugs	??
VI. Tutorial	??
61. SQL	??
El Modelo de Datos Relacional	??
Formalidades del Modelo Relacional de Datos	??
Operaciones en el Modelo de Datos Relacional	??
El Lenguaje SQL	??
62. Arquitectura	??
Postgres Conceptos de arquitectura	??
63. Empezando	??
Configurando el entorno	??
Ejecución del Monitor Interactivo (psql)	??
Administrando una Base de datos	??
64. El Lenguaje de consultas	??
Monitor interactivo	??
Conceptos	??
Creación de una nueva clase	??
Llenando una clase con instancias	??
Consutar a una clase	??
Redireccionamiento de consultas SELECT	??
Joins (uniones) entre clases	??
Actualizaciones	??
Borrados	??
Uso de funciones de conjunto	??
65. Características Avanzadas de SQL en Postgres	??
Herencia	??
Valores No-Atómicos	??
Time Travel (Viaje en el tiempo)	??
Más características avanzadas	??
VII. Apéndices	??
UG1. ayuda de fecha/hora	??
Zonas horarias	??
Historia	??

DG1. El Repositorio del CVS	??
Organización del árbol de CVS	??
Tomando Las Fuentes Vía CVS Anónimo.....	??
Tomando Los Fuentes Vía CVSup.....	??
DG2. Documentación.....	??
Mapa de la documentación	??
El proyecto de documentación	??
Fuentes de la documentación	??
Haciendo documentaciones	??
Páginas man	??
Generación de copias impresas para v6.5	??
Herramientas.....	??
Otras herramientas	??
Bibliografía	??

Resumen

Postgres, desarrollado originalmente en el Departamento de Ciencias de Computación de la UC de Berkeley, fue pionero de muchos de los conceptos referentes a objetos que ahora están disponibles en algunas bases de datos comerciales. Proporciona soporte a lenguaje SQL92/SQL93, integridad en transacciones y capacidad para extensión de tipos. PostgreSQL es un descendiente open-source de este código original de Berkeley.

Capítulo 1. Introduction

Este documento es el manual de usuario del sistema de mantenimiento de bases de datos PostgreSQL¹, originariamente desarrollado en la Universidad de California en Berkeley. PostgreSQL está basada en Postgres release 4.2². El proyecto Postgres, liderado por el Profesor Michael Stonebraker, fue esponsorizado por diversos organismos oficiales u oficiosos de los EEUU: la Agencia de Proyectos de Investigación Avanzada de la Defensa de los EEUU (DARPA), la Oficina de Investigación de la Armada (ARO), la Fundación Nacional para la Ciencia (NSF), y ESL, Inc.

¿Qué es Postgres?

Los sistemas de mantenimiento de Bases de Datos relacionales tradicionales (DBMS,s) soportan un modelo de datos que consisten en una colección de relaciones con nombre, que contienen atributos de un tipo específico. En los sistemas comerciales actuales, los tipos posibles incluyen numéricos de punto flotante, enteros, cadenas de caracteres, cantidades monetarias y fechas. Está generalmente reconocido que este modelo será inadecuado para las aplicaciones futuras de procesado de datos. El modelo relacional sustituyó modelos previos en parte por su "simplicidad espartana". Sin embargo, como se ha mencionado, esta simplicidad también hace muy difícil la implementación de ciertas aplicaciones. Postgres ofrece una potencia adicional sustancial al incorporar los siguientes cuatro conceptos adicionales básicos en una vía en la que los usuarios pueden extender fácilmente el sistema

- clases
- herencia
- tipos
- funciones

Otras características aportan potencia y flexibilidad adicional:

- Restricciones (Constraints)
- Disparadores (triggers)
- Reglas (rules)
- Integridad transaccional

Estas características colocan a Postgres en la categoría de las Bases de Datos identificadas como *objeto-relacionales*. Nótese que éstas son diferentes de las referidas como *orientadas a objetos*, que en general no son bien aprovechables para soportar lenguajes de Bases de Datos relacionales tradicionales. Postgres tiene algunas características que son propias del mundo de las bases de datos orientadas a objetos. De hecho, algunas Bases de Datos comerciales han incorporado recientemente características en las que Postgres fue pionera. .

Breve historia de Postgres

El Sistema Gestor de Bases de Datos Relacionales Orientadas a Objetos conocido como PostgreSQL (y brevemente llamado Postgres95) está derivado del paquete Postgres escrito en Berkeley. Con cerca de una década de desarrollo tras él, PostgreSQL es el gestor de bases de datos de código abierto más avanzado hoy en día, ofreciendo control de concurrencia multi-versión, soportando casi toda la sintaxis SQL (incluyendo subconsultas, transacciones, y tipos y funciones definidas por el usuario), contando también con un amplio conjunto de enlaces con lenguajes de programación (incluyendo C, C++, Java, perl, tcl y python).

El proyecto Postgres de Berkeley

La implementación del DBMS Postgres comenzó en 1986. Los conceptos iniciales para el sistema fueron presentados en *The Design of Postgres* y la definición del modelo de datos inicial apareció en *The Postgres Data Model*. El diseño del sistema de reglas fue descrito en ese momento en *The Design of the Postgres Rules System*. La lógica y arquitectura del gestor de almacenamiento fueron detalladas en *The Postgres Storage System*.

Postgres ha pasado por varias revisiones importantes desde entonces. El primer sistema de pruebas fue operacional en 1987 y fue mostrado en la Conferencia ACM-SIGMOD de 1988. Lanzamos la Versión 1, descrita en *The Implementation of Postgres*, a unos pocos usuarios externos en Junio de 1989. En respuesta a una crítica del primer sistema de reglas (*A Commentary on the Postgres Rules System*), éste fue rediseñado (*On Rules, Procedures, Caching and Views in Database Systems*) y la Versión 2, que salió en Junio de 1990, lo incorporaba. La Versión 3 apareció en 1991 y añadió una implementación para múltiples gestores de almacenamiento, un ejecutor de consultas mejorado y un sistema de reescritura de reglas nuevo. En su mayor parte, las siguientes versiones hasta el lanzamiento de Postgres95 (ver más abajo) se centraron en mejorar la portabilidad y la fiabilidad.

Postgres forma parte de la implementación de muchas aplicaciones de investigación y producción. Entre ellas: un sistema de análisis de datos financieros, un paquete de monitorización de rendimiento de motores a reacción, una base de datos de seguimiento de asteroides y varios sistemas de información geográfica. También se ha utilizado como una herramienta educativa en varias universidades. Finalmente, Illustra Information Technologies³ (posteriormente absorbida por Informix⁴) tomó el código y lo comercializó. Postgres llegó a ser el principal gestor de datos para el proyecto científico de computación Sequoia 2000⁵ a finales de 1992.

El tamaño de la comunidad de usuarios externos casi se duplicó durante 1993. Pronto se hizo obvio que el mantenimiento del código y las tareas de soporte estaban ocupando tiempo que debía dedicarse a la investigación. En un esfuerzo por reducir esta carga, el proyecto terminó oficialmente con la Versión 4.2.

Postgres95

En 1994, Andrew Yu⁶ y Jolly Chen⁷ añadieron un intérprete de language SQL a Postgres. Postgres95 fue publicado a continuación en la Web para que encontrara su propio hueco en el mundo como un descendiente de dominio público y código abierto del código original Postgres de Berkeley.

El código de Postgres95 fue adaptado a ANSI C y su tamaño reducido en un 25%. Muchos cambios internos mejoraron el rendimiento y la facilidad de mantenimiento. Postgres95 v1.0.x se ejecutaba en torno a un 30-50% más rápido en el Wisconsin Benchmark comparado con Postgres v4.2. Además de corrección de errores, éstas fueron las principales mejoras:

- El language de consultas Postquel fue reemplazado con SQL (implementado en el servidor). Las subconsultas no fueron soportadas hasta PostgreSQL (ver más abajo), pero podían ser emuladas en Postgres95 con funciones SQL definidas por el usuario. Las funciones agregadas fueron reimplementadas. También se añadió una implementación de la cláusula GROUP BY. La interfaz `libpq` permaneció disponible para programas escritos en C.
- Además del programa de monitorización, se incluyó un nuevo programa (`psql`) para realizar consultas SQL interactivas usando la librería GNU `readline`.

- Una nueva librería de interfaz, `libpgtcl`, soportaba clientes basados en Tcl. Un shell de ejemplo, `pgtclsh`, aportaba nuevas órdenes Tcl para interactuar con el motor Postgres95 desde programas `tcl`.
- Se revisó la interfaz con objetos grandes. Los objetos grandes de Inversion fueron el único mecanismo para almacenar objetos grandes (el sistema de archivos de Inversion fue eliminado).
- Se eliminó también el sistema de reglas a nivel de instancia, si bien las reglas siguieron disponibles como reglas de reescritura.
- Se distribuyó con el código fuente un breve tutorial introduciendo las características comunes de SQL y de Postgres95.
- Se utilizó GNU make (en vez de BSD make) para la compilación. Postgres95 también podía ser compilado con un gcc sin parches (al haberse corregido el problema de alineación de variables de longitud doble).

PostgreSQL

En 1996, se hizo evidente que el nombre “Postgres95” no resistiría el paso del tiempo. Elegimos un nuevo nombre, PostgreSQL, para reflejar la relación entre el Postgres original y las versiones más recientes con capacidades SQL. Al mismo tiempo, hicimos que los números de versión partieran de la 6.0, volviendo a la secuencia seguida originalmente por el proyecto Postgres.

Durante el desarrollo de Postgres95 se hizo hincapié en identificar y entender los problemas en el código del motor de datos. Con PostgreSQL, el énfasis ha pasado a aumentar características y capacidades, aunque el trabajo continúa en todas las áreas.

Las principales mejoras en PostgreSQL incluyen:

- Los bloqueos de tabla han sido sustituidos por el control de concurrencia multi-versión, el cual permite a los accesos de sólo lectura continuar leyendo datos consistentes durante la actualización de registros, y permite copias de seguridad en caliente desde `pg_dump` mientras la base de datos permanece disponible para consultas.
- Se han implementado importantes características del motor de datos, incluyendo subconsultas, valores por defecto, restricciones a valores en los campos (constraints) y disparadores (triggers).
- Se han añadido funcionalidades en línea con el estándar SQL92, incluyendo claves primarias, identificadores entrecomillados, forzado de tipos cadena literales, conversión de tipos y entrada de enteros binarios y hexadecimales.
- Los tipos internos han sido mejorados, incluyendo nuevos tipos de fecha/hora de rango amplio y soporte para tipos geométricos adicionales.
- La velocidad del código del motor de datos ha sido incrementada aproximadamente en un 20-40%, y su tiempo de arranque ha bajado el 80% desde que la versión 6.0 fue lanzada.

Acerca de esta versión

PostgreSQL está disponible sin coste. Este manual describe la version 6.5 de PostgreSQL.

Se usará Postgres para referirse a la versión distribuida como PostgreSQL.

Compruebe la Guía del Administrador para ver la lista de plataformas soportadas. En general, Postgres puede portarse a cualquier sistema compatible Unix/Posix con soporte completo a la librería libc.

Recursos

Este manual está organizado en diferentes partes:

Tutorial

Introducción para nuevos usuarios. No cubre características avanzadas.

Guía del Usuario

Información general para el usuario, incluye comandos y tipos de datos.

Guía del Programador

Información avanzada para programadores de aplicaciones. Incluyendo tipos y extensión de funciones, librería de interfaces y lo referido al diseño de aplicaciones.

Guía del Administrador

Información sobre instalación y administración. Lista de equipo soportado.

Guía del Desarrollador

Información para desarrolladores de Postgres. Este documento es para aquellas personas que están contribuyendo al proyecto de Postgres; la información referida al desarrollo de aplicaciones aparece en la *Guía del Programador*. Actualmente incluido en la *Guía del Programador*.

Manual de Referencia

Información detallada sobre los comandos. Actualmente incluido en la *Guía del Usuario*.

Además de este manual, hay otros recursos que le servirán de ayuda para la instalación y el uso de Postgres:

man pages

Las páginas de manual(man pages) contienen más información sobre los comandos.

FAQs(Preguntas Frecuentes)

La sección de Preguntas Frecuentes(FAQ) contiene respuestas a preguntas generales y otros asuntos que tienen que ver con la plataforma en que se desarrolle.

LEAME(READMEs)

Los archivos llamados LEAME(README) están disponibles para algunas contribuciones.

Web Site

El sitio web de Postgres⁸ contiene información que algunas distribuciones no incluyen. Hay un catálogo llamado *mhonarc* que contiene el histórico de las listas de correo electrónico. Aquí podrá encontrar bastante información.

Listas de Correo

La lista de correo *pgsql-general*⁹ (*archive*¹⁰) es un buen lugar para contestar sus preguntas.

Usted!

Postgres es un producto de código abierto. Como tal, depende de la comunidad de usuarios para su soporte. A medida que empiece a usar Postgres, empezará a depender de otros para que le ayuden, ya sea por medio de documentación o en las listas de correo. Considere contribuir lo que aprenda. Si aprende o descubre algo que no esté documentado, escríbalo y contribuya. Si añade nuevas características al código, hágalas saber.

Aun aquellos con poca o ninguna experiencia pueden proporcionar correcciones y cambios menores a la documentación, lo que es una buena forma de empezar. El *pgsql-docs*¹¹ (*archivo*¹²) de la lista de correos es un buen lugar para comenzar sus pesquisas.

Terminología

En la documentación siguiente, *sitio* (o *site*) se puede interpretar como la máquina en la que está instalada Postgres. Dado que es posible instalar más de un conjunto de bases de datos Postgres en una misma máquina, este término denota, de forma más precisa, cualquier conjunto concreto de programas binarios y bases de datos de Postgres instalados.

El *superusuario* de Postgres es el usuario llamado *postgres* que es dueño de los ficheros de la bases de datos y binarios de Postgres. Como superusuario de la base de datos, no le es aplicable ninguno de los mecanismos de protección y puede acceder a cualquiera de los datos de forma arbitraria. Además, al superusuario de Postgres se le permite ejecutar programas de soporte que generalmente no están disponibles para todos los usuarios. Tenga en cuenta que el superusuario de Postgres *no* es el mismo que el superusuario de Unix (que es conocido como *root*). El superusuario debería tener un identificador de usuario (*UID*) distinto de cero por razones de seguridad.

El *administrador de la base de datos* (*database administrator*) o DBA, es la persona responsable de instalar Postgres con mecanismos para hacer cumplir una política de seguridad para un sitio. El DBA puede añadir nuevos usuarios por el método descrito más adelante y mantener un conjunto de bases de datos plantilla para usar *concreatedb*.

El *postmaster* es el proceso que actúa como una puerta de control (*clearing-house*) para las peticiones al sistema Postgres. Las aplicaciones frontend se conectan al *postmaster*, que mantiene registros de los errores del sistema y de la comunicación entre los procesos backend. El *postmaster* puede aceptar varios argumentos desde la línea de órdenes para poner a punto su comportamiento. Sin embargo, el proporcionar argumentos es necesario sólo si se intenta trabajar con varios sitios o con uno que no se ejecuta a la manera por defecto.

El backend de Postgres (el programa ejecutable *postgres real*) lo puede ejecutar el superusuario directamente desde el intérprete de órdenes de usuario de Postgres (con el nombre de la base de datos como un argumento). Sin embargo, hacer esto eli-

mina el buffer pool compartido y bloquea la tabla asociada con un postmaster/sitio, por ello esto no está recomendado en un sitio multiusuario.

Notación

"..." o `/usr/local/pgsql/` delante de un nombre de fichero se usa para representar el camino (path) al directorio home del superusuario de Postgres.

En la sinopsis, los corchetes ("`[`" y "`]`") indican una expresión o palabra clave opcional. Cualquier cosa entre llaves ("`{`" y "`}`") y que contenga barras verticales ("`|`") indica que debe elegir una de las opciones que separan las barras verticales.

En los ejemplos, los paréntesis ("`(`" y "`)`") se usan para agrupar expresiones booleanas. "`|`" es el operador booleano OR.

Los ejemplos mostrarán órdenes ejecutadas desde varias cuentas y programas. Las órdenes ejecutadas desde la cuenta del root estarán precedidas por "`>`". Las órdenes ejecutadas desde la cuenta del superusuario de Postgres estarán precedidas por "`%`", mientras que las órdenes ejecutadas desde la cuenta de un usuario sin privilegios estarán precedidas por "`$`". Las órdenes de SQL estarán precedidas por "`=>`" o no estarán precedidas por ningún prompt, dependiendo del contexto.

Nota: En el momento de escribir (Postgres v6.5) la notación de las órdenes flagging (o flojos) no es universalmente estable o congruente en todo el conjunto de la documentación. Por favor, envíe los problemas a la Lista de Correo de la Documentación (o Documentation Mailing List)¹³.

Y2K Statement (Informe sobre el efecto 2000)

Autor: Escrito por Thomas Lockhart¹⁴ el 22-10-1998.

El Equipo de Desarrollo Global (o Global Development Team) de PostgreSQL proporciona el árbol de código de software de Postgres como un servicio público, sin garantía y sin responsabilidad por su comportamiento o rendimiento. Sin embargo, en el momento de la escritura:

- El autor de este texto, voluntario en el equipo de soporte de Postgres desde Noviembre de 1996, no tiene constancia de ningún problema en el código de Postgres relacionado con los cambios de fecha en torno al 1 de Enero de 2000 (Y2K).
- El autor de este informe no tiene constancia de la existencia de informes sobre el problema del efecto 2000 no cubiertos en las pruebas de regresión, o en otro campo de uso, sobre versiones de Postgres recientes o de la versión actual. Podríamos haber esperado oír algo sobre problemas si existiesen, dada la base que hay instalada y dada la participación activa de los usuarios en las listas de correo de soporte.
- Por lo que el autor sabe, las suposiciones que Postgres hace sobre las fechas que se escriben usando dos números para el año están documentadas en la Guía del Usuario¹⁵ en el capítulo de los tipos de datos. Para años escritos con dos números, la transición significativa es 1970, no el año 2000; ej. "70-01-01" se interpreta como "1970-01-01", mientras que "69-01-01" se interpreta como "2069-01-01".

- Los problemas relativos al efecto 2000 en el SO (sistema operativo) sobre el que esté instalado Postgres relacionados con la obtención de "la fecha actual" se pueden propagar y llegar a parecer problemas sobre el efecto 2000 producidos por Postgres.

Diríjase a The Gnu Project¹⁶ y a The Perl Institute¹⁷ para leer una discusión más profunda sobre el asunto del efecto 2000, particularmente en lo que tiene que ver con el open source o código abierto, código por el que no hay que pagar.

Copyrights y Marcas Registradas

La traducción de los textos de copyright se presenta aquí únicamente a modo de aclaración y no ha sido aprobada por sus autores originales. Los únicos textos de copyright, garantías, derechos y demás legalismos que tienen validez son los originales en inglés o una traducción aprobada por los autores y/o sus representantes legales. .

PostgreSQL tiene Copyright © 1996-2000 por PostgreSQL Inc. y se distribuye bajo los términos de la licencia de Berkeley.

Postgres95 tiene Copyright © 1994-5 por los Regentes de la Universidad de California. Se autoriza el uso, copia, modificación y distribución de este software y su documentación para cualquier propósito, sin ningún pago, y sin un acuerdo por escrito, siempre que se mantengan el copyright del párrafo anterior, este párrafo y los dos párrafos siguientes en todas las copias.

En ningún caso la Universidad de California se hará responsable de daños, causados a cualquier persona o entidad, sean estos directos, indirectos, especiales, accidentales o consiguientes, incluyendo lucro cesante que resulten del uso de este software y su documentación, incluso si la Universidad ha sido notificada de la posibilidad de tales daños.

La Universidad de California rehusa específicamente ofrecer cualquier garantía, incluyendo, pero no limitada únicamente a, la garantía implícita de comerciabilidad y capacidad para cumplir un determinado propósito. El software que se distribuye aquí se entrega "tal y cual", y la Universidad de California no tiene ninguna obligación de mantenimiento, apoyo, actualización, mejoramiento o modificación.

Unix es una marca registrada de X/Open, Ltd. Sun4, SPARC, SunOS y Solaris son marcas registradas de Sun Microsystems, Inc. DEC, DECstation, Alpha AXP y ULTRIX son marcas registradas de Digital Equipment Corp. PA-RISC y HP-UX son marcas registradas de Hewlett-Packard Co. OSF/1 es marca registrada de Open Software Foundation.

Notas

1. <http://postgresql.org/>
2. <http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html>
3. <http://www.illustra.com/>
4. <http://www.informix.com/>
5. http://www.sdsc.edu/0/Parts_Collabs/S2K/s2k_home.html
6. <mailto:ayu@informix.com>
7. <http://http.cs.berkeley.edu/~jolly/>
8. postgresql.org
9. <mailto:pgsql-general@postgresql.org>
10. <http://www.PostgreSQL.ORG/mhonarc/pgsql-general/>

11. <mailto:pgsql-docs@postgresql.org>
12. <http://www.PostgreSQL.ORG/mhonarc/pgsql-docs/>
13. <mailto:docs@postgresql.org>
14. <mailto:lockhart@alumni.caltech.edu>
15. <http://www.postgresql.org/docs/user/datatype.htm>
16. <http://www.gnu.org/software/year2000.html>
17. <http://language.perl.com/news/y2k.html>

Capítulo 2. Sintaxis SQL

Una descripción de la sintaxis general de SQL.

SQL manipula un conjunto de datos. El lenguaje esta compuesto por varias *palabras clave*. Se permite expresiones aritméticas y procedimentales. Nosotros trataremos estos temas en este capítulo; en los sucesivos capítulos incluiremos detalles de los tipos de datos, funciones, y operadores.

Palabras Clave

SQL92 define *Palabras Clave* para el lenguaje que tienen un significado específico. Algunas palabras están *reservadas*, lo cual indica que su aparición esta restringida sólo en ciertos contextos. Otras Palabras clave *no están restringidas*, , lo cual indica que en ciertos contextos tienen un significado específico pero no es obligatorio.

Postgres implementa un subconjunto extendido de los lenguajes SQL92 y SQL3 lenguajes. Algunos elementos del lenguaje no están tan restringidos en esta implementación como en el lenguajes estándar, en parte debido a las características extendidas de Postgres.

Información sobre las palabras clave de SQL92 y SQL3 son derivadas de *Date and Darwen, 1997*.

Palabras clave reservadas

SQL92 y SQL3 tienen *Palabras clave reservadas* las cuales no están permitidas ni como identificador ni para cualquier uso distinto de señales fundamentales en declaraciones SQL . Postgres tiene palabras clave adicionales con las mismas restricciones. En particular, estas palabras clave no están permitidas para nombre de tablas o campos, aunque en algunos casos están permitidas para ser etiquetas de columna (pe. en la cláusula AS).

Sugerencia: Cualquier cadena puede ser especificada como un identificador si va entre doble comillas ("como esa"). Se debe tener cuidado desde tanto un identificador será sensible a las mayúsculas / minúsculas y contendrá espacios en blanco u otro caracteres especiales.

Las siguientes palabras reservadas de Postgres no son palabras reservadas de SQL92 ni de SQL3 Estas están permitidas para ser etiquetas de columna, pero no identificadores:

```
ABORT ANALYZE
BINARY
CLUSTER CONSTRAINT COPY
DO
EXPLAIN EXTEND
LISTEN LOAD LOCK
MOVE
NEW NONE NOTIFY
RESET
SETOF SHOW
UNLISTEN UNTIL
VACUUM VERBOSE
```

Las siguientes palabras reservadas de Postgres son también palabras reservadas de SQL92 o SQL3 y está permitido que estén presente como etiqueta de columna pero no como identificador:

```
CASE COALESCE CROSS CURRENT CURRENT_USER
DEC DECIMAL
ELSE END
FALSE FOREIGN
GLOBAL GROUP
LOCAL
NULLIF NUMERIC
ORDER
POSITION PRECISION
SESSION_USER
TABLE THEN TRANSACTION TRUE
USER
WHEN
```

Las siguientes palabras reservadas de Postgres también son palabras reservadas de SQL92 o SQL3 :

```
ADD ALL ALTER AND ANY AS ASC
BEGIN BETWEEN BOTH BY
CASCADE CAST CHAR CHARACTER CHECK CLOSE
  COLLATE COLUMN COMMIT CONSTRAINT CREATE
  CURRENT_DATE CURRENT_TIME CURRENT_TIMESTAMP
  CURSOR
DECLARE DEFAULT DELETE DESC DISTINCT DROP
EXECUTE EXISTS EXTRACT
FETCH FLOAT FOR FROM FULL
GRANT
HAVING
IN INNER INSERT INTERVAL INTO IS
JOIN
LEADING LEFT LIKE LOCAL
NAMES NATIONAL NATURAL NCHAR NO NOT NULL
ON OR OUTER
PARTIAL PRIMARY PRIVILEGES PROCEDURE PUBLIC
REFERENCES REVOKE RIGHT ROLLBACK
SELECT SET SUBSTRING
TO TRAILING TRIM
UNION UNIQUE UPDATE USING
VALUES VARCHAR VARYING VIEW
WHERE WITH WORK
```

Las siguientes palabras reservadas de SQL92 no son palabras clave reservadas de Postgres pero si se usan como nombre de función se traducen siempre por la función length:

```
CHAR_LENGTH CHARACTER_LENGTH
```


Las siguientes palabras reservadas de SQL92 o SQL3 no son palabras clave reservadas de Postgres pero si se usan como nombre de tipo se traducen siempre en un tipo alternativo/nativo:

```
BOOLEAN DOUBLE FLOAT INT INTEGER INTERVAL REAL SMALLINT
```

Las siguientes palabras clave reservadas tanto de SQL92 o SQL3 no son palabras clave en Postgres. Esto hace que su uso no sea valido en Postgres en el momento de la escritura (v6.5) pero serán palabras reservadas en el futuro:

Nota: Algunas de estas palabras clave representan funciones en SQL92. Estas funciones están definidas en Postgres, pero el interprete no considera los nombre como palabras clave y las permite en otros contextos.

```
ALLOCATE ARE ASSERTION AT AUTHORIZATION AVG
BIT BIT_LENGTH
CASCADED CATALOG COLLATION CONNECT CONNECTION
CONTINUE CONVERT CORRESPONDING COUNT
DATE DEALLOCATE DEC DESCRIBE DESCRIPTOR
DIAGNOSTICS DISCONNECT DOMAIN
ESCAPE EXCEPT EXCEPTION EXEC EXTERNAL
FIRST FOUND
GET GO GOTO
IDENTITY INDICATOR INPUT INTERSECT
LAST LOWER
MAX MIN MODULE
OCTET_LENGTH OPEN OUTPUT OVERLAPS
PREPARE PRESERVE
ROWS
SCHEMA SECTION SESSION SIZE SOME
SQL SQLCODE SQLERROR SQLSTATE SUM SYSTEM_USER
TEMPORARY TRANSLATE TRANSLATION
UNKNOWN UPPER USAGE
VALUE
WHENEVER WRITE
```

Palabras clave no-reservadas

SQL92 y SQL3 tienen *Palabras clave no-reservadas* which have a prescribed meaning in the language but which are also allowed as identifiers. Postgres que tienen un significado preestablecida en el lenguaje pero también se puede utilizar como identificadores. Postgres tiene palabras clave adicionales con la misma restricción de uso. En particular, estas palabras clave se pueden usar como nombre de columnas o tablas.

Las siguientes palabras clave no-reservadas de Postgres no son palabras clave no-reservadas de SQL92 ni SQL3 :

```
ACCESS AFTER AGGREGATE
BACKWARD BEFORE
CACHE CREATEDB CREATEUSER CYCLE
DATABASE DELIMITERS
EACH ENCODING EXCLUSIVE
FORWARD FUNCTION
```

```

HANDLER
INCREMENT INDEX INHERITS INSENSITIVE INSTEAD ISNULL
LANCOMPILER LOCATION
MAXVALUE MINVALUE MODE
NOCREATEDB NOCREATEUSER NOTHING NOTNULL
OIDS OPERATOR
PASSWORD PROCEDURAL
RECIPE RENAME RETURNS ROW RULE
SEQUENCE SERIAL SHARE START STATEMENT STDIN STDOUT
TRUSTED
VALID VERSION

```

Las siguientes palabras clave no-reservadas de Postgres son palabras clave reservadas de SQL92 o SQL3 :

```

ABSOLUTE ACTION
CONSTRAINTS
DAY DEFERRABLE DEFERRED
HOUR
IMMEDIATE INITIALLY INSENSITIVE ISOLATION
KEY
LANGUAGE LEVEL
MATCH MINUTE MONTH
NEXT
OF ONLY OPTION
PENDANT PRIOR PRIVILEGES
READ RELATIVE RESTRICT
SCROLL SECOND
TIME TIMESTAMP TIMEZONE_HOUR TIMEZONE_MINUTE TRIGGER
YEAR
ZONE

```

Las siguientes palabras clave no-reservadas de Postgres también son palabras clave no-reservadas de SQL92 o SQL3 :

```

COMMITTED SERIALIZABLE TYPE

```

Las siguientes palabras clave no-reservadas tanto de SQL92 o SQL3 no son palabras clave de ninguna clase en Postgres:

```

ADA
C CATALOG_NAME CHARACTER_SET_CATALOG CHARACTER_SET_NAME
  CHARACTER_SET_SCHEMA CLASS_ORIGIN COBOL COLLATION_CATALOG
  COLLATION_NAME COLLATION_SCHEMA COLUMN_NAME
  COMMAND_FUNCTION CONDITION_NUMBER
  CONNECTION_NAME CONSTRAINT_CATALOG CONSTRAINT_NAME
  CONSTRAINT_SCHEMA CURSOR_NAME
DATA DATE_TIME_INTERVAL_CODE DATE_TIME_INTERVAL_PRECISION
  DYNAMIC_FUNCTION
FORTRAN
LENGTH
MESSAGE_LENGTH MESSAGE_OCTET_LENGTH MORE MUMPS
NAME NULLABLE NUMBER
PAD PASCAL PLI

```

```

REPEATABLE RETURNED_LENGTH RETURNED_OCTET_LENGTH
RETURNED_SQLSTATE ROW_COUNT
SCALE SCHEMA_NAME SERVER_NAME SPACE SUBCLASS_ORIGIN
TABLE_NAME
UNCOMMITTED UNNAMED

```

Comentarios

Un *Comentario* es una secuencia arbitraria de caracteres precedido por un doble guión hasta final de línea. También está soportado la doble barra pe.:

```

- This is a standard SQL comment
// And this is another supported comment style, like C++

```

También soportamos el bloque de comentarios al estilo C pe.:

```

/* multi
   line
   comment
*/

```

Nombres

El nombre en SQL es una secuencia de caracteres alfanuméricos menor que NAME-
DATALEN, comenzando por un carácter alfanumérico. Por defecto, NAMEDATA-
LEN esta definido a 32, pero en el momento que montar el sistema, NAMEDATA-
LEN puede cambiarse cambiando el #define en src/backend/include/postgres.h.
Subrayado ("_") esta considerado como un carácter alfabético.

En algunos contextos, los nombre pueden contener otros caracteres si están entre-
comillados por doble comillas. Por ejemplo, nombres de tablas o campos pueden
contener otros caracteres no validos como los espacios, ampersand (&), etc. usando
esta técnica.

Constantes

Hay tres *tres tipos implícitos de constantes* usadas Postgres: cadenas, enteros y números
de coma flotante. Las Constantes también pueden ser especificadas con un tipo ex-
plícito, el cual puede una representación más adecuada y una manejo más eficiente.
Las constantes implícitas se describen más abajo; las constantes explícitas se tratarán
más adelante.

Constantes tipo Cadenas

Las cadenas son secuencias arbitrarias de caracteres ASCII limitadas por comillas sim-
ples (" ' ", pe. 'Esto es una cadena')SQL92 permite que las comillas simples pue-
dan estar incluidos en una cadena tecleando dos comillas simples adyacentes (pe.

'Dianne"s horse'). En Postgres las comillas simples deben estar precedidas por una contra barra ("\"), pe.. 'Dianne\"s horse'). para incluir una contra barra en una constante de tipo cadena, teclear dos contra barras. Los caracteres no imprimibles también deben incluir en la cadena precedidos de una contra barra (pe '\tab').

Constantes tipo Entero

Las constantes tipo enteros son una colección de dígitos ASCII sin punto decimal. Los rangos de valores validos van desde -2147483648 al +2147483647. Esto variará dependiendo del sistema operativo y la máquina host.

Destacar que el entero más largo puede ser especificado para int8 utilizando una notación de cadena SQL92 o una notación del tipo Postgres:

```
int8 '4000000000' - string style
'4000000000'::int8 - Postgres (historical) style
```

Constantes tipo Punto Flotante

Floating point constants consta de una parte entera , un punto decimal, y una parte decimal o la notación científica con el siguiente formato:

```
{dig} . {dig} [e [+ -] {dig}]
```

Donde *dig* is one or more digits. You must include at least one *dig* es uno o más dígitos. Usted puede incluir como mínimo un dig después del periodo y después de [+ -] si esas opciones. Un exponente sin mantisa tiene una mantisa insertada a 1. No debe haber ningún carácter extra incluido en la cadena.

Una constante de tipo punto flotante es del tipo float8. Para float4 se puede especificar explícitamente usando la notación de cadena de SQL92o notación de tipo Postgres:

```
float4 '1.23' - string style
'1.23'::float4 - Postgres (historical) style
```

Constantes Postgres de tipos definido por el usuario

Una constante de un tipo *arbitrario* puede ser usando utilizando alguna de las siguientes notaciones:

```
type 'string'
'string'::type
CAST 'string' AS type
```

El valor de dentro de la cadena se pasa como entrada a rutina de conversión para el tipo llamado *type*. El resultado es una constante del tipo indicado. La tipología puede omitirse si no hay ambigüedad sobre el tipo de constate que debe ser, en este caso este está automáticamente forzado.

Constantes de tipo Array

Las constantes de tipo *Array* de cualquier tipo Postgres, incluidos otras arrays, constantes de cadena, etc. El formato general de cualquier constante array es el siguiente:

```
{val1delimval2delim}
```

Donde *delim* es el delimitador para el tipo almacenado en la clase `pg_type`. (Para los tipos preconstruidos, es el carácter coma. Un ejemplo de constante de tipo array es:

```
{{1,2,3},{4,5,6},{7,8,9}}
```

Esta constante es de dos dimensiones, una array de 3 por 3 consiste en tres subarrays de enteros.

Un elemento de una array individual puede y debe estar entre marcas delimitadoras siempre que sea posible para evitar problemas de ambigüedad con respecto a espacios en blanco iniciales.

Campos y Columnas

Campos

Un *Campo* es cualquier atributo de una clase dada o uno de lo siguiente:

`oid`

el identificador único de la instancia que añade Postgres a todas las instancias automáticamente. Los Oids no son reutilizable y tienen una longitud de 32 bits.

`xmin`

El identificador de la transacción insertada.

`xmax`

El identificador de la transacción borrada.

`cmin`

El identificador del comando dentro de la transacción.

`cmax`

El identificador del comando borrado.

Para más información de estos campos consultar *Stonebraker, Hanson, Hong, 1987*. El tiempo está representado internamente como una instancia del tipo dato `abstime`. Los identificadores de las transacciones y comandos son de 32 bits. Las transacciones se asignan secuencialmente empezando por 512.

Columnas

Una *columna* se construye de esta forma:

```
instance{.composite_field}.field '['number']'
```

Una *instance* identifica una clase concreta y podemos entenderla como un particularización de las instancias de esta clase. Cada nombre de variable es una variable instancia, un sustituto de la clase definida por el significado de la cláusula FROM, o la palabra clave NEW o CURRENT. NEW y CURRENT sólo pueden aparecer en una tramo de la acción de la regla, mientras otras variables de instancia pueden usarse en cualquier declaración SQL. *composite_field* un campo de uno de los tipos compuestos de Postgres, mientras que los sucesivos campos direccionan los atributos de la clase/es que evalúa los campo compuesto. Finalmente *field* es un campo normal (tipo base) de la última clase/s direccionada. Si *field* es de tipo array, entonces el designador opcional *number* indica el elemento específico del array. Si no se indica el número, entonces se devolverán todos los elementos del array.

Operadores

Cualquier operador predefinido o definido por el usuario puede usarse en SQL. Para la lista de operadores predefinidos consultar *Operadores*. Para la lista de los operadores definidos por el usuario consultar su administrador de sistema o ejecuta la consulta sobre la clase *pg_operator* class. Los paréntesis pueden usarse para agrupar arbitrariamente los operadores en expresiones.

Expresiones

SQL92 permite *expresiones* para transformar datos en tablas. Las expresiones pueden contener operadores (ver *Operadores* para más detalles) y funciones (*Funciones* tiene más información).

Una expresión es una de las siguientes:

```
( a_expr )
constantes
atributos
a_expr binary_operator a_expr
a_expr right_unary_operator
left_unary_operator a_expr
parametros
expresiones funcionales
expresiones de agregación
```

Nosotros ya hemos hablado de las constantes y atributos. Las tres clases de expresiones de operadores son respectivamente operadores binarios (infijo), unarios por la derecha (sufijo) y unarios por la izquierda (prefijo). Las siguientes secciones hablan de la distintas opciones.

Parámetros

Un *Parámetro* se usa para indicar un parámetro en una función SQL. Típicamente este es el uso de la definición de la declaración de la función SQL. La forma con paréntesis

es:

```
$number
```

Por ejemplo, consideramos la definición de la función, dept, como

```
CREATE FUNCTION dept (name)
RETURNS dept
AS 'select * from
    dept where name=$1'
LANGUAGE 'sql';
```

Expresiones Funcionales

Una *Expresión Funcional* es el nombre de una función legal SQL, seguida por sus lista de argumentos entre paréntesis:

```
function (a_expr [, a_expr ... ] )
```

Por ejemplo, el siguiente calcula la raíz cuadrada del salario de un empleado:

```
sqrt(emp.salary)
```

Expresiones de Agregación

Una *expresión de agregación* representa la aplicación de una función de agregación a través de las filas seleccionadas por la consulta. Una función de agregación reduce múltiples entradas a un solo valor de salida, como la suma o la media de la entrada. La sintaxis de la expresión de agregación es la siguiente:

```
aggregate_name (expression)
aggregate_name (ALL expression)
aggregate_name (DISTINCT expression)
aggregate_name ( * )
```

Donde *aggregate_name* es la agregación previamente definida, y *expresiones* cualquier expresión que no contenga a su vez ninguna expresión de agregación.

La primera forma de expresión de agregación llama a la agregación a través de todas las filas de entrada la expresión devuelve un valor no nulo. La segunda forma es similar a la primera, pero ALL es por defecto. La tercera forma llama a la agregación para todas las filas de entrada con valores distintos entre si y no nulo. La última forma llama a la agregación para cada una de las filas de entrada sean con valor nulo o no; si no se especifica un valor específico de entrada, generalmente sólo es útil para la agregación count().

Por ejemplo, count(*) devuelve el número total de filas de entrada; count(f1) devuelve

el número de filas de entrada donde *f1* no es nulo; `count(distinct f1)` devuelve el número de distintos valores no nulos de *f1*.

Lista Objetivo

Una *Lista Objetivo* es una lista de uno o más elementos separados por comas y entre paréntesis, cada una debe ser de la forma:

```
a_expr [ AS result_attnname ]
```

Donde *result_attnname* es el nombre del atributo que ha sido creado (o el nombre del atributo que ya existía en el caso de una sentencia de actualización.) Si *result_attnname* no está presente, entonces *a_expr* debe contener sólo un nombre de atributo el cual se asumirá como el nombre del campo resultado. Sólo se usa el nombrado por defecto en Postgres si *a_expr* es un atributo.

Calificadores

Un *calificador* consiste en cualquier número de cláusulas conectadas por operadores lógicos:

```
NOT
AND
OR
```

Una cláusula es un *a_expr* que se evalúa como un `boolean` sobre el conjunto de instancias.

Lista From

La *Lista From* es una lista de *expresiones from*, separadas por comas. Cada "expresión from" es de esta forma:

```
[ class_reference ] instance_variable
{, [ class_ref ] instance_variable... }
```

Donde *class_reference* es de la forma

```
class_name [ * ]
```

La "expresión from" define una o más variables instancia sobre el rango que la clase indica en *class_reference*. Uno también puede requerir la variable instancia sobre el rango de todas la clases que están por debajo de las clases indicadas en la jerarquía de herencia especificando el designador asterisco (*).

Capítulo 3. Data Types

Describes the built-in data types available in Postgres.

Postgres has a rich set of native data types available to users. Users may add new types to Postgres using the **DEFINE TYPE** command described elsewhere.

In the context of data types, the following sections will discuss SQL standards compliance, porting issues, and usage. Some Postgres types correspond directly to SQL92-compatible types. In other cases, data types defined by SQL92 syntax are mapped directly into native Postgres types. Many of the built-in types have obvious external formats. However, several types are either unique to Postgres, such as open and closed paths, or have several possibilities for formats, such as the date and time types.

Tabla 3-1. Postgres Data Types

Postgres Type	SQL92 or SQL3 Type	Description
bool	boolean	logical boolean (true/ false)
box		rectangular box in 2D plane
char(n)	character(n)	fixed-length character string
cidr		IP version 4 network or host address
circle		circle in 2D plane
date	date	calendar date without time of day
decimal	decimal(p,s)	exact numeric for p <= 9, s = 0
float4/8	float(p)	floating-point number with precision p
float8	real, double precision	double-precision floating-point number
inet		IP version 4 network or host address
int2	smallint	signed two-byte integer
int4	int, integer	signed 4-byte integer
int8		signed 8-byte integer
line		infinite line in 2D plane
lseg		line segment in 2D plane
money	decimal(9,2)	US-style currency
numeric	numeric(p,s)	exact numeric for p == 9, s = 0
path		open and closed geometric path in 2D plane

Postgres Type	SQL92 or SQL3 Type	Description
point		geometric point in 2D plane
polygon		closed geometric path in 2D plane
serial		unique id for indexing and cross-reference
time	time	time of day
timespan	interval	general-use time span
timestamp	timestamp with time zone	date/time
varchar(n)	character varying(n)	variable-length character string

Nota: The cidr and inet types are designed to handle any IP type but only ipv4 is handled in the current implementation. Everything here that talks about ipv4 will apply to ipv6 in a future release.

Tabla 3-2. Postgres Function Constants

Postgres Function	SQL92 Constant	Description
getpgusername()	current_user	user name in current session
date('now')	current_date	date of current transaction
time('now')	current_time	time of current transaction
timestamp('now')	current_timestamp	date and time of current transaction

Postgres has features at the forefront of ORDBMS development. In addition to SQL3 conformance, substantial portions of SQL92 are also supported. Although we strive for SQL92 compliance, there are some aspects of the standard which are ill considered and which should not live through subsequent standards. Postgres will not make great efforts to conform to these features; however, these tend to apply in little-used or obscure cases, and a typical user is not likely to run into them.

Most of the input and output functions corresponding to the base types (e.g., integers and floating point numbers) do some error-checking. Some of the operators and functions (e.g., addition and multiplication) do not perform run-time error-checking in the interests of improving execution speed. On some systems, for example, the numeric operators for some data types may silently underflow or overflow.

Note that some of the input and output functions are not invertible. That is, the result of an output function may lose precision when compared to the original input.

Nota: The original Postgres v4.2 code received from Berkeley rounded all double precision floating point results to six digits for output. Starting with v6.1, floating point numbers are allowed to retain most of the intrinsic precision of the type (typically 15 digits for doubles, 6 digits for 4-byte floats). Other types with underlying floating point fields (e.g. geometric types) carry similar precision.

Numeric Types

Numeric types consist of two- and four-byte integers and four- and eight-byte floating point numbers.

Tabla 3-3. Postgres Numeric Types

Numeric Type	Storage	Description	Range
decimal	variable	User-specified precision	no limit
float4	4 bytes	Variable-precision	6 decimal places
float8	8 bytes	Variable-precision	15 decimal places
int2	2 bytes	Fixed-precision	-32768 to +32767
int4	4 bytes	Usual choice for fixed-precision	-2147483648 to +2147483647
int8	8 bytes	Very large range fixed-precision	+/- > 18 decimal places
numeric	variable	User-specified precision	no limit
serial	4 bytes	Identifier or cross-reference	0 to +2147483647

The numeric types have a full set of corresponding arithmetic operators and functions. Refer to *Operadores numéricos* and *Funciones Matemáticas* for more information.

The int8 type may not be available on all platforms since it relies on compiler support for this.

The Serial Type

The serial type is a special-case type constructed by Postgres from other existing components. It is typically used to create unique identifiers for table entries. In the current implementation, specifying

```
CREATE TABLE tablename (colname SERIAL);
```

is equivalent to specifying:

```
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename
```

```
(colname INT4 DEFAULT nextval('tablename_colname_seq');
CREATE UNIQUE INDEX tablename_colname_key on tablename (colname);
```

Atención

The implicit sequence created for the serial type will *not* be automatically removed when the table is dropped.

Implicit sequences supporting the serial are not automatically dropped when a table containing a serial type is dropped. So, the following commands executed in order will likely fail:

```
CREATE TABLE tablename (colname SERIAL);
DROP TABLE tablename;
CREATE TABLE tablename (colname SERIAL);
```

The sequence will remain in the database until explicitly dropped using **DROP SEQUENCE**.

Monetary Type

Obsolete Type: The money is now obsolete. Use numeric or decimal instead.

The money type supports US-style currency with fixed decimal point representation. If Postgres is compiled with USE_LOCALE then the money type should use the monetary conventions defined for *locale(7)*.

Tabla 3-4. Postgres Monetary Types

Monetary Type	Storage	Description	Range
money	4 bytes	Fixed-precision	-21474836.48 to +21474836.47

numeric will replace the money type, and should be preferred.

Character Types

SQL92 defines two primary character types: char and varchar. Postgres supports these types, in addition to the more general text type, which unlike varchar does not require an upper limit to be declared on the size of the field.

Tabla 3-5. Postgres Character Types

Character Type	Storage	Recommendation	Description
----------------	---------	----------------	-------------

Character Type	Storage	Recommendation	Description
char	1 byte	SQL92-compatible	Single character
char(n)	(4+n) bytes	SQL92-compatible	Fixed-length blank padded
text	(4+x) bytes	Best choice	Variable-length
varchar(n)	(4+n) bytes	SQL92-compatible	Variable-length with limit

There is one other fixed-length character type. The name type only has one purpose and that is to provide Postgres with a special type to use for internal names. It is not intended for use by the general user. It's length is currently defined as 32 chars but should be reference using NAMEDATALEN. This is set at compile time and may change in a future release.

Tabla 3-6. Postgres Specialty Character Type

Character Type	Storage	Description
name	32 bytes	Thirty-two character internal type

Date/Time Types

PostgreSQL supports the full set of SQL date and time types.

Tabla 3-7. PostgreSQL Date/Time Types

Type	Description	Storage	Earliest	Latest	Resolution
timestamp	for data containing both date and time	8 bytes	4713 BC	AD 1465001	1 mic
interval	for time intervals	12 bytes	-178000000 years	178000000 years	1 mir
date	for data containing only dates	4 bytes	4713 BC	32767 AD	1 day
time	for data containing only times of the day	4 bytes	00:00:00.00	23:59:59.99	1 mic

Nota: To ensure compatibility to earlier versions of PostgreSQL we also continue to provide datetime (equivalent to timestamp) and timespan (equivalent to interval). The types abstime and reltime are lower precision types which are used internally. You are discouraged from using any of these types in new applications and move any old ones over when appropriate. Any or all of these type might disappear in a future release.

Date/Time Input

Date and time input is accepted in almost any reasonable format, including ISO-compatible, SQL-compatible, traditional Postgres, and others. The ordering of month and day in date input can be ambiguous, therefore a setting exists, to specify how it should be interpreted. The command `SET DateStyle TO 'US'` or `SET DateStyle TO 'NonEuropean'` specifies the variant “month before day”, the command `SET DateStyle TO 'European'` sets the variant “day before month”. The former is the default.

See *ayuda de fecha/hora* for the exact parsing rules of date/time input and for the recognized time zones.

Remember that any date or time input needs to be enclosed into single quotes, like text strings.

date

The following are possible inputs for the date type.

Tabla 3-8. PostgreSQL Date Input

Example	Description
January 8, 1999	Unambiguous
1999-01-08	ISO-8601 format, preferred
1/8/1999	US; read as August 1 in European mode
8/1/1999	European; read as August 1 in US mode
1/18/1999	US; read as January 18 in any mode
1999.008	Year and day of year
19990108	ISO-8601 year, month, day
990108	ISO-8601 year, month, day
1999.008	Year and day of year
99008	Year and day of year
January 8, 99 BC	Year 99 before the common era

Tabla 3-9. PostgreSQL Month Abbreviations

Month	Abbreviations
April	Apr
August	Aug
December	Dec
February	Feb
January	Jan
July	Jul
June	Jun

Month	Abbreviations
March	Mar
November	Nov
October	Oct
September	Sep, Sept

Nota: The month `May` has no explicit abbreviation, for obvious reasons.

Tabla 3-10. PostgreSQL Day of Week Abbreviations

Day	Abbreviation
Sunday	Sun
Monday	Mon
Tuesday	Tue, Tues
Wednesday	Wed, Weds
Thursday	Thu, Thur, Thurs
Friday	Fri
Saturday	Sat

time

The following are valid time inputs.

Tabla 3-11. PostgreSQL Time Input

Example	Description
04:05:06.789	ISO-8601
04:05:06	ISO-8601
04:05	ISO-8601
040506	ISO-8601
04:05 AM	Same as 04:05; AM does not affect value
04:05 PM	Same as 16:05; input hour must be <= 12
z	Same as 00:00:00
zulu	Same as 00:00:00
allballs	Same as 00:00:00

timestamp

Valid input for the timestamp type consists of a concatenation of a date and a time, followed by an optional AD or BC, followed by an optional time zone. (See below.) Thus

```
1999-01-08 04:05:06 -8:00
```

is a valid timestamp value, which is ISO-compliant. In addition, the wide-spread format

```
January 8 04:05:06 1999 PST
```

is supported.

Tabla 3-12. PostgreSQL Time Zone Input

Time Zone	Description
PST	Pacific Standard Time
-8:00	ISO-8601 offset for PST
-800	ISO-8601 offset for PST
-8	ISO-8601 offset for PST

interval

intervals can be specified with the following syntax:

```
Quantity Unit [Quantity Unit...] [Direction]
@ Quantity Unit [Direction]
```

where: Quantity is ..., -1, 0, 1, 2, ...; Unit is second, minute, hour, day, week, month, year, decade, century, millenium, or abbreviations or plurals of these units; Direction can be ago or empty.

Special values

The following SQL-compatible functions can be used as date or time input for the corresponding datatype: `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`.

PostgreSQL also supports several special constants for convenience.

Tabla 3-13. PostgreSQL Special Date/Time Constants

Constant	Description
current	Current transaction time, deferred
epoch	1970-01-01 00:00:00+00 (Unix system time zero)
infinity	Later than other valid times
-infinity	Earlier than other valid times

Constant	Description
invalid	Illegal entry
now	Current transaction time
today	Midnight today
tomorrow	Midnight tomorrow
yesterday	Midnight yesterday

'now' is resolved when the value is inserted, 'current' is resolved everytime the value is retrieved. So you probably want to use 'now' in most applications. (Of course you *really* want to use CURRENT_TIMESTAMP, which is equivalent to 'now'.)

Date/Time Output

Output formats can be set to one of the four styles ISO-8601, SQL (Ingres), traditional Postgres, and German, using the **SET DateStyle**. The default is the ISO format.

Tabla 3-14. PostgreSQL Date/Time Output Styles

Style Specification	Description	Example
'ISO'	ISO-8601 standard	1997-12-17 07:37:16-08
'SQL'	Traditional style	12/17/1997 07:37:16.00 PST
'Postgres'	Original style	Wed Dec 17 07:37:16 1997 PST
'German'	Regional style	17.12.1997 07:37:16.00 PST

The output of the date and time styles is of course only the date or time part in accordance with the above examples

The SQL style has European and non-European (US) variants, which determines whether month follows day or vica versa. (See also above at Date/Time Input, how this setting affects interpretation of input values.)

Tabla 3-15. PostgreSQL Date Order Conventions

Style Specification	Example	
European	17/12/1997 15:37:16.00 MET	
US	12/17/1997 07:37:16.00 PST	

interval output looks like the input format, expect that units like week or century are converted to years and days. In ISO mode the output looks like

```
[ Quantity Units [ ... ] ] [ Days ] Hours:Minutes [ ago ]
```

There are several ways to affect the appearance of date/time types:

- The PGDATESTYLE environment variable used by the backend directly on postmaster startup.
- The PGDATESTYLE environment variable used by the frontend libpq on session startup.
- **SET DATESTYLE** SQL command.

Time Zones

PostgreSQL endeavors to be compatible with SQL92 definitions for typical usage. However, the SQL92 standard has an odd mix of date and time types and capabilities. Two obvious problems are:

- Although the date type does not have an associated time zone, the time type can or does.
- The default time zone is specified as a constant integer offset from GMT/UTC.

Time zones in the real world can have no meaning unless associated with a date as well as a time since the offset may vary through the year with daylight savings time boundaries.

To address these difficulties, PostgreSQL associates time zones only with date and time types which contain both date and time, and assumes local time for any type containing only date or time. Further, time zone support is derived from the underlying operating system time zone capabilities, and hence can handle daylight savings time and other expected behavior.

PostgreSQL obtains time zone support from the underlying operating system for dates between 1902 and 2038 (near the typical date limits for Unix-style systems). Outside of this range, all dates are assumed to be specified and used in Universal Coordinated Time (UTC).

All dates and times are stored internally in Universal UTC, alternately known as Greenwich Mean Time (GMT). Times are converted to local time on the database server before being sent to the client frontend, hence by default are in the server time zone.

There are several ways to affect the time zone behavior:

- The TZ environment variable used by the backend directly on postmaster startup as the default time zone.
- The PGTZ environment variable set at the client used by libpq to send time zone information to the backend upon connection.
- The SQL command **SET TIME ZONE** sets the time zone for the session.

If an invalid time zone is specified, the time zone becomes GMT (on most systems anyway).

Nota: If the compiler option `USE_AUSTRALIAN_RULES` is set then `EST` refers to Australia Eastern Std Time, which has an offset of +10:00 hours from UTC.

Internals

PostgreSQL uses Julian dates for all date/time calculations. They have the nice property of correctly predicting/calculating any date more recent than 4713BC to far into the future, using the assumption that the length of the year is 365.2425 days.

Date conventions before the 19th century make for interesting reading, but are not consistant enough to warrant coding into a date/time handler.

Boolean Type

Postgres supports bool as the SQL3 boolean type. bool can have one of only two states: 'true' or 'false'. A third state, 'unknown', is not implemented and is not suggested in SQL3; NULL is an effective substitute. bool can be used in any boolean expression, and boolean expressions always evaluate to a result compatible with this type.

bool uses 1 byte of storage.

Tabla 3-16. Postgres Boolean Type

State	Output	Input
True	't'	TRUE, 't', 'true', 'y', 'yes', '1'
False	'f'	FALSE, 'f', 'false', 'n', 'no', '0'

Geometric Types

Geometric types represent two-dimensional spatial objects. The most fundamental type, the point, forms the basis for all of the other types.

Tabla 3-17. Postgres Geometric Types

Geometric Type	Storage	Representation	Description
point	16 bytes	(x,y)	Point in space
line	32 bytes	((x1,y1),(x2,y2))	Infinite line
lseg	32 bytes	((x1,y1),(x2,y2))	Finite line segment
box	32 bytes	((x1,y1),(x2,y2))	Rectangular box
path	4+32n bytes	((x1,y1),...)	Closed path (similar to polygon)
path	4+32n bytes	[(x1,y1),...]	Open path
polygon	4+32n bytes	((x1,y1),...)	Polygon (similar to closed path)
circle	24 bytes	<(x,y),r>	Circle (center and radius)

A rich set of functions and operators is available to perform various geometric operations such as scaling, translation, rotation, and determining intersections.

Point

Points are the fundamental two-dimensional building block for geometric types.

point is specified using the following syntax:

```
( x , y )
  x , y
where
  x is the x-axis coordinate as a floating point number
  y is the y-axis coordinate as a floating point number
```

Line Segment

Line segments (lseg) are represented by pairs of points.

lseg is specified using the following syntax:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1      ,      x2 , y2
where
  (x1,y1) and (x2,y2) are the endpoints of the segment
```

Box

Boxes are represented by pairs of points which are opposite corners of the box.

box is specified using the following syntax:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1      ,      x2 , y2
where
  (x1,y1) and (x2,y2) are opposite corners
```

Boxes are output using the first syntax. The corners are reordered on input to store the lower left corner first and the upper right corner last. Other corners of the box can be entered, but the lower left and upper right corners are determined from the input and stored.

Path

Paths are represented by connected sets of points. Paths can be "open", where the first and last points in the set are not connected, and "closed", where the first and last point are connected. Functions `popen(p)` and `pclose(p)` are supplied to force a path to be open or closed, and functions `isopen(p)` and `isclosed(p)` are supplied to select either type in a query.

path is specified using the following syntax:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
( x1 , y1 ) , ... , ( xn , yn )
( x1 , y1 , ... , xn , yn )
  x1 , y1 , ... , xn , yn
where
  (x1,y1),...,(xn,yn) are points 1 through n
  a leading "[" indicates an open path
  a leading "(" indicates a closed path
```

Paths are output using the first syntax. Note that Postgres versions prior to v6.1 used a format for paths which had a single leading parenthesis, a "closed" flag, an integer count of the number of points, then the list of points followed by a closing parenthesis. The built-in function `upgradepath` is supplied to convert paths dumped and reloaded from pre-v6.1 databases.

Polygon

Polygons are represented by sets of points. Polygons should probably be considered equivalent to closed paths, but are stored differently and have their own set of support routines.

polygon is specified using the following syntax:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1 , ... , xn , yn )
    x1 , y1 , ... , xn , yn
where
  (x1,y1),...,(xn,yn) are points 1 through n
```

Polygons are output using the first syntax. Note that Postgres versions prior to v6.1 used a format for polygons which had a single leading parenthesis, the list of x-axis coordinates, the list of y-axis coordinates, followed by a closing parenthesis. The built-in function `upgradepoly` is supplied to convert polygons dumped and reloaded from pre-v6.1 databases.

Circle

Circles are represented by a center point and a radius.

circle is specified using the following syntax:

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y , r
```

where
 (x,y) is the center of the circle
 r is the radius of the circle

Circles are output using the first syntax.

IP Version 4 Networks and Host Addresses

The `cidr` type stores networks specified in CIDR (Classless Inter-Domain Routing) notation. The `inet` type stores hosts and networks in CIDR notation using a simple variation in representation to represent simple host TCP/IP addresses.

Tabla 3-18. PostgresIP Version 4 Types

IPv4 Type	Storage	Description	Range
<code>cidr</code>	variable	CIDR networks	Valid IPv4 CIDR blocks
<code>inet</code>	variable	nets and hosts	Valid IPv4 CIDR blocks

CIDR

The `cidr` type holds a CIDR network. The format for specifying classless networks is `x.x.x.x/y` where `x.x.x.x` is the network and `/y` is the number of bits in the netmask. If `/y` omitted, it is calculated using assumptions from the older classfull naming system except that it is extended to include at least all of the octets in the input.

Here are some examples:

Tabla 3-19. PostgresIP Types Examples

CIDR Input	CIDR Displayed
192.168.1	192.168.1/24
192.168	192.168.0/24
128.1	128.1/16
128	128.0/16
128.1.2	128.1.2/24
10.1.2	10.1.2/24
10.1	10.1/16
10	10/8

inet

The `inet` type is designed to hold, in one field, all of the information about a host including the CIDR-style subnet that it is in. Note that if you want to store proper CIDR networks, you should use the `cidr` type. The `inet` type is similar to the `cidr` type except that the bits in the host part can be non-zero. Functions exist to extract the various elements of the field.

The input format for this function is `x.x.x.x/y` where `x.x.x.x` is an internet host and `y` is the number of bits in the netmask. If the `/y` part is left off, it is treated as `/32`. On output, the `/y` part is not printed if it is `/32`. This allows the type to be used as a straight host type by just leaving off the bits part.

Capítulo 4. Operadores

Describe los operadores propios disponibles en Postgres.

Postgres proporciona un gran número de tipos de operadores. Estos operadores están declarados en el catálogo del sistema `pg_operator`. Cada entrada en `pg_operator` incluye el nombre del procedimiento que implementa el operador y las clases OIDs de los tipos de entrada y salida.

Para ver todas las variantes del operador de concatenación de strings “||” pruebe,

```
SELECT oprleft, oprright, oprresult, oprcode
FROM pg_operator WHERE oprname = '||';

oprleft|oprright|oprresult|oprcode
-----+-----+-----+-----
      25|       25|       25|textcat
    1042|    1042|    1042|textcat
    1043|    1043|    1043|textcat
(3 rows)
```

Los usuarios pueden invocar a los operadores utilizando el nombre del operador de este modo:

```
select * from emp where salary < 40000;
```

De otra manera, los usuarios pueden llamar a las funciones que implementan los operadores directamente. En este caso la pregunta anterior se haría así:

```
select * from emp where int4lt(salary, 40000);
```

`psql` tiene un comando (`\dd`) para mostrar estos operadores.

Lexical Precedence

Los operadores tienen una precedencia que está codificada dentro del parser. La mayoría de los operadores tienen la misma precedencia y son asociativos. Esto puede acarrear comportamientos poco intuitivos. Por ejemplo, los operadores booleanos “<” y “>” tienen una precedencia diferente que los operadores booleanos “<=” y “>=”.

Tabla 4-1. Orden de operadores (precedencia decreciente)

Elemento	Precedencia	Descripción
UNION	izquierda	constructor SQL de se
::		convertor de tipos de
[]	izquierda	delimitadores de array

Elemento	Precedencia	Descripción
.	izquierda	delimitadores de tabla
-	derecha	menos unario
;	izquierda	terminación de declaración
:	derecha	exponenciación
	izquierda	comienzo de intervalo
* / %	izquierda	multiplicación, división
+ -	izquierda	adición, sustracción
IS		test para TRUE, FALSE
ISNULL		test para NULL
NOTNULL		test para NOT NULL
(todos los demás operadores)		nativos y definidos por el usuario
IN		fijar miembro
BETWEEN		continente
LIKE		concordancia de patrones
< >		desigualdad booleana
=	derecha	igualdad
NOT	derecha	negación
AND	izquierda	intersección lógica
OR	izquierda	unión lógica

Operadores generales

Los operadores mostrados aquí están definidos para un número de tipos de datos nativos, que van desde los tipos numéricos hasta los tipos date/time.

Tabla 4-2. Postgres Operators

Operador	Descripción	Utilización
<	Menor que?	1 < 2
<=	Menor o igual que?	1 <= 2
<>	No igual?	1 <> 2
=	Igual?	1 = 1
>	Mayor que?	2 > 1
>=	Mayor o igual que?	2 >= 1
	Concatena strings	'Postgre' 'SQL'
!=	NOT IN	3 != i
~~	Como	'scrappy,marc,hermit' ~~ '%scrappy%'

Operador	Descripción	Utilización
!~	No como	'bruce' !~ '%al%'
~	Concordancia (regex), sensible a mayusc/minusc	'thomas' ~ '.*thomas.*'
~*	Concordancia (regex), sensible a mayusc/minusc	'thomas' ~* '.*Thomas.*'
!~	No concuerda (regex), sensible a mayusc/minusc	'thomas' !~ '.*Thomas.*'
!~*	No concuerda (regex), sensible a mayusc/minusc	'thomas' !~* '.*vadim.*'

Operadores numéricos

Tabla 4-3. Postgres Operadores numéricos

Operador	Descripción	Utilización
!	Factorial	3 !
!!	Factorial (operador izquierdo)	!! 3
%	Módulo	5 % 4
%	Truncado	% 4.5
*	Multipliación	2 * 3
+	Suma	2 + 3
-	Resta	2 - 3
/	División	4 / 2
:	Exponenciación natural	: 3.0
;	Logaritmo natural	(; 5.0)
@	Valor Absoluto	@ -5.0
^	Exponenciación	2.0 ^ 3.0
/	Raíz cuadrada	/ 25.0
/	Raíz cúbica	/ 27.0

Operadores geométricos

Tabla 4-4. Postgres Operadores geométricos

Operador	Descripción	Utilización
----------	-------------	-------------

Operador	Descripción	Utilización
+	Translación	'((0,0),(1,1))':::box + '(2,0,0)':::punto
-	Translación	'((0,0),(1,1))':::box - '(2,0,0)':::punto
*	Escalado / rotación	'((0,0),(1,1))':::box * '(2,0,0)':::punto
/	Escalado / rotación	'((0,0),(2,2))':::box / '(2,0,0)':::punto
#	Intersección	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'
#	Número de puntos en polígono	# '(1,0),(0,1),(-1,0))'
##	Punto más próximo	'(0,0)':::punto ## '(2,0),(0,2))':::lseg
&&	Se superpone a?	'((0,0),(1,1))':::caja && '((0,0),(2,2))':::caja
&<	Se superpone por la izquierda?	'((0,0),(1,1))':::caja &< '((0,0),(2,2))':::caja
&>	Se superpone por la derecha?	'((0,0),(3,3))':::caja &> '((0,0),(2,2))':::caja
<->	Distancia entre	'((0,0),1)':::círculo <-> '(5,0),1)':::círculo
<<	A la izquierda de?	'((0,0),1)':::círculo << '(5,0),1)':::círculo
<^	Está debajo de?	'((0,0),1)':::círculo <^ '(0,5),1)':::círculo
>>	A la derecha de?	'((5,0),1)':::círculo >> '((0,0),1)':::círculo
>^	Esta encima de?	'((0,5),1)':::círculo >^ '(0,0),1)':::círculo
?#	Interseca o se superpone	'((-1,0),(1,0))':::lseg ?# '((-2,-2),(2,2))':::caja;
?-	Es horizontal?	'(1,0)':::punto ?- '(0,0)':::punto
?-	Es perpendicular?	'((0,0),(0,1))':::lseg ?- '((0,0),(1,0))':::lseg
@-@	Longitud de circunferencia	@-@ '(0,0),(1,0))':::path
?	Es vertical?	'(0,1)':::punto ? '(0,0)':::punto
?	Es paralelo?	'((-1,0),(1,0))':::lseg ? '((-1,2),(1,2))':::lseg
@	Contenido en	'(1,1)':::punto @ '((0,0),2)':::círculo
@@	Centro de	@@ '(0,0),10)':::círculo

Operador	Descripción	Utilización
~=	Parecido a	'((0,0),(1,1))':poligono ~= '((1,1),(0,0))':poligono

Operadores de intervalos de tiempo

El tipo de dato de intervalos de tiempo, `interval`, es un legado de los tipos `date/time` originales y no está tan bien soportado como los tipos más modernos. Hay varios operadores para este tipo.

Tabla 4-5. Postgres Operadores de intervalos de tiempo

Operador	Descripción	Utilización
#<	Intervalo menor que?	
#<=	Intervalo menor o igual que?	
#<>	Intervalo no igual que?	
#=	Intervalo igual que?	
#>	Intervalo mayor que?	
#>=	Intervalo mayor o igual que?	
<#>	Convertir a un intervalo de tiempo	
<<	Intervalo menor que?	
	Comienzo de intervalo	
~=	Parecido a	
<?>	Tiempo dentro del intervalo?	

Operadores IP V4 CIDR

Tabla 4-6. Postgres Operadores IP V4 CIDR

Operador	Descripción	Utilización
<	Menor que	'192.168.1.5':cidr < '192.168.1.6':cidr
<=	Menor o igual que	'192.168.1.5':cidr <= '192.168.1.5':cidr
=	Igual que	'192.168.1.5':cidr = '192.168.1.5':cidr

Operador	Descripción	Utilizacióne
>=	Mayor o igual que	'192.168.1.5'::cidr >= '192.168.1.5'::cidr
>	Mayor que	'192.168.1.5'::cidr > '192.168.1.4'::cidr
<>	No igual que	'192.168.1.5'::cidr <> '192.168.1.4'::cidr
<<	Está contenido en	'192.168.1.5'::cidr << '192.168.1/24'::cidr
<<=	Está contenido en o es igual a	'192.168.1/24'::cidr <<= '192.168.1/24'::cidr
>>	Contiene	'192.168.1/24'::cidr >> '192.168.1.5'::cidr
>>=	Contiene o es igual que	'192.168.1/24'::cidr >>= '192.168.1/24'::cidr

Operadores IP V4 INET

Tabla 4-7. PostgresOperadores IP V4 INET

Operador	Descripción	Utilización
<	Menor que	'192.168.1.5'::inet < '192.168.1.6'::inet
<=	Menor o igual que	'192.168.1.5'::inet <= '192.168.1.5'::inet
=	Igual que	'192.168.1.5'::inet = '192.168.1.5'::inet
>=	Mayor o igual que	'192.168.1.5'::inet >= '192.168.1.5'::inet
>	Mayor que	'192.168.1.5'::inet > '192.168.1.4'::inet
<>	No igual	'192.168.1.5'::inet <> '192.168.1.4'::inet
<<	Está contenido en	'192.168.1.5'::inet << '192.168.1/24'::inet
<<=	Está contenido o es igual a	'192.168.1/24'::inet <<= '192.168.1/24'::inet
>>	Contiene	'192.168.1/24'::inet >> '192.168.1.5'::inet
>>=	Contiene o es igual a	'192.168.1/24'::inet >>= '192.168.1/24'::inet

Capítulo 5. Funciones

Describe las funciones built-in disponibles en Postgres.

Están disponibles muchos tipos de datos para la conversión a otros tipos relacionados. En adición, existen algunos tipos específicos de funciones. Algunas funciones también están disponibles a través de operadores y pueden ser documentadas solo como operadores.

Funciones SQL

“Funciones SQL” son contrucciones definidas por el standard SQL92, que tiene sintaxis igual que funciones pero que no pueden ser implementadas como simples funciones.

Tabla 5-1. Funciones SQL

Funciones	Retorna	Descripcion	Ejemplo
COALESCE(<i>list</i>)	no-NULO	retorna el primer valor no-NULO en la lista	COALESCE(<i>r"le</i> >, <i>c2</i> + 5, 0)
NULLIF(<i>input,value</i>)	<i>input</i> or NULO	retorna NULO si <i>input</i> = <i>value</i>	NULLIF(<i>c1</i> , 'N/A')
CASE WHEN <i>expr</i> THEN <i>expr</i> [...] ELSE <i>expr</i> END	<i>expr</i>	retorna la expresión para la primera cláusula verdadera	CASE WHEN <i>c1</i> = 1 THEN 'match' ELSE 'no match' END

Funciones Matemáticas

Tabla 5-2. Funciones Matemáticas

Funciones	Retorna	Descripcion	Ejemplo
dexp(float8)	float8	redimensiona al exponente especificado	dexp(2.0)
dpow(float8,float8)	float8	redimensiona un numero al exponente especificado	dpow(2.0, 16.0)
float(int)	float8	convierte un entero a punto flotante	float(2)

Funciones	Retorna	Descripcion	Ejemplo
float4(int)	float4	convierte un entero a punto flotante	float4(2)
integer(float)	int	convierte un punto flotante a entero	integer(2.0)

String Functions

SQL92 define funciones de texto con sintaxis específica. Algunas son implementadas usando otras funciones Postgres. Los tipos de Texto soportados para SQL92 son char, varchar, y text.

Tabla 5-3. SQL92 String Functions

Funciones	Retorna	Descripcion	Ejemplo
char_length(string)	int4	longitud del texto	char_length('jose')
character_length(string)	int4	longitud del texto	char_length('jose')
lower(string)	string	convierte el texto a minúsculas	lower('TOM')
octet_length(string)	int4	almacena el tamaño del texto	octet_length('jose')
position(string in string)	int4	localiza la posición de un subtexto especificado	position('o' in 'Tom')
substring(string [from int] [for int])	string	extrae un subtexto especificado	substring('Tom' from 2 for 2)
trim([leading trailing both] [string] from string)	string	borra caracteres de un texto	trim(both 'x' from 'xTomx')
upper(text)	text	convierte un texto a mayúsculas	upper('tom')

La mayoría de funciones de texto están disponibles para tipos text, varchar() y char(). Algunas son usadas internamente para implementar las funciones de texto SQL92 descritas arriba.

Tabla 5-4. Funciones de Texto

Funciones	Retorna	Descripcion	Ejemplo
char(text)	char	convierte un texto a tipo char	char('text string')

Funciones	Retorna	Descripcion	Ejemplo
char(varchar)	char	convierte un varchar a tipo char	char(varchar 'varchar string')
initcap(text)	text	primera letra de cada palabra a mayúsculas	initcap('thomas')
lpad(text,int,text)	text	relleno de caracteres por la izquierda a la longitud especificada	lpad('hi',4,'??')
ltrim(text,text)	text	recorte de caracteres por la izquierda del texto	ltrim('xxxxtrim','x')
textpos(text,text)	text	localiza un subtexto especificado	position('high','ig')
rpadd(text,int,text)	text	relleno de caracteres por la derecha a la longitud especificada	rpadd('hi',4,'x')
rtrim(text,text)	text	recorte de caracteres por la derecha del texto	rtrim('trimxxxx','x')
substr(text,int[,int])	text	extrae el subtexto especificado	substr('hi there',3,5)
text(char)	text	convierte char a tipo text	text('char string')
text(varchar)	text	convierte varchar a tipo text	text(varchar 'varchar string')
translate(text,from,to)	text	convierte character a string	translate('12345', '1', 'a')
varchar(char)	varchar	convierte char a tipo varchar	varchar('char string')
varchar(text)	varchar	convierte text a tipo varchar	varchar('text string')

La mayoría de funciones explícitamente definidas para texto trabajarán para argumentos char () y varchar().

Funciones de Fecha/Hora

Las funciones de Fecha/Hora provee un poderoso conjunto de herramientas para manipular varios tipos Date/Time.

Tabla 5-5. Date/Time Functions

Funciones	Retorna	Descripcion	Ejemplo
-----------	---------	-------------	---------

Funciones	Retorna	Descripcion	Ejemplo
abstime(datetime)	abstime	convierte a abstime	abstime('now'::datetime)
age(datetime,datetime)	timespan	preserva meses y años	age('now','1957-06-13'::datetime)
datetime(abstime)	datetime	convierte abstime a datetime	datetime('now'::abstime)
datetime(date)	datetime	convierte date a datetime	datetime('today'::date)
datetime(date,time)	datetime	convierte a datetime	datetime('1998-02-24'::datetime, '23:07'::time);
date_part(text,datetime)	float8	porción de fecha	date_part('dow','now'::datetime)
date_part(text,timespan)	float8	porción de hora	date_part('hour','4 hrs 3 mins'::timespan)
date_trunc(text,datetime)	datetime	fecha truncada	date_trunc('month','now'::abstime)
isfinite(abstime)	bool	un tiempo finito ?	isfinite('now'::abstime)
isfinite(datetime)	bool	una hora finita ?	isfinite('now'::datetime)
isfinite(timespan)	bool	una hora finita ?	isfinite('4 hrs'::timespan)
reltime(timespan)	reltime	convierte a reltime	reltime('4 hrs'::timespan)
timespan(reltime)	timespan	convierte a timespan	timespan('4 hours'::reltime)

Para las funciones `date_part` and `date_trunc`, los argumentos pueden ser 'year', 'month', 'day', 'hour', 'minute', y 'second', así como las mas especializadas cantidades 'decade', 'century', 'millenium', 'millisecond', y 'microsecond'. `date_part` permite 'dow' para retornar el día de la semana 'epoch' para retornar los segundos desde 1970 (para datetime) o 'epoch' para retornar el total de segundos transcurridos(para timespan)

Funciones de Formato

Author: Written by Karel Zak¹ on 2000-01-24.

Las funciones de formato proveen un poderoso conjunto de herramientas para convertir varios datatypes (date/time, int, float, numeric) a texto formateado y convertir de texto formateado a su datatypes original.

Tabla 5-6. Funciones de Formato

Funciones	Retorna	Descripcion	Ejemplo
to_char(datetime, text)	text	convierte datetime a string	to_char('now'::datetime, 'HH12:MI:SS')
to_char(timestamp, text)	text	convierte timestamp a string	to_char(now(), 'HH12:MI:SS')
to_char(int, text)	text	convierte int4/int8 a string	to_char(125, '999')
to_char(float, text)	text	convierte float4/float8 a string	to_char(125.8, '999D9')
to_char(numeric, text)	text	convierte numeric a string	to_char(-125.8, '999D99S')
to_datetime(text, text)	datetime	convierte string a datetime	to_datetime('05 Dec 2000 13', 'DD Mon YYYY HH')
to_date(text, text)	date	convierte string a date	to_date('05 Dec 2000', 'DD Mon YYYY')
to_timestamp(text, text)	date	convierte string a timestamp	to_timestamp('05 Dec 2000', 'DD Mon YYYY')
to_number(text, text)	numeric	convierte string a numeric	to_number('12,454.8', '99G999D9S')

Para todas las funciones de formato, el segundo argumento es format-picture.

Tabla 5-7. Format-pictures para date/time to_char() versión.

Format-picture	Descripción
HH	hora del día(01-12)
HH12	hora del día(01-12)
MI	minuto (00-59)

Format-picture	Descripción
SS	segundos (00-59)
SSSS	segundos pasados la medianoche(0-86399)
Y,YYY	año(4 o mas dígitos) con coma
YYYY	año(4 o mas dígitos)
YYY	últimos 3 dígitos del año
YY	últimos 2 dígitos del año
Y	último dígito del año
MONTH	nombre completo del mes(9-letras) - todos los caracteres en mayúsculas
Month	nombre completo del mes(9-letras) - el primer carácter en mayúsculas
month	nombre completo del mes(9-letras) - todos los caracteres en minúsculas
MON	nombre abreviado del mes(3-letras) - todos los caracteres en mayúsculas
Mon	nombre abreviado del mes(3-letras) - el primer carácter en mayúsculas
mon	nombre abreviado del mes(3-letras) - todos los caracteres en minúsculas
MM	mes (01-12)
DAY	nombre completo del día(9-letters) - todos los caracteres en mayúsculas
Day	nombre completo del día(9-letters) - el primer carácter en mayúsculas
day	nombre completo del día(9-letters) - todos los caracteres en minúsculas
DY	nombre abreviado del día(3-letters) - todos los caracteres en mayúsculas
Dy	nombre abreviado del día(3-letters) - el primer carácter en mayúsculas
dy	nombre abreviado del día(3-letters) - todos los caracteres en minúsculas
DDD	día del año(001-366)
DD	día del mes(01-31)
D	día de la semana(1-7; SUN=1)
W	semana del mes
WW	número de la semana en el año
CC	centuria (2-digits)
J	día juliano(días desde Enero 1, 4712 BC)
Q	quarter
RM	mes en numeral romano(I-XII; I=ENE)

Todos los format-pictures permiten usar sufijos (postfix / prefix). El sufijo es valido para una near format-picture. El 'FX' es solo prefijo global.

Tabla 5-8. Suffixes para format-pictures para date/time to_char() version.

Sufijo	Descripción	Ejemplo
FM	modo relleno- prefix	FMMonth
TH	numero ordinal superior - postfix	DDTH
th	numero ordinal inferior - postfix	DDTH
FX	FX - (Fixed format) conmutador global de format-picture . The TO_DATETIME / TO_DATE salta los espacios en blanco si esta opción no es usada. Debe ser usada como primer item en formt-picture.	FX Month DD Day
SP	spell mode (not implement now)	DDSP

'\ ' - debe ser usado como doble \ , ejemplo '\HH\MI\SS'

''' - el texto entre comillas es saltado y no retocado. Si quieres escribir ' ' ' a la salida debes usar '\ ' , ejemplo '\ "YYYY Month\ " ' . .

text - el to_char() de PostgreSQL soporta texto sin ''' , pero el texto entre las comillas es mas rápido y tienes la seguridad que el texto no será interpretado como keyword (format-picture), ejemplo '"Hello Year: "YYYY" ' . .

Tabla 5-9. Format-pictures para number (int/float/numeric) to_char() version.

Format-picture	Descripción
9	valor retornado con el número especificado de dígitos y si no estan disponibles usa espacios en blanco
0	como 9, pero en lugar de espacios en blanco usa ceros
. (period)	punto decimal
, (comma)	separador de grupo (miles)
PR	retorna el valor negativo en angle brackets
S	retorna el valor negativo con el signo menos (usa locales)
L	símbolo monetario (usa locales)

Format-picture	Descripción
D	punto decimal (usa locales)
G	separador de grupos (usa locales)
MI	retorna el signo menos en la posición especificada (si número < 0)
PL	retorna el signo mas en la posición especificada (si número > 0) - PostgreSQL extension
SG	retorna el signo mas/menos en la posición especificada - PostgreSQL extension
RN	retorna el número como número romano(número debe ser entre 1 y 3999)
TH or th	convierte el número a número ordinal (no convertir números menores que cero y números decimales) - PostgreSQL extension
V	$\text{arg1} * (10^n)$; - retorna un valor multiplicado por 10^n (donde 'n' es número de '9's despues de 'V'). El to_char() no soporta el uso de 'V' y punto decimal juntos, ejemplo "99.9V99".
EEEE	numeros cientificos . ahora no soportados.

Note: Un signo formateado via 'SG', 'PL' o 'MI' is not anchor in number; to_char(-12, 'S9999') produce:

```
'   -12'
```

, but to_char(-12, 'MI9999') produce:

```
'-  12'
```

. Oracle no permite usar 'MI' delante de '9', en Oracle tiene que ser siempre despues de '9'.

Tabla 5-10. El to_char() en ejemplos.

Input	Output
to_char(now(), 'Day, HH12:MI:SS')	'Tuesday , 05:39:18'
to_char(now(), 'FMDay, HH12:MI:SS')	'Tuesday, 05:39:18'
to_char(-0.1, '99.99')	' -.10'
to_char(-0.1, 'FM9.99')	'-.1'
to_char(0.1, '0.9')	' 0.1'
to_char(12, '9990999.9')	' 0012.0'
to_char(12, 'FM9990999.9')	'0012'

Input	Output
to_char(485, '999')	' 485 '
to_char(-485, '999')	' -485 '
to_char(485, '9 9 9')	' 4 8 5 '
to_char(1485, '9,999')	' 1,485 '
to_char(1485, '9G999')	' 1 485 '
to_char(148.5, '999.999')	' 148.500 '
to_char(148.5, '999D999')	' 148,500 '
to_char(3148.5, '9G999D999')	' 3 148,500 '
to_char(-485, '999S')	'485- '
to_char(-485, '999MI')	'485- '
to_char(485, '999MI')	'485 '
to_char(485, 'PL999')	' +485 '
to_char(485, 'SG999')	' +485 '
to_char(-485, 'SG999')	' -485 '
to_char(-485, '9SG99')	'4-85 '
to_char(-485, '999PR')	' <485 > '
to_char(485, 'L999')	'DM 485 '
to_char(485, 'RN')	' CDLXXXV '
to_char(485, 'FMRN')	'CDLXXXV '
to_char(5.2, 'FMRN')	'V '
to_char(482, '999th')	' 482nd '
to_char(485, '"Good number:"999')	'Good number: 485 '
to_char(485.8, '"Pre-decimal:"999" Post-decimal:" .999')	'Pre-decimal: 485 Post- decimal: .800 '
to_char(12, '99V999')	' 12000 '
to_char(12.4, '99V999')	' 12400 '
to_char(12.45, '99V9')	' 125 '

Funciones Geométricas

Los tipos geométricos point, box, lseg, line, path, polygon, and circle tienen un gran conjunto de funciones nativas soportadas.

Tabla 5-11. Funciones Geométricas

Funciones	Retorna	Descripcion	Ejemplo
-----------	---------	-------------	---------

Funciones	Retorna	Descripción	Ejemplo
area(box)	float8	área del rectángulo	area('((0,0),(1,1))':box)
area(circle)	float8	área del círculo	area('((0,0),2.0)':circle)
box(box,box)	box	rectángulo de intersección de rectángulos	box('((0,0),(1,1))','((0.5,0.5),(2,2))')
center(box)	point	centro del objeto	center('((0,0),(1,2))':box)
center(circle)	point	centro del objeto	center('((0,0),2.0)':circle)
diameter(circle)	float8	diámetro del círculo	diameter('((0,0),2.0)':circle)
height(box)	float8	tamaño vertical del rectángulo	height('((0,0),(1,1))':box)
isclosed(path)	bool	ruta cerrada ?	isclosed('((0,0),(1,1),(2,0))':path)
isopen(path)	bool	ruta abierta ?	isopen('[(0,0),(1,1),(2,0)]':path)
length(lseg)	float8	longitud de la línea segmento	length('((-1,0),(1,0))':lseg)
length(path)	float8	longitud de la ruta	length('((0,0),(1,1),(2,0))':path)
pclose(path)	path	convierte path a closed	pclose('[(0,0),(1,1),(2,0)]':path)
point(lseg,lseg)	point	intersección	point('((-1,0),(1,0))':lseg,'((-2,-2),(2,2))':lseg)
points(path)	int4	número de puntos	points('[(0,0),(1,1),(2,0)]':path)
popen(path)	path	convierte path a open	popen('((0,0),(1,1),(2,0))':path)

Funciones	Retorna	Descripción	Ejemplo
radius(circle)	float8	radio del círculo	radius('((0,0),2.0)')::circle)
width(box)	float8	tamaño horizontal	width('((0,0),(1,1))')::box)

Tabla 5-12. Funciones de conversión de tipos Geométricos

Funciones	Retorna	Descripción	Ejemplo
box(circle)	box	convierte círculo a rectángulo	box('((0,0),2.0)')::circle)
box(point,point)	box	convierte puntos a rectángulo	box('(0,0)'::point,'(1,1)'::point)
box(polygon)	box	convierte polígono a rectángulo	box('((0,0),(1,1),(2,0))')::polygon)
circle(box)	circle	convierte a círculo	circle('((0,0),(1,1))')::box)
circle(point,float8)	circle	convierte a círculo	circle('(0,0)'::point,2.0)
lseg(box)	lseg	convierte diagonal a lseg	lseg('((-1,0),(1,0))')::box)
lseg(point,point)	lseg	convierte a lseg	lseg('(-1,0)'::point,'(1,0)'::point)
path(polygon)	point	convierte a path	path('((0,0),(1,1),(2,0))')::polygon)
point(circle)	point	convierte a punto (centro)	point('((0,0),2.0)')::circle)
point(lseg,lseg)	point	convierte a punto (intersección)	point('((-1,0),(1,0))'::lseg,'((-2,-2),(2,2))'::lseg)
point(polygon)	point	centro de polígono	point('((0,0),(1,1),(2,0))')::polygon)
polygon(box)	polygon	convierte a polígono con 12 puntos	polygon('((0,0),(1,1))')::box)

Funciones	Retorna	Descripción	Ejemplo
<code>polygon(circle)</code>	<code>polygon</code>	convierte a polígono con 12 puntos	<code>polygon('((0,0),2.0)::circle)</code>
<code>polygon(<i>npts</i>,circle)</code>	<code>polygon</code>	convierte a polígono <i>npts</i>	<code>polygon(12,'((0,0),2.0)::circle)</code>
<code>polygon(path)</code>	<code>polygon</code>	convierte a <code>polygon</code>	<code>polygon('((0,0),(1,1),(2,0))::path)</code>

Tabla 5-13. Funciones de Actualización Geométrica

Funciones	Retorna	Descripción	Ejemplo
<code>isoldpath(path)</code>	<code>path</code>	test path for pre-v6.1 form	<code>isold-path('(1,3,0,0,1,1,2,0)::path)</code>
<code>revertpoly(polygon)</code>	<code>polygon</code>	convierte pre-v6.1 polygon	<code>revert-poly('((0,0),(1,1),(2,0))::polygon)</code>
<code>upgradepath(path)</code>	<code>path</code>	convierte pre-v6.1 path	<code>upgrade-path('(1,3,0,0,1,1,2,0)::path)</code>
<code>upgrade-poly(polygon)</code>	<code>polygon</code>	convierte pre-v6.1 polygon	<code>upgrade-poly('(0,1,2,0,1,0)::polygon)</code>

Funciones PostgresIP V4

Tabla 5-14. Funciones PostgresIP V4

Funciones	Retorna	Descripción	Ejemplo
<code>broadcast(cidr)</code>	<code>text</code>	contruye la dirección broadcast como texto	<code>broadcast('192.168.1.5/24')</code>
<code>broadcast(inet)</code>	<code>text</code>	contruye la dirección broadcast como texto	<code>broadcast('192.168.1.5/24')</code>
<code>host(inet)</code>	<code>text</code>	extrae la dirección host como texto	<code>host('192.168.1.5/24')</code>

Funciones	Retorna	Descripcion	Ejemplo
masklen(cidr)	int4	calcula la longitud del netmask	masklen('192.168.1.5/24')
masklen(inet)	int4	calcula la longitud del netmask	masklen('192.168.1.5/24')
netmask(inet)	text	contruye el netmask como texto	netmask('192.168.1.5/24')

Notas

1. <mailto:zakkr@zf.jcu.cz>

Capítulo 6. Conversión de tipos

Las consultas SQL pueden, intencionadamente o no, requerir mezclar diferentes tipos de datos en una misma expresión. Postgres posee grandes facilidades para evaluar expresiones que contengan diferentes tipos.

En muchos casos un usuario no necesita comprender los detalles del mecanismo de conversión de tipos. Sin embargo, la conversión implícita realizada por Postgres puede afectar a los resultados de una consulta. Estos resultados pueden ser ajustados por un usuario o por un programador usando conversión de tipos *explícita*.

Este capítulo es una introducción a los mecanismos y convenciones de conversión de tipos en Postgres. Diríjase a las secciones correspondientes en la guía del usuario y en la guía del programador para obtener más información sobre tipos de datos específicos, funciones y operadores permitidos.

La guía del programador tiene más detalles sobre los algoritmos exactos usados por la conversión implícita de tipos.

Conceptos generales

SQL es un lenguaje con una definición de tipos rígida. Así, cada dato tiene asociado un tipo de dato que determina como se comporta y como se permite usar. Postgres tiene un sistema de tipos extensible que es mucho más general y flexible que otras implementaciones RDBMS. Por lo tanto, la mayoría de las reglas para convertir tipos en Postgres pueden ser regidas por unas normas generales bastante mejores que unas normas heurísticas que permitan a las expresiones con tipos distintos mezclados ser significantes, de la misma manera sucede con los tipos definidos por el usuario.

El analizador de Postgres clasifica los elementos léxicos en solo cinco categorías fundamentales: enteros, reales, cadenas, nombres y palabras clave. La mayoría de los tipos extendidos son convertidos en cadenas en primer lugar. El lenguaje de definición SQL permite especificar nombres de tipo con cadenas. Este mecanismo es usado por Postgres para indicar al analizador el camino correcto. Por ejemplo, la consulta:

```
tgl=> SELECT text 'Origin' AS "Label", point '(0,0)' AS "Value";
Label |Value
-----+-----
Origin|(0,0)
(1 row)
```

tiene dos cadenas, de tipo text y de tipo point. Si un tipo no es especificado, entonces el tipo unknown es asignado inicialmente. En posteriores fases se resolverá tal y como se describe más adelante.

Hay cuatro construcciones fundamentales en SQL las cuales requieren distintas reglas de conversión de tipos en el analizador de Postgres:

Operadores

Postgres permite tanto expresiones con operadores de un solo argumento como con operadores de dos argumentos.

Llamadas a funciones

Gran parte del sistema de tipos de Postgres está construido alrededor de un rico conjunto de funciones. Las llamadas a funciones tienen uno o más argumentos los cuales, para cualquier consulta específica, deben ser adaptados a las funciones disponibles en el sistema.

Objetivos de consultas

Una declaración SQL INSERT pone los resultados de una consulta en una tabla. Las expresiones en la consulta debe ser ajustadas, y quizás convertidas, a las columnas del objetivo del INSERT.

Consultas UNION

Debido a que todos los resultados de una declaración UNION SELECT deben aparecer como un único conjunto de columnas, los tipos de cada clausula SELECT deben ser ajustados y convertidos a un conjunto uniforme.

Muchas de las reglas de conversión de tipos generales usan convenciones sencillas que están en las tablas del sistema de funciones y operadores de Postgres. Hay algo de heurística en las reglas de conversión para dar un mejor soporte a las convenciones de los tipos nativos estándar de SQL92 como smallint, integer, y float.

El analizador de Postgres usa la convención de que todas las funciones de conversión de tipo toman un solo argumento como tipo de origen y se llaman de la misma manera que el tipo de destino. Se considera que cualquier función que cumpla este criterio es una función de conversión válida, y debe ser usada por el analizador de esta manera. Esta simple afirmación le da al analizador el poder para explorar las posibilidades de conversión de tipo sin dificultad, permitiendo a los tipos definidos por el usuario usar las mismas características de manera transparente.

El analizador esta provisto de una lógica adicional para permitir ajustarse más a la conducta correcta de los tipos estándar SQL. Hay cinco categorías de tipos definidas: boolean, string, numeric, geometric y user-defined. Cada categoría, con la excepción de user-defined, tiene un "tipo preferido" el cual es usado para resolver ambigüedades entre los candidatos. Cada tipo "user-defined" es su propio "tipo preferido", así las expresiones ambiguas (aquellas en las que el analizador tiene varios candidatos) con solo un tipo definido por el usuario pueden resolverse con una única solución, mientras que las que tienen varios tipos definidos por el usuario serán ambiguas y darán un error.

Las expresiones ambiguas que tienen posibles soluciones con solo una categoría de tipos son fáciles de resolver, mientras que las expresiones ambiguas con posibles soluciones de distintas categorías dan fácilmente un error y preguntan al usuario una aclaración.

Guidelines

Todas las reglas de conversión de tipos están diseñadas teniendo presentes diversos principios:

- Las conversiones implícitas no deberían tener nunca un resultado sorprendente o impredecible.
- Los tipos definidos por el usuario, de los cuales el analizador no tiene conocimiento a priori, deben de estar situados en un lugar alto dentro de la jerarquía de tipos. Dentro de expresiones con tipos mezclados, los tipos nativos deberían ser convertidos siempre a tipos definidos por el usuario (por supuesto, solo si la conversión es necesaria).
- Los tipos definidos por el usuario no están relacionados. Por lo general, Postgres no tiene disponible información sobre las relaciones entre tipos aparte de la lógica codificada para los tipos predefinidos y las relaciones implícitas basadas en las funciones disponibles en el catálogo.
- No debería haber una carga extra del analizador o del ejecutor si una consulta no necesita conversión implícita de tipos. De esta manera, si una consulta esta bien construida y los tipos ya están adaptados, entonces la consulta debería realizarse

sin consumir tiempo extra en el analizador y sin realizar funciones de conversión innecesarias dentro de la consulta.

Adicionalmente, si una consulta normalmente requiere una conversión implícita para una función, y entonces el usuario define una función explícita con los tipos de los argumentos correctos, el analizador debería usar esta nueva función y no realizar nunca más una conversión implícita usando la función antigua.

Operadores

Procedimiento de conversión

Operador de evaluación

1. Inspecciona en busca de un ajuste exacto en el catálogo del sistema pg_operator.
 - a. Si un argumento de un operador binario es unknown, entonces se asume que es del mismo tipo que el otro argumento.
 - b. Invierte los argumentos, y busca un ajuste exacto con un operador el cual apunta a él mismo ya que es conmutativo. Si lo halla, entonces invierte los argumentos en el árbol del analizador y usa este operador.
2. Busca el mejor ajuste.
 - a. Hace una lista de todos los operadores con el mismo nombre.
 - b. Si solo hay un operador en la lista usa este si el tipo de la entrada puede ser forzado, y genera un error si el tipo no puede ser forzado.
 - c. Guarda todos los operadores con los ajustes más explícitos de tipos. Guarda todo si no hay ajustes explícitos y salta al siguiente paso. Si solo queda un candidato, usa este si el tipo puede ser forzado.
 - d. Si algún argumento de entrada es "unknown", categoriza los candidatos de entrada como boolean, numeric, string, geometric, o user-defined. Si hay una mezcla de categorías, o más de un tipo definido por el usuario, genera un error porque la elección correcta no puede ser deducida sin más pistas. Si solo está presente una categoría, entonces asigna el tipo preferido a la columna de entrada que previamente era "unknown".
 - e. Escoge el candidato con los ajustes de tipos más exactos, y que ajustan el "tipo preferido" para cada categoría de columna del paso previo. Si todavía queda más de un candidato, o si no queda ninguno, entonces se genera un error.

Ejemplos

Operador exponente

Solo hay un operador exponente definido en el catálogo, y toma argumentos float8. El examinador asigna un tipo inicial int4 a ambos argumentos en la expresión de esta consulta:

```
tgl=> select 2 ^ 3 AS "Exp";
Exp
--
      8
(1 row)
```

De esta manera, el analizador hace una conversión de tipo sobre ambos operadores y la consulta es equivalente a

```
tgl=> select float8(2) ^ float8(3) AS "Exp";
Exp
--
      8
(1 row)
```

or

```
tgl=> select 2.0 ^ 3.0 AS "Exp";
Exp
--
      8
(1 row)
```

Nota: Esta ultima forma es la que tiene menos sobrecarga, ya que no se llama a funciones para hacer un conversión implícita de tipo. Esto no es una ventaja para pequeñas consultas, pero puede tener un gran impacto en el rendimiento de consultas que abarquen muchas tablas.

Concatenación de cadenas

Una sintaxis similar es usada tanto para trabajar con tipos alfanuméricos como con tipos complejos extendidos. Las cadenas alfanuméricas con tipo sin especificar son ajustadas con los operadores candidatos afines.

Un argumento sin especificar:

```
tgl=> SELECT text 'abc' || 'def' AS "Text and Unknown";
Text and Unknown
-----
abcdef
(1 row)
```

En este caso el analizador mira si existe algún operador que necesite el operador text en ambos argumentos. Si existe, asume que el segundo operador debe ser interpretado como de tipo text.

Concatenación con tipos sin especificar:

```

tgl=> SELECT 'abc' || 'def' AS "Unspecified";
Unspecified
-----
abcdef
(1 row)

```

En este caso hay ninguna pista inicial sobre que tipo usar, ya que no se han especificado tipos en la consulta. De esta manera, el analizador busca en todos los operadores candidatos aquellos en los que todos los argumentos son de tipo alfanumérico. Elige el "tipo preferido" para las cadenas alfanuméricas, text, para esta consulta.

Nota: Si un usuario define un nuevo tipo y define un operador "||" para trabajar con el, entonces esta consulta tal como esta escrita no tendrá éxito. El analizador tendría ahora tipos candidatos de dos categorías, y no podría decidir cual de ellos usar.

Factorial

Este ejemplo ilustra un interesante resultado. Tradicionalmente, el operador factorial está definido solo para enteros. El catalogo de operadores de Postgres tiene solamente una entrada para el factorial, que toma un entero como operador. Si recibe un argumento numérico no entero, Postgres intentará convertir este argumento a un entero para la evaluación del factorial.

```

tgl=> select (4.3 !);
?column?
-----
      24
(1 row)

```

Nota: Por supuesto, esto conduce a un resultado matemáticamente sospechoso, debido a que en principio el factorial de un número no entero no está definido. De cualquier modo, el papel de una base de datos no es enseñar matemáticas, sino más bien ser una herramienta para manipular datos. Si un usuario decide obtener el factorial de un número real, Postgres intentará hacerlo.

Funciones

Evaluación de función

1. Busca una entrada exacta en el catálogo del sistema pg_proc.
2. Busca la mejor entrada.

- a. Hace una lista de todas las funciones con el mismo nombre y con el mismo número de argumentos.
- b. Si solo hay una función en la lista, usa esta si los tipos de la entrada pueden ser convertidos, y produce un error si los tipos no pueden ser convertidos.
- c. Guarda todas las funciones con los ajustes más explícitos para los tipos. Guarda todas si no hay ajustes explícitos y salta al siguiente paso. Si solo queda un candidato, usa este si el tipo puede ser convertido.
- d. Si cualquiera de los argumentos de entrada son de tipo desconocido, clasifica los argumentos de entrada candidatos en categorías como boolean, numeric, string, geometric o user-defined. Si hay una mezcla de categorías, o más de un tipo definido por el usuario, se produce un error debido a que la elección correcta no puede ser deducida si no se aportan más pistas. Si solo hay una categoría, entonces asigna el "tipo preferido" a la columna de entrada que antes era de tipo desconocido.
- e. Escoge el candidato con el ajuste de tipos más exacto, y el cual ajusta el "tipo preferido" a cada categoría de columna desde el paso anterior. Si hay más de un candidato, o si no hay ninguno, entonces se produce un error.

Ejemplos

Función factorial

Solo hay una función factorial definida en el catálogo pg_proc. Debido a esto, las siguientes consultas convierten automáticamente el argumento int2 a int4:

```
tgl=> select int4fac(int2 '4');
int4fac
-----
      24
(1 row)
```

y es de hecho transformado por el analizador a

```
tgl=> select int4fac(int4(int2 '4'));
int4fac
-----
      24
(1 row)
```

Función substring

Hay dos funciones substr declaradas en pg_proc. Sin embargo, solo una tiene dos argumentos, de tipos text y int4.

Si es llamada con una constante de cadena de tipo sin especificar, el tipo es ajustado directamente con la única función candidata de tipo:

```
tgl=> select substr('1234', 3);
substr
-----
      34
(1 row)
```

Si la cadena es declarada como tipo varchar, como puede ser en el caso de que venga de una tabla, entonces el analizador intentará convertirla al tipo text:

```
tgl=> select substr(varchar '1234', 3);
substr
-----
      34
(1 row)
```

lo que es transformado por el analizador a:

```
tgl=> select substr(text(varchar '1234'), 3);
substr
-----
      34
(1 row)
```

Nota: Hay algunas estrategias en el analizador para optimizar la relación entre los tipos char, varchar y text. En este caso, la función `substr` es llamada directamente con una cadena varchar en vez de hacer una llamada para realizar una conversión explícita.

Y, si la función es llamada con un int4, el analizador intentará convertirlo a text

```
tgl=> select substr(1234, 3);
substr
-----
      34
(1 row)
```

realmente se ejecuta como

```
tgl=> select substr(text(1234), 3);
substr
-----
      34
(1 row)
```

Resultados de consultas

Evaluación del resultado

1. Busca un ajuste exacto con el resultado.
2. Si es necesario intenta convertir la expresión directamente al tipo del resultado.
3. Si el resultado es un tipo de longitud fija (por ejemplo char o varchar declarado con una longitud) entonces intenta encontrar una función que ajuste la longitud con el mismo nombre que el tipo de los dos argumentos, el primero el nombre del tipo y el segundo un entero con la longitud.

Ejemplos

Almacenamiento de varchar

Para cada columna declarada como varchar(4) la siguiente consulta asegura que el resultado tiene el tamaño adecuado:

```

tgl=> CREATE TABLE vv (v varchar(4));
CREATE
tgl=> INSERT INTO vv SELECT 'abc' || 'def';
INSERT 392905 1
tgl=> select * from vv;
v
---
abcd
(1 row)

```

Consultas UNION

La construcción UNION es algo diferente en cuanto que es más posible el que haya tipos distintos en un resultado.

Evaluación de UNION

1. Comprueba si los tipos son idénticos para todos los resultados.
2. Convierte cada resultado de la cláusula UNION para ajustarlo al tipo de la primera cláusula SELECT o de la columna de resultado.

Ejemplos

Tipos sin especificar

```

tgl=> SELECT text 'a' AS "Text" UNION SELECT 'b';
Text
---
a
b
(2 rows)

```

UNION simple

```

tgl=> SELECT 1.2 AS Float8 UNION SELECT 1;
Float8
----
1
1.2
(2 rows)

```

UNION transpuesto

Los tipos del UNION son forzados a ajustarse a los tipos de la primera clausula en el UNION:

```

tgl=> SELECT 1 AS "All integers"
tgl-> UNION SELECT '2.2'::float4
tgl-> UNION SELECT 3.3;
All integers
-----
          1
          2
          3
(3 rows)

```

Una estrategia alternativa del analizador podría ser escoger el "mejor" tipo del grupo, pero esto es más difícil debido a la técnica recursiva usada en el analizador. De cualquier modo, se usa el "mejor" tipo cuando hacemos una selección *dentro* de una tabla:

```

tgl=> CREATE TABLE ff (f float);
CREATE
tgl=> INSERT INTO ff
tgl-> SELECT 1
tgl-> UNION SELECT '2.2'::float4
tgl-> UNION SELECT 3.3;
INSERT 0 3
tgl=> SELECT f AS "Floating point" from ff;
Floating point
-----
          1
2.20000004768372
          3.3
(3 rows)

```


Capítulo 7. Indices and Keys

Indexes are primarily used to enhance database performance. They should be defined on table columns (or class attributes) which are used as qualifications in repetitive queries. Inappropriate use will result in slower performance, since update and insertion times are increased in the presence of indices.

Two forms of indices may be defined:

- For a *value index*, the key fields for the index are specified as column names; a column may also have an associated operator class. An operator class is used to specify the operators to be used for a particular index. For example, a btree index on four-byte integers would use the `int4_ops` class; this operator class includes comparison functions for four-byte integers. The default operator class is the appropriate operator class for that field type.
- For a *functional index*, an index is defined on the result of a user-defined function applied to one or more attributes of a single class. These functional indices can be used to obtain fast access to data based on operators that would normally require some transformation to apply them to the base data.

Postgres provides btree, rtree and hash access methods for secondary indices. The btree access method is an implementation of the Lehman-Yao high-concurrency btrees. The rtree access method implements standard rtrees using Guttman's quadratic split algorithm. The hash access method is an implementation of Litwin's linear hashing. We mention the algorithms used solely to indicate that all of these access methods are fully dynamic and do not have to be optimized periodically (as is the case with, for example, static hash access methods).

The Postgres query optimizer will consider using btree indices in a scan whenever an indexed attribute is involved in a comparison using one of: `<`, `<=`, `=`, `>=`, `>`

Both box classes support indices on the `box` data type in Postgres. The difference between them is that `bigbox_ops` scales box coordinates down, to avoid floating point exceptions from doing multiplication, addition, and subtraction on very large floating-point coordinates. If the field on which your rectangles lie is about 20,000 units square or larger, you should use `bigbox_ops`. The `poly_ops` operator class supports rtree indices on `polygon` data.

The Postgres query optimizer will consider using an rtree index whenever an indexed attribute is involved in a comparison using one of: `<<`, `&<`, `&>`, `>>`, `@`, `~=`, `&&`

The Postgres query optimizer will consider using a hash index whenever an indexed attribute is involved in a comparison using the `=` operator.

Currently, only the BTREE access method supports multi-column indexes. Up to 7 keys may be specified.

Use `DROP INDEX` to remove an index.

The `int24_ops` operator class is useful for constructing indices on `int2` data, and doing comparisons against `int4` data in query qualifications. Similarly, `int42_ops` support indices on `int4` data that is to be compared against `int2` data in queries.

The following select list returns all `ops_names`:

```
SELECT am.amname AS acc_name,
       opc.opcname AS ops_name,
       opr.oprname AS ops_comp
FROM   pg_am am, pg_amop amop,
       pg_opclass opc, pg_operator opr
```

```

WHERE amop.amopid = am.oid AND
      amop.amopclaid = opc.oid AND
      amop.amopopr = opr.oid
ORDER BY acc_name, ops_name, ops_comp

```

Keys

Author: Written by Herouth Maoz¹ This originally appeared on the User's Mailing List on 1998-03-02 in response to the question: "What is the difference between PRIMARY KEY and UNIQUE constraints?".

Subject: Re: [QUESTIONS] PRIMARY KEY | UNIQUE

What's the difference between:

```

PRIMARY KEY(fields,...) and
UNIQUE (fields,...)

```

- Is this an alias?
- If PRIMARY KEY is already unique, then why is there another kind of key named UNIQUE?

A primary key is the field(s) used to identify a specific row. For example, Social Security numbers identifying a person.

A simply UNIQUE combination of fields has nothing to do with identifying the row. It's simply an integrity constraint. For example, I have collections of links. Each collection is identified by a unique number, which is the primary key. This key is used in relations.

However, my application requires that each collection will also have a unique name. Why? So that a human being who wants to modify a collection will be able to identify it. It's much harder to know, if you have two collections named "Life Science", the one tagged 24433 is the one you need, and the one tagged 29882 is not.

So, the user selects the collection by its name. We therefore make sure, within the database, that names are unique. However, no other table in the database relates to the collections table by the collection Name. That would be very inefficient.

Moreover, despite being unique, the collection name does not actually define the collection! For example, if somebody decided to change the name of the collection from "Life Science" to "Biology", it will still be the same collection, only with a different name. As long as the name is unique, that's OK.

So:

- Primary key:
 - Is used for identifying the row and relating to it.

- Is impossible (or hard) to update.
- Should not allow NULLs.
- Unique field(s):
 - Are used as an alternative access to the row.
 - Are updateable, so long as they are kept unique.
 - NULLs are acceptable.

As for why no non-unique keys are defined explicitly in standard SQL syntax? Well, you must understand that indices are implementation-dependent. SQL does not define the implementation, merely the relations between data in the database. Postgres does allow non-unique indices, but indices used to enforce SQL keys are always unique.

Thus, you may query a table by any combination of its columns, despite the fact that you don't have an index on these columns. The indexes are merely an implementational aid which each RDBMS offers you, in order to cause commonly used queries to be done more efficiently. Some RDBMS may give you additional measures, such as keeping a key stored in main memory. They will have a special command, for example

```
CREATE MEMSTORE ON <table> COLUMNS <cols>
```

(this is not an existing command, just an example).

In fact, when you create a primary key or a unique combination of fields, nowhere in the SQL specification does it say that an index is created, nor that the retrieval of data by the key is going to be more efficient than a sequential scan!

So, if you want to use a combination of fields which is not unique as a secondary key, you really don't have to specify anything - just start retrieving by that combination! However, if you want to make the retrieval efficient, you'll have to resort to the means your RDBMS provider gives you - be it an index, my imaginary MEMSTORE command, or an intelligent RDBMS which creates indices without your knowledge based on the fact that you have sent it many queries based on a specific combination of keys... (It learns from experience).

Partial Indices

Author: This is from a reply to a question on the e-mail list by Paul M. Aoki² on 1998-08-11.

A *partial index* is an index built over a subset of a table; the subset is defined by a predicate. Postgres supported partial indices with arbitrary predicates. I believe IBM's db2 for as/400 supports partial indices using single-clause predicates.

The main motivation for partial indices is this: if all of the queries you ask that can profitably use an index fall into a certain range, why build an index over the whole table and suffer the associated space/time costs? (There are other reasons too; see *Stonebraker, M, 1989b* for details.)

The machinery to build, update and query partial indices isn't too bad. The hairy parts are index selection (which indices do I build?) and query optimization (which indices do I use?); i.e., the parts that involve deciding what predicate(s) match the

workload/query in some useful way. For those who are into database theory, the problems are basically analogous to the corresponding materialized view problems, albeit with different cost parameters and formulae. These are, in the general case, hard problems for the standard ordinal SQL types; they're super-hard problems with black-box extension types, because the selectivity estimation technology is so crude.

Check *Stonebraker, M, 1989b*, *Olson, 1993*, and for more information.

Notas

1. herouth@oumail.openu.ac.il
2. aoki@CS.Berkeley.EDU

Capítulo 8. Matrices

Nota: Este debe convertirse en una capítulo sobre el comportamiento de los matrices.
¿Voluntarios? - thomas 1998-01-12

Postgres permite que los atributos de una instancia sean definidos como una matriz multidimensional de longitud fija o variable. Pueden crearse matrices de cualquier tipo (incluyendo tipos definidos por el usuario). Para ilustrar su uso, primero creamos una clase con matrices de tipos base.

```
CREATE TABLE SAL_EMP (  
    name          text,  
    pay_by_quarter int4[],  
    schedule      text[][]);
```

La consulta de arriba creará una clase llamada SAL_EMP con una cadena de tipo *text* (name), una matriz unidimensional de tipo *int4* (pay_by_quarter), que representa el salario trimestral del empleado y una matriz bidimensional de tipo *text* (schedule), el cual representa el horario semanal del empleado. Ahora hacemos algunos *INSERT*; fíjese que cuando se agregan elementos a una matriz, encerramos los valores entre llaves y los separamos con comas. Si usted conoce el lenguaje C, esto no es muy diferente de la sintaxis que se utiliza para inicializar estructuras.

```
INSERT INTO SAL_EMP  
VALUES ('Bill',  
    '{10000, 10000, 10000, 10000}',  
    '{{"meeting", "lunch"}, {}}');  
  
INSERT INTO SAL_EMP  
VALUES ('Carol',  
    '{20000, 25000, 25000, 25000}',  
    '{{"talk", "consult"}, {"meeting"}}');
```

Por defecto Postgres utiliza la convención de «numeración basada en uno» para las matrices, esto es, una matriz de *n* elementos comienza con *array[1]* y finaliza con *array[n]*. Ahora, podemos hacer algunas consultas sobre SAL_EMP. Primero, mostramos cómo acceder a un elemento de una de las matrices a la vez. Esta consulta recupera los nombres de los empleados cuyos pagos cambiaron en el segundo trimestre:

```
SELECT name  
FROM SAL_EMP  
WHERE SAL_EMP.pay_by_quarter[1] <>  
SAL_EMP.pay_by_quarter[2];
```

```
+-----+  
|name  |  
+-----+  
|Carol |  
+-----+
```

La siguiente consulta recupera el pago del tercer trimestre de todos los empleados:

```
SELECT SAL_EMP.pay_by_quarter[3] FROM SAL_EMP;
```

```
+-----+
|pay_by_quarter |
+-----+
|10000          |
+-----+
|25000          |
+-----+
```

También podemos acceder arbitrariamente a distintas porciones de la matriz o submatrices. Esta consulta recupera el primer elemento de la agenda de Bill para los primeros dos días de la semana.

```
SELECT SAL_EMP.schedule[1:2][1:1]
       FROM SAL_EMP
       WHERE SAL_EMP.name = 'Bill';
```

```
+-----+
|schedule          |
+-----+
|{"meeting"}, {" "}|
+-----+
```

Capítulo 9. Herencia

Creemos dos clases. La clase `capitals` contiene las capitales de los estados que son también ciudades. Naturalmente, la clase `capitals` debe heredar de `cities`.

```
CREATE TABLE cities (  
    name          text,  
    population     float,  
    altitude       int    - (in ft)  
);  
  
CREATE TABLE capitals (  
    state          char(2)  
) INHERITS (cities);
```

En este caso, una instancia de `capitals` *hereda* (*inherits*) todos los atributos (`name`, `population`, `altitude`) de la clase `cities`. El tipo del atributo `name` es `text`, un tipo de dato nativo de Postgres para cadenas ASCII de longitud variable. El tipo del atributo `population` es `float`, un tipo de datos, también nativo, para números de punto flotante de doble precisión. Además `capitals` tiene un atributo extra, `state`, que muestra el estado al que pertenece. En Postgres una clase puede heredar de ninguna o varias otras clases, y una consulta puede hacer referencia tanto a todas las instancias de una clase como a todas las instancias de sus descendientes.

Nota: En realidad, la jerarquía de la herencia es un gráfico dirigido y acíclico.

Por ejemplo, la siguiente consulta encuentra todas las ciudades situadas a una altitud de 500 pies o más:

```
SELECT name, altitude  
FROM cities  
WHERE altitude > 500;
```

```
+-----+-----+  
|name      | altitude |  
+-----+-----+  
|Las Vegas | 2174     |  
+-----+-----+  
|Mariposa  | 1953     |  
+-----+-----+
```

Por otro lado, para encontrar los nombres de todas las ciudades, incluyendo las capitales de estado, que están localizadas a un altitud por encima de los 500 pies, la consulta sería:

```
SELECT c.name, c.altitude  
FROM cities* c  
WHERE c.altitude > 500;
```

Lo que devuelve lo siguiente:

```
+-----+-----+  
|name      | altitude |  
+-----+-----+  
|Las Vegas | 2174     |  
+-----+-----+
```

```
|Mariposa | 1953 |
+-----+-----+
|Madison  | 845  |
+-----+-----+
```

Aquí, el `""` después de `cities` indica que la consulta debe realizarse sobre `cities` y todas las clases que estén por debajo de ella en la jerarquía de herencia. Muchas de las órdenes que ya hemos analizado (**SELECT**, **UPDATE** y **DELETE**) permiten la utilización de `""`, así como otros, como pueden ser **ALTER TABLE**.

Capítulo 10. Multi-Version Concurrency Control (Control de la Concurrency Multi Versión)

Multi-Version Concurrency Control (MVCC) es una técnica avanzada para mejorar las prestaciones de una base de datos en un entorno multiusuario. Vadim Mikheev¹ ha proporcionado la implementación para Postgres.

Introducción

A diferencia de la mayoría de otros sistemas de bases de datos que usan bloqueos para el control de concurrencia, Postgres mantiene la consistencia de los datos un modelo multiversión. Esto significa que mientras se consulta una base de datos, cada transacción ve una imagen de los datos (una *versión de la base de datos*) como si fuera tiempo atrás, sin tener en cuenta el estado actual de los datos que hay por debajo. Esto evita que la transacción vea datos inconsistentes que pueden ser causados por la actualización de otra transacción concurrente en la misma fila de datos, proporcionando *aislamiento transaccional* para cada sesión de la base de datos.

La principal diferencia entre multiversión y el modelo de bloqueo es que en los bloqueos MVCC derivados de una consulta (lectura) de datos no entran en conflicto con los bloqueos derivados de la escritura de datos y de este modo la lectura nunca bloquea la escritura y la escritura nunca bloquea la lectura.

Aislamiento transaccional

El estándar ANSI/ISO SQL define cuatro niveles de aislamiento transaccional en función de tres hechos que deben ser tenidos en cuenta entre transacciones concurrentes. Estos hechos no deseados son:

lecturas "sucias"

Una transacción lee datos escritos por una transacción no esperada, no cursada.

lecturas no repetibles

Una transacción vuelve a leer datos que previamente había leído y encuentra que han sido modificados por una transacción cursada.

lectura "fantasma"

Una transacción vuelve a ejecutar una consulta, devolviendo un conjunto de filas que satisfacen una condición de búsqueda y encuentra que otras filas que satisfacen la condición han sido insertadas por otra transacción cursada.

Los cuatro niveles de aislamiento y sus correspondientes acciones se describen más abajo.

Tabla 10-1. Niveles de aislamiento de Postgres

	Lectura "sucia"	Lectura no repetible	Lectura "fantasma"
--	-----------------	----------------------	--------------------

	Lectura "sucia"	Lectura no repetible	Lectura "fantasma"
Lectura no cursada	Posible	Posible	Posible
Lectura cursada	No posible	Posible	Posible
Lectura repetible	No posible	No posible	Posible
Serializable	No posible	No posible	No posible

Postgres ofrece lectura cursada y niveles de aislamiento serializables.

Nivel de lectura cursada

Lectura cursada es el nivel de aislamiento por defecto en Postgres. Cuando una transacción se ejecuta en este nivel, la consulta sólo ve datos cursados antes de que la consulta comenzara y nunca ve ni datos "sucios" ni los cambios en transacciones concurrentes cursados durante la ejecución de la consulta.

Si una fila devuelta por una consulta mientras se ejecuta una declaración **UPDATE** (o **DELETE**, o **SELECT FOR UPDATE**) está siendo actualizada por una transacción concurrente no cursada, entonces la segunda transacción que intente actualizar esta fila esperará a que la otra transacción se curse o pare. En caso de que pare, la transacción que espera puede proceder a cambiar la fila. En caso de que se curse (y si la fila todavía existe, por ejemplo, no ha sido borrada por la otra transacción), la consulta será reejecutada para esta fila y se comprobará que la nueva fila satisface la condición de búsqueda de la consulta. Si la nueva versión de la fila satisface la condición, será actualizada (o borrada, o marcada para ser actualizada).

Tenga en cuenta que los resultados de la ejecución de **SELECT** o **INSERT** (con una consulta) no se verán afectados por transacciones concurrentes.

Nivel de aislamiento serializable

La *serialización* proporciona el nivel más alto de aislamiento transaccional. Cuando una transacción está en el nivel serializable, la consulta sólo ve los datos cursados antes de que la transacción comience y nunca ve ni datos sucios ni los cambios de transacciones concurrentes cursados durante la ejecución de la transacción. Por lo tanto, este nivel emula la ejecución de transacciones en serie, como si las transacciones fueran ejecutadas una detrás de otra, en serie, en lugar de concurrentemente.

Si una fila devuelta por una consulta durante la ejecución de una declaración **UPDATE** (o **DELETE**, o **SELECT FOR UPDATE**) está siendo actualizada por una transacción concurrente no cursada, la segunda transacción que trata de actualizar esta fila esperará a que la otra transacción se curse o pare. En caso de que pare, la transacción que espera puede proceder a cambiar la fila. En el caso de una transacción concurrente se curse, una transacción serializable será parada con el mensaje

```
ERROR:  Can't serialize access due to concurrent update
```

porque una transacción serializable no puede modificar filas cambiadas por otras transacciones después de que la transacción serializable haya empezado.

Nota: Tenga en cuenta que los resultados de la ejecución de **SELECT** o **INSERT** (con una consulta) no se verán afectados por transacciones concurrentes.

Bloqueos y tablas

Postgres ofrece varios modos de bloqueo para controlar el acceso concurrente a los datos en tablas. Algunos de estos modos de bloqueo los adquiere Postgres automáticamente antes de la ejecución de una declaración, mientras que otros son proporcionados para ser usados por las aplicaciones. Todos los modos de bloqueo (excepto para AccessShareLock) adquiridos en una transacción se mantienen hasta la duración de la transacción.

Además de bloqueos, también se usa compartición en exclusiva para controlar accesos de lectura/escritura a las páginas de tablas en un buffer compartido. Este método se pone en marcha inmediatamente después de que un tuplo es traído o actualizado.

Bloqueos a nivel de tabla

AccessShareLock

Un modo de bloqueo adquirido automáticamente sobre tablas que están siendo consultadas. Postgres libera estos bloqueos después de que se haya ejecutado una declaración.

Conflictos con AccessExclusiveLock.

RowShareLock

Adquirido por **SELECT FOR UPDATE** y **LOCK TABLE** para declaraciones `IN ROW SHARE MODE`.

Entra en conflictos con los modos ExclusiveLock y AccessExclusiveLock.

RowExclusiveLock

Lo adquieren **UPDATE**, **DELETE**, **INSERT** y **LOCK TABLE** para declaraciones `IN ROW EXCLUSIVE MODE`.

Choca con los modos ShareLock, ShareRowExclusiveLock, ExclusiveLock y AccessExclusiveLock.

ShareLock

Lo adquieren **CREATE INDEX** y **LOCK TABLE** para declaraciones `IN SHARE MODE`.

Está en conflicto con los modos RowExclusiveLock, ShareRowExclusiveLock, ExclusiveLock y AccessExclusiveLock.

ShareRowExclusiveLock

Lo toma **LOCK TABLE** para declaraciones `IN SHARE ROW EXCLUSIVE MODE`.

Está en conflicto con los modos RowExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock y AccessExclusiveLock.

ExclusiveLock

Lo toma **LOCK TABLE** para declaraciones `IN EXCLUSIVE MODE`.

Entra en conflicto con los modos RowShareLock, RowExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock y AccessExclusiveLock.

AccessExclusiveLock

Lo toman **ALTER TABLE**, **DROP TABLE**, **VACUUM** y **LOCK TABLE**.

Choca con RowShareLock, RowExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock y AccessExclusiveLock.

Nota: Sólo AccessExclusiveLock bloquea la declaración **SELECT** (sin `FOR UPDATE`).

Bloqueos a nivel de fila

Este tipo de bloqueos se producen cuando campos internos de una fila son actualizados (o borrados o marcados para ser actualizados). Postgres no retiene en memoria ninguna información sobre filas modificadas y de este modo no tiene límites para el número de filas bloqueadas sin incremento de bloqueo.

Sin embargo, tenga en cuenta que **SELECT FOR UPDATE** modificará las filas seleccionadas marcándolas, de tal modo que se escribirán en el disco.

Los bloqueos a nivel de fila no afecta a los datos consultados. Estos son usados para bloquear escrituras *a la misma fila* únicamente.

Bloqueo e índices

Aunque Postgres proporciona desbloqueo para lectura/escritura de datos en tablas, no ocurre así para cada método de acceso al índice implementado en en Postgres.

Los diferentes tipos de índices son manejados de la siguiente manera:

Índices GiST y R-Tree

Nivel de bloqueo de índice del tipo Compartición/exclusividad para acceso lectura/escritura. El bloqueo tiene lugar después de que la declaración se haya ejecutado.

Índices hash

Se usa el bloqueo a nivel de página para acceso lectura/escritura. El bloqueo tiene lugar después de que la página haya sido procesada.

Los bloqueos a nivel de página producen mejor concurrencia que los bloqueos a nivel de índice pero pueden provocar "puntos muertos".

Btree

Se usan bloqueos a nivel de página de compartición/exclusividad en los accesos de lectura/escritura. Los bloqueos se llevan a cabo inmediatamente después de que el tuplo índice sea insertado o buscado.

Los índices Btree proporciona la más alta concurrencia sin provocar "estados muertos".

Chequeos de consistencia de datos en el nivel de aplicación

Ya que las lecturas en Postgres no bloquean los datos, sin tener en cuenta el nivel de aislamiento de la transacción, los datos leídos por una transacción pueden ser sobrescritos por otra. En otras palabras, si una fila es devuelta por **SELECT** esto no significa que esta fila realmente exista en el momento en que se devolvió (un tiempo después de que la declaración o la transacción comenzaran, por ejemplo) ni que la fila esté protegida de borrados o actualizaciones por la transacción concurrente antes de que ésta se lleve a cabo o se pare.

Para asegurarse de la existencia de una fila y protegerla contra actualizaciones concurrentes, debería usar **SELECT FOR UPDATE** o una declaración de tipo **LOCK TABLE** más apropiada. Esto debe tenerse en cuenta cuando desde otros entornos se estén portando aplicaciones hacia Postgres utilizando el modo serializable.

Nota: Antes de la versión 6.5 Postgres usaba bloqueos de lectura, así que la consideración anterior es también válida cuando actualice a 6.5 (o superior) desde versiones anteriores de Postgres.

Notas

1. <mailto:vadim@krs.ru>

Capítulo 11. Configurando su entorno

Esta sección trata sobre cómo configurar su propio entorno, de modo que pueda usar aplicaciones de interfaz de usuario. Se asume que Postgres ha sido correctamente instalado y arrancado. Consulte la Guía del Administrador y las notas de instalación para ver cómo instalar Postgres.

Postgres es una aplicación cliente/servidor. Como usuario, usted sólo necesita acceso a la parte cliente de la instalación (un ejemplo de aplicación cliente es el monitor interactivo `psql`). Para simplificar las cosas asumiremos que Postgres se ha instalado en el directorio `/usr/local/pgsql`. Sin embargo, donde vea el directorio `/usr/local/pgsql` debería sustituirlo por el nombre del directorio donde Postgres esté realmente instalado. Todos los comandos Postgres se instalan en el directorio `/usr/local/pgsql/bin`. Tenga en cuenta que debe añadir este directorio al path de su shell. Si utiliza una variante del Berkeley C shell, tal como `csh` o `tcsh`, debería añadir

```
set path = ( /usr/local/pgsql/bin path )
```

en el fichero `.login` de su directorio personal. Si usa una variante del Bourne shell, como `sh`, `ksh` o `bash`, deberá añadir

```
$ PATH=/usr/local/pgsql/bin:$PATH
$ export PATH
```

al fichero `.profile` en su directorio personal. De ahora en adelante asumiremos que que ha añadido el directorio bin de Postgres a su path. Además, haremos frecuentemente referencia a “configurar una variable del shell” o “configurar una variable de entorno” a lo largo de este documento. Si no entiende completamente el último párrafo sobre cómo modificar su path de búsqueda, debería consultar las páginas del manual de Unix que describen su shell antes de continuar.

Si el administrador de su sitio no configuró las cosas como vienen por defecto, quizás tenga que realizar alguna tarea más. Por ejemplo, si el servidor de bases de datos es una máquina remota, necesitará especificar el valor de la variable de entorno `PGHOST` con el nombre de la máquina que sirve la base de datos. La variable de entorno `PGPORT` puede también ser necesaria. La cuestión de fondo es esta; usted intenta arrancar una aplicación y recibe el mensaje de error que dice que no puede conectar con el `postmaster`. Debería consultar inmediatamente con el administrador de su sitio para asegurarse que su entorno está correctamente configurado.

Capítulo 12. Administración de una Base de Datos

Nota: Actualmente esta sección es una copia disfrazada del tutorial. Será necesario ampliarla. - thomas 1998-01-12

a pesar de que el *administrador local* es responsable por la gestión general de la instalación de Postgres, algunas bases de datos instaladas pueden ser administradas por otra persona, llamada el *administrador de la base de datos*. La responsabilidad de la administración se delega en el momento en que se crea la base de datos. A un usuario se le puede dar privilegio para crear nuevas bases de datos y/o nuevos usuarios. Un usuario que tenga los dos tipos de privilegio puede realizar la mayoría de las labores administrativas en Postgres, pero normalmente no tendrá los mismos privilegios de sistema operativo que el administrador local.

La Guía del Administrador del PostgreSQL trata estos tópicos con mas detalle.

Creación de Bases de Datos

Las bases de datos se crean dentro de Postgres con el comando **create base-de-datos**. `createdb` es un utilitario hecho para suministrar la misma función fuera de Postgres, a partir de la línea de comandos.

El motor de Postgres debe estar corriendo para que cualquiera de los dos métodos funcione, y el usuario que da el comando debe ser el *supe-usuario* de Postgres, o haber obtenido privilegio por parte del super-usuario para crear bases de datos.

Para crear una base de datos llamada "mibd" a partir de la línea de comandos, escriba

```
% createdb mibd
```

y para obtener el mismo resultado dentro de `psql` escriba

```
* CREATE DATABASE mibd;
```

Si no tiene el privilegio necesario para crear una base de datos, verá el siguiente mensaje:

```
% createdb mibd
WARN:user "your username" is not allowed to create/destroy databases
createdb: database creation failed on mibd.
```

Postgres le permite crear cualquier número de bases de datos en un servidor y usted será automáticamente el administrador de la base de datos que acaba de crear. Los nombres de las bases de datos deben comenzar por una letra y están limitados a una longitud total de 32 caracteres.

Ubicaciones Alternativas de las Bases de Datos

Es posible crear una base de datos en un lugar diferente del que fue destinado para el efecto durante la instalación. Recuerde que cualquier consulta a la base de datos es hecha realmente a través del motor de la base de datos, de manera que el lugar donde sea creada la base de datos debe permitir el acceso al motor.

Las ubicaciones alternativas de bases de datos se crean y son referidas por medio de una variable de estado que da el camino absoluto al lugar donde se almacenará la base de datos. Esta variable de estado debe haber sido definida antes de arrancar el motor y el lugar para donde apunta debe permitir escritura desde la cuenta del administrador postgres. Consulte con el administrador local sobre ubicaciones preconfiguradas para bases de datos. Se puede usar cualquier nombre de variable válido para indicar locales alternativos, aunque se recomienda usar nombres de variables con el prefijo "PGDATA" para evitar confusiones con otras variables.

Nota: En versiones antiguas de Postgres, también se permitía el uso de nombres absolutos de fichero para especificar diferentes locales de almacenamiento. Aunque es preferible el uso de variables de estado ya que da mayor flexibilidad al administrador local para gestionar el espacio en disco, también es posible usar caminos absolutos para especificar ubicaciones alternativas. La Guía del Administrador discute como activar esta funcionalidad.

Por razones de seguridad y de integridad, a cualquier camino o variable de estado dada se le agregan al final algunos caminos adicionales. Las ubicaciones alternativas deben ser preparadas ejecutando `initlocation`.

Para crear un área de almacenamiento usando la variable `PGDATA2` (que para este ejemplo tiene el valor `/alt/postgres`), asegúrese que `/alt/postgres` existe y se puede escribir en él a partir de la cuenta del administrador de Postgres. Posteriormente, desde la línea de comandos, escriba

```
% initlocation $PGDATA2
Creating Postgres database system directory /alt/postgres/data
Creating Postgres database system directory /alt/postgres/data/base
```

Para crear una base de datos en el área de almacenamiento alternativa `PGDATA2`, a partir de la línea de comandos, use el siguiente comando:

```
% createdb -D PGDATA2 mibd
```

y para hacer lo mismo a partir de `psql` escriba

```
* CREATE DATABASE mibd WITH LOCATION = 'PGDATA2';
```

Si no tiene el privilegio necesario para crear bases de datos, verá el siguiente mensaje:

```
% createdb mibd
WARN:user "your username" is not allowed to create/destroy databases
createdb: database creation failed on mibd.
```


Si el local elegido no existe o el motor de la base de datos no tiene autorización para entrar en el o escribir en subdirectorios, verá lo siguiente:

```
% createdb -D /alt/postgres/data mibd
ERROR:  Unable to create database directory /alt/postgres/data/base/mydb
createdb: database creation failed on mibd.
```

Acceso a una Base de Datos

Una vez haya creado una base de datos, puede accederla de las siguientes formas:

- ejecutando los programas monitores de Postgres (Por ejemplo `psql`) que le permite introducir, editar y ejecutar comandos SQL interactivamente.)
- escribiendo un programa en C que use la librería de subrutinas LIBPQ. Esta le permite enviar comandos SQL desde C y recibir los resultados y mensajes de vuelta en su programa. Esta interfaz se discute mas ampliamente en la sección ??.

Puede querer arrancar `psql` para experimentar los ejemplos en este manual. El `psql` puede ser activado para la base de datos `mibd` escribiendo el comando:

```
% psql mibd
```

Será saludado con el siguiente mensaje:

```
Welcome to the POSTGRESQL interactive sql monitor:
Please read the file COPYRIGHT for copyright terms of POSTGRESQL

type \? for help on slash commands
type \q to quit
type \g or terminate with semicolon to execute query
You are currently connected to the database: mibd

mibd=>
```

Este símbolo indica que el monitor lo escucha y que puede escribir pedidos SQL dentro de un área de trabajo que mantiene el monitor. El programa `psql` responde a códigos de escape que comiencen con la barra invertida, “\”. Por ejemplo, puede obtener ayuda sobre la sintaxis de varios comandos SQL de Postgres por medio de:

```
mibd=> \h
```

Una vez termine de introducir sus consultas en el área de trabajo, puede pasar el contenido al servidor de Postgres escribiendo:

```
mibd=> \g
```

Esto le dice al servidor que debe procesar su pedido. Si termina su pedido con punto y coma, no necesita el comando “\g”. `psql` procesará automáticamente los pedidos que terminen con punto y coma. Para leer peticiones a partir de un fichero, digamos `miFichero`, en vez de introducir las interactivamente, escriba:

```
mibd=> \i miFichero
```

Para salir de `psql` y regresar a Unix, escriba

```
mibd=> \q
```

y `psql` finalizará y lo hará regresar a su shell de comandos. (Para ver otros comandos de `psql`, escriba `\h` mientras ejecuta `psql`.) En los pedidos SQL se puede usar libremente espacio en blanco (espacio, tabuladores nuevas líneas). Comentarios de una línea se indican con `--`. Todo lo que aparezca después de las dos rayas y hasta el fin de la línea será ignorado. Para comentarios de varias líneas o dentro de una línea se usa `/* ... */`

Privilegios para Bases de Datos

Privilegios para Tablas

TBD

Destrucción de una Base de Datos

Si usted es el administrador de la base de datos `mibd`, puede destruirla usando el siguiente comando Unix:

```
% dropdb mibd
```

Esto retira físicamente todos los ficheros Unix asociados con la base de datos y no podrán ser recuperados, de manera que debe ser hecho con mucha premeditación.

Capítulo 13. Almacenamiento en disco

Esta sección necesita ser escrita. Encontrará algo de información en la FAQ. ¿Voluntarios? - thomas 1998-01-11

Capítulo 14. Instrucciones SQL

Esta es la información de referencia para las instrucciones de SQL soportadas por Postgres.

ABORT

Nombre

ABORT — Aborta la transaccion en curso

Synopsis

```
ABORT [ WORK | TRANSACTION ]
```

Inputs

None.

Outputs

```
ROLLBACK
```

Mensaje devuelto si es completado con éxito.

```
NOTICE: ROLLBACK: no transaction in progress ROLLBACK
```

Si no hay transacciones en curso actualmente.

Descripcion

ABORT deshace la transaccion en curso y causa que todas las actualizaciones hechas por la transaccion sean descartadas. Este comando es identico en comportamiento al comando **ROLLBACK** de SQL92 y esta presente solamente por razones historicas.

Notas

Utilice **COMMIT** para terminar exitosamente una transaccion.

Utilizacion

Para abortar todos los cambios:

```
ABORT WORK;
```

Compatibilidad

SQL92

Este comando es una extension PostgreSQL presente por razones historicas, **ROLL-BACK** es el comando SQL92 equivalente.

MODIFICAR GRUPO

Nombre

MODIFICAR GRUPO — Añadir usuarios a un grupo, eliminar usuarios de un grupo

Synopsis

```
MODIFICAR GRUPO nombre AÑADIR USUARIO nombre de usuario [, ... ]
MODIFICAR GRUPO nombre ELIMINAR USUARIO nombre de usuario [, ... ]
```

Entradas

nombre

El nombre del grupo a modificar.

nombre de usuario

Usuarios que van a ser añadidos o eliminados del grupo. Los nombres de usuarios deben existir.

Resultados

```
MODIFICAR GRUPO
```

Mensaje recibido si la variación fue correcta.

Descripción

MODIFICAR GRUPO se usa para cambiar el añadir usuarios a un grupo o eliminarlos de un grupo. Sólo los administradores de bases de datos pueden usar esta orden. Añadir un usuario a un grupo no crea ese usuario. Igualmente, eliminar a un usuario de un grupo no significa que se elimine al usuario en si mismo.

Usar *CREATE GROUP* para crear un grupo nuevo y *DROP GROUP* para eliminar un grupo.

Forma de uso

Añadir usuarios a un grupo:

```
MODIFICAR GRUPO personal AÑADIR USUARIO karl, john
```

Eliminar un usuario de un grupo

```
MODIFICAR GRUPO trabajadores ELIMINAR USUARIO beth
```

Compatibilidad

SQL92

No existe la orden **MODIFICAR GRUPO** en SQL92. El concepto de reglas es similar.

MODIFICAR TABLA

Nombre

MODIFICAR TABLA — Propiedades de las modificaciones de tablas

Synopsis

```
MODIFICAR TABLA tabla [ * ]
    AÑADIR [ COLUMNNA ] columna tipo
MODIFICAR TABLA tabla [ * ]
    MODIFICAR [ COLUMNNA ] columna { SET DEFAULT valor | DROP DEFAULT }
MODIFICAR TABLA tabla [ * ]
    RENOMBRAR [ COLUMNNA ] columna A nueva columna
MODIFICAR TABLA tabla
    RENOMBRAR A nueva tabla
```

Entradas

tabla

El nombre de un tabla existente para modificarla.

columna

Nombre de una columna nueva o ya existente.

tipo

Tipo de la nueva columna.

nueva columna

Nuevo nombre para una columna ya existente.

nueva tabla

Nuevo nombre para la tabla.

Resultados

MODIFICAR

Mensaje recibido de la columna o la tabla que se ha renombrado.

ERROR

Mensaje recibido si la tabla o la columna no son válidas.

Descripción

MODIFICAR TABLA cambia la definición de una tabla existente. La orden **AÑADIR COLUMNA** añade una nueva columna a la tabla usando la misma sintaxis que **CREATE TABLE**. La orden **MODIFICAR COLUMNA** le permite poner o eliminar los valores por defecto de la columna. Notese que los valores por defecto sólo se aplicaran a las líneas que inserte nuevas. La cláusula **RENOMBRAR** causa que el nombre de una tabla o de una columna cambie sin que se modifique ninguno de los datos contenidos en la tabla afectada. De este modo, la tabla o la columna permanecerá del mismo tipo y tamaño después de que este comando sea ejecutado.

Usted debe ser el creador de esta tabla para poder cambiar su esquema.

Notas

The palabra clave **COLUMNA** es muy usada por lo que se debe omitir.

“*” siguiendo a un nombre de una talba indica que la orden debe ejecutarse sobre esa tabla y todas las tablas que esten bajo ella en la jerarquía subseguente; por defecto, el atributo no será añadido a o renombrado en ninguna de las subclases. Esto siempre

se debe hacer cuando se añade o modifica un atributo en una superclase. Si no es así, las preguntas en la jerarquía subsecuente como

```
SELECCIONAR nueva columna DESDE SuperClase*
```

no funcionarán porque las subclases habrán perdido un atributo que se encontraba en la superclase.

En la presente implementación, las cláusulas por defecto y limitadoras para la nueva columna serán ignoradas. Usted puede usar la orden `PONER VALORES POR DEFECTO` de **MODIFICAR TABLA** para poner los valores por defecto más tarde. (Usted tendrá también que actualizar las líneas existentes a los nuevos valores por defecto, usando `UPDATE`.)

Usted debe ser el creador de la clase para poder cambiar el esquema. Renombrar cualquier parte de un esquema del catálogo de un sistema no está permitido. La *Guía del Usuario de PostgreSQL* tiene más información de las herencias subsecuentes.

Diríjase a **CREAR TABLA** para una descripción más amplia de los argumentos válidos.

Moda de uso

Para añadir a una columna de tipo `VARCHAR` a una tabla:

```
MODIFICAR TABLA distribuidores AÑADIR COLUMNA direcciones VARCHAR(30);
```

Para renombrar una columna existente:

```
MODIFICAR TABLA distribuidores RENOMBRAR COLUMNA direcciones A ciudad;
```

Para renombrar una tabla existente:

```
MODIFICAR TABLA distribuidores RENOMBRA A proveedores;
```

Compatibilidad

SQL92

La orden `AÑADIR COLUMNA` está asumida con la excepción de que no soporta los valores por defecto y limitaciones, como se explicó más arriba. La orden `MODIFICAR COLUMNA` está en sumisión completa.

SQL92 especifica algunas capacidades adicionales para **MODIFICAR TABLA** órdenes que no están todavía directamente soportadas por PostgreSQL:

```
MODIFICAR TABLA tabla AÑADIR definición de limitación de tabla
MODIFICAR TABLA tabla ELIMINAR LIMITACION limitación { RESTRICT | CAS-
CADE }
```

Añadir o eliminar una limitación de tabla (como una limitación de comprobación, limitación única, o limitación de orden extraña). Para crear o eliminar una limitación única, crear o eliminar un índice único, respectivamente (ver *CREATE INDEX*). Para cambiar otras clase de limitaciones necesita recrear y recargar la tabla usando otros parámetros para la *CREATE TABLE* orden.

Por ejemplo, para eliminar cualquier limitación en una tabla distribuidores:

```
CREAR TABLA temp COMO SELECCIONAR * DESDE distribuidores;
ELIMINAR TABLA distribuidores;
CREAR TABLA distribuidores COMO SELECCIONAR * DESDE temp;
ELIMINAR TABLA temp;
```

```
MODIFICAR TABLA tabla ELIMINAR [ COLUMNA ] columna { RESTRICT | CASCA-
DE }
```

Eliminar una columna de una tabla. Corrientemente, para eliminar una columan existente la tabla debe ser recreada y recargada:

```
CREAR TABLA temp COMO SELECCIONAR did, ciudad DESDE distribuidores;
ELIMINAR TABLA distribuidores;
CREAR TABLA distribuidores (
    did      DECIMAL(3)  DEFAULT 1,
    name     VARCHAR(40) NOT NULL,
);
INSERTAR DENTRO distribuidores SELECCIONAR * DESDE temp;
ELIMINAT TABLA temp;
```

Las clausulas para renombrar columnas y tablas son extensiones para PostgreSQL SQL92 no las provee.

MODIFICAR USUARIO

Nombre

MODIFICAR USUARIO — Modificar la información de la cuenta de usuario

Synopsis

```
MODIFICAR USUARIO nombre de usuario
  [ WITH PASSWORD 'palabra clave' ]
  [ CREATEDB | NOCREATEDB ] [ CREATEUSER | NOCREATEUSER ]
  [ VALID UNTIL 'abstime' ]
```

Entradas

nombre de usuario

El nombre del usuario cuyos detalles van a ser modificados.

palabra clave

La nueva palabra clave que va a ser usada en esta cuenta.

CREATEDB

NOCREATEDB

Estas clausulas definen la capacidad de un usuario para crear bases de datos. Si se especifica CREATEDB, el usuario podrá definir sus propias bases de datos. Usando NOCREATEDB se deniega a un usuario la capacidad de crear bases de datos.

CREATEUSER

NOCREATEUSER

Estas clausulas determinan si un usuario está autorizado a crear nuevos usuarios él mismo. Esta opción hace ser además al usuario un superusuario que puede pasar por encima de todas las restricciones de acceso.

abstime

La fecha (y, opcionalmente, la hora) en la que la palabra clave de este usuario expirará.

Resultados

MODIFICAR USUARIO

Mensaje recibido si la modificación es correcta.

ERROR: MODIFICAR USUARIO: usuario "nombre de usuario" no existe

Mensaje de error recibido si el usuario especificado no existe en la base de datos.

Descripción

MODIFICAR USUARIO se usa para cambiar los atributos de la cuenta de un usuario de PostgreSQL. Sólo un superusuario de una base de datos puede cambiar privilegios y fechas de caducidad de palabras clave con esta orden. Ordinariamente los usuarios sólo pueden cambiar su propia palabra clave.

Usar *CREAR USUARIO* para crear un nuevo usuario y *DROP USER* para eliminar un usuario.

Modo de uso

Cambiar la palabra clave de un usuario:

```
MODIFICAR USUARIO davide CON PALABRA CLAVE 'hu8jmn3';
```

Cambiar la validez de un usuario hasta la fecha

```
MODIFICAR USUARIO manuel VALIDO HASTA '31 En 2030';
```

Cambiar la validez de un usuario hasta la fecha, especificando que su autorización expirara al mediodía del 4 de Mayo de 1998 usando la zona horaria que tiene 1 hora más que el UTC

```
MODIFICAR USUARIO chris VALIDO HASTA '4 May 12:00:00 1998 +1';
```

Dar a un usuario la capacidad de crear otros usuarios y nuevas bases de datos.

```
MODIFICAR USUARIO miriam CREATEUSER CREATEDB;
```

Compatibilidad

SQL92

No hay orden **MODIFICAR USUARIO** en SQL92. El standar deja la definición de usuarios a la implementación.

BEGIN

Nombre

BEGIN — Comienza una transaccion en modo encadenado

Synopsis

```
BEGIN [ WORK | TRANSACTION ]
```

Inputs

WORK
TRANSACTION

Palabras clave opcionales. No tienen efecto.

Outputs

BEGIN

esto significa que una nueva transaccion ha sido comenzada.

NOTICE: BEGIN: already a transaction in progress

Esto indica que una transaccion ya esta en progreso. La transaccion en curso no se ve afectada.

Descripcion

Por defecto, PostgreSQL ejecuta las transacciones en *modo no encadenado* (tambien conocido como “autocommit” en otros sistemas de base de datos). En otras palabras, cada estado de usuario es ejecutado en su propia transaccion y un commit se ejecuta implicitamente al final del estatuto (si la ejecucion fue exitosa, de otro modo se ejecuta un rollback). **BEGIN** inicia una transaccion de usuario en modo encadenado, i.e. todos los estados de usuarios despues de un comando **BEGIN** se ejecutaran en una transaccion unica hasta un explicito **COMMIT**, **ROLLBACK**, o aborte la ejecucion. Los estados en modo encadenado se ejecutan mucho mas rapido, porque la transaccion start/commit requiere una actividad significativa de CPU y de disco. La ejecucion de multiples estados dentro de una transaccion tambien es requerida para la consistencia cuando se cambian muchas tablas relacionadas.

El nivel de aislamiento por defecto de las transacciones en PostgreSQL es **READ COMMITTED**, donde las consultas dentro de la transaccion solo tiene en cuenta los cambios consolidados antes de la ejecucion de la consulta. Asi pues, debes utilizar **SET TRANSACTION ISOLATION LEVEL SERIALIZABLE** justo despues de **BEGIN** si necesitas aislamiento de transacciones mas riguroso. Las consultas del tipo **SERIALIZABLE** solo tendran en cuenta los cambios consolidados antes de que la transaccion entera comience (realmente, antes de la ejecucion del primer estado DML en una transaccion serializable).

Si la transaccion esta consolidada, PostgreSQL asegurara que todas las actualizaciones sean hechas o si no que ninguna de ellas lo sea. Las transacciones tienen la propiedad estandar **ACID** (atomica, consistente, aislada y durable).

Notas

Remitase a *LOCK* para informacion ampliada sobre el bloqueo de tablas durante una transaccion.

Utilice **COMMIT** o **ROLLBACK** para terminar una transaccion.

Utilizacion

Para comenzar una transaccion de usuario:

```
BEGIN WORK;
```

Compatibilidad

SQL92

BEGIN es una extension de lenguaje de PostgreSQL. No hay ningun comando **BEGIN** explicito en SQL92; la iniciacion de una transaccion siempre esta implicita y es terminado o con un estado **COMMIT** o con **ROLLBACK**.

Nota: Muchos sistemas de bases de datos relacionales ofrecen una caracteristica de autocommit como una comodidad.

Por cierto, la palabra **BEGIN** es utilizada para diferentes propositos en SQL embebido. Queda avisado para que sea cuidadoso acerca de las transacciones semanticas cuando traslade aplicaciones de base de datos.

SQL92 tambien requiere **SERIALIZABLE** para ser el nivel de aislamiento de transaccion por defecto.

CLOSE

Nombre

CLOSE — Cierra un cursor

Synopsis

```
CLOSE cursor
```

Inputs

cursor

El nombre de un cursor abierto a cerrar.

Outputs

CLOSE

Mensaje devuelto si el cursor es cerrado exitosamente.

NOTICE PerformPortalClose: portal "*cursor*" not found

Esta alerta se da si el *cursor* no esta declarado o ya ha sido cerrado.

Descripcion

CLOSE libera los recursos asociados con un cursor abierto. Después de que sea cerrado el cursor, no se permiten operaciones subsiguientes en él. Un cursor debería ser cerrado cuando no va a ser necesitado.

Un close implícito es ejecutado para cada cursor abierto cuando una transacción es terminada por un **COMMIT** o un **ROLLBACK**.

Notas

Postgres no tiene un estado de cursor **OPEN** explícito; un cursor se considera abierto cuando es declarado. Utilice el estado **DECLARE** para declarar un cursor.

Utilizacion

Cerrar el cursor *liahoma*:

```
CLOSE liahona;
```

Compatibilidad

SQL92

CLOSE es totalmente compatible con SQL92.

CLUSTER

Nombre

CLUSTER — Proporciona aviso de almacenaje agrupado (clustering) al servidor.

Synopsis

```
CLUSTER indexname ON table
```

Entradas

Nombre del indice

El nombre de un indice.

table

El nombre de una tabla.

Salidas

```
CLUSTER
```

El agrupamiento se hizo exitosamente.

```
ERROR: relation <tablerelation_number> inherits "table"
```

BEGIN RATIONALE: Esto no esta documentado en ningun lugar. Parece que no es posible agrupar una tabla que es heredada. END RATIONALE:

```
ERROR: Relation table does not exist!
```

BEGIN RATIONALE: La relacion especificada no fue mostrada en el mensaje de error, la cual contiene una cadena aleatoria en lugar del nombre de una relación. END RATIONALE:

Descripción

CLUSTER manda a Postgres que agrupe la clase especificada por *table* basandose aproximadamente en el indice especificado por *indexname*. El indice debe haber sido definido ya en *classname*.

Cuando una clase se agrupa, es fisicamente reordenada basandose en la informacion del indice. El agrupamiento es estatico. En otras palabras, mientras que la clase es actualizada, los cambios no son agrupados. No se hace ningun intento de mantener agrupadas nuevas instancias o tuplas actualizadas. Si uno quiere, puede reagruparlas manualmente ejecutando el comando de nuevo.

Notas

La tabla actualmente esta copiada a una tabla temporal con el orden del indice, despues se renombra a su nombre original. Por esta razon, todos los premisos concedidos y otros indices se pierden cuando se ejecuta el agrupamiento (clustering).

En los casos en que accedes a una lineas solas aleatoreamente dentro de una tlabla, el orden actual de los datos en el global de la tabla no es importante. Sin embargo, si

tienes tendencia a acceder a algunos datos mas que a otros, y hay un indice que los agrupa, te beneficiaras del uso de **CLUSTER**.

Otro lugar en el que **CLUSTER** es de ayuda es en los casos en los que utilizas un indice para extraer muchas lineas de una tabla, o un unico valor de un indice tiene multiples lineas con las que coincide, **CLUSTER** ayudara porque una vez el indice identifica el total de paginas (de disco) para la primera linea con la que coincide, todas las otras lineas que coinciden probablemente esten ya en la misma pagina del total, ahorrando accesos a disco y acelerando la consulta.

Hay dos maneras para agrupar datos. La primera es con el comando **CLUSTER**, que reordena la tabla original con la ordenacion del indice que especifiques. Esta puede ser lenta en tablas grandes porque las lineas se van a buscar desde el global de la tabla en orden de indice, y si el global de la tabla esta desordenada, las entradas estan en paginas aleatorias, de este modo hay una pagina de disco recuperada por cada linea movida. Postgres tiene una cache, pero la mayoria de una tabla grande no cabra en la cache.

Otra manera para agrupar datos es utilizar

```
SELECT columnlist INTO TABLE newtable
FROM table ORDER BY columnlist
```

que utiliza el codigo de ordenacion de Postgres en la clausula ORDER BY para hacer coincidir los indices, y que es mucho mas rapido para datos desordenados. Despues borra la tabla vieja, utiliza **ALTER TABLE/RENAME** para renombrar como *temp* la tabla vieja, y recrear cualquier indice. El unico problema es que no se conservan los OID. De ahi en adelante, **CLUSTER** deberia ser rapido porque la mayoria de los datos ya han sido ordenados, y se utiliza el indice existente.

Nota de traductor: Un índice agrupado es aquel que llegado al final de su árbol b-tree no contiene un puntero a una página de disco en la que está la tupla, sino la propia tupla.

Utilizacion

Agrupamiento de la relacion empleados basandose en su atributo salario

```
CLUSTER emp_ind ON emp;
```

Compatibilidad

SQL92

No hay ningun estatuto de lenguaje **CLUSTER** en SQL92.

COMMIT

Nombre

COMMIT — Realiza la transacción actual

Synopsis

COMMIT [WORK | TRANSACTION]

Inputs

WORK
TRANSACTION

Palabra clave opcional. No tiene efecto.

Outputs

COMMIT

Mensaje devuelto si la transacción se realiza con éxito.

NOTICE: COMMIT: no transaction in progress

Si no hay transacciones en progreso.

Description

COMMIT realiza la transacción actual. Todos los cambios realizados por la transacción son visibles a las otras transacciones, y se garantiza que se conservan si se produce una caída de la máquina.

Notes

Las palabras clave **WORK** y **TRANSACTION** son demasiado informativas, y pueden ser omitidas.

Use **ROLLBACK** para abortar una transacción.

Usage

Para hacer todos los cambios permanentes:

```
COMMIT WORK;
```

Compatibilidad

SQL92

SQL92 solo especifica las dos formas, COMMIT y COMMIT WORK. Por lo demás, es totalmente compatible.

COPY

Nombre

COPY — Copia datos entre ficheros y tablas

Synopsis

```
COPY [ BINARY ] table [ WITH OIDS ]
    FROM { 'filename' | stdin }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ]
COPY [ BINARY ] table [ WITH OIDS ]
    TO { 'filename' | stdout }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ]
```

Inputs

BINARY

Cambia el comportamiento del formato de campos, forzando a todos los datos a almacenarse o leerse como objetos binarios, en lugar de como texto.

table

El nombre de una tabla existente.

WITH OIDS

Copia el identificador de objeto interno único (OID) para cada fila.

filename

La ruta absoluta en formato Unix del fichero de entrada o salida.

stdin

Especifica que la entrada viene de un conducto o terminal.

stdout

Especifica que la salida va a un conducto o terminal.

delimiter

UN caracter que delimita los campos de entrada o salida.

null print

Una cadena para representar valores NULL. El valor por defecto es “\N” (backslash-N), por razones históricas. Puede preferir, por ejemplo, una cadena vacía.

Nota: En una copia de entrada, cualquier dato que coincida con esta cadena será almacenado como un valor NULL, por lo que debería asegurarse de usar la misma cadena que usó para la copia de salida-

Outputs

COPY

La copia se completó satisfactoriamente.

ERROR: reason

La copia falló por la razón indicada en el mensaje de error.

Descripción

COPY mueve datos entre tablas de Postgres y ficheros del sistema de archivos estándar. **COPY** indica al servidor Postgres que lea o escriba de o a un fichero. El fichero ha de ser directamente visible para el servidor, y el nombre completo ha de especificarse desde el punto de vista del servidor. Si se especifica *stdin* o *stdout*, los datos van de la aplicación cliente al servidor (o viceversa).

Notes

La palabra clave **BINARY** obliga a que todos los datos se almacenen o lean como objetos binarios en lugar de como texto. Esto es algo más rápido que el comportamiento normal de **COPY** pero el resultado no es generalmente portable, y los ficheros generados son algo más grandes aunque este es un factor que depende de los datos en sí. Por defecto, cuando se copia un texto se usa un tabulador (“\t”) como delimitador. El

delimitador puede cambiarse por cualquier otro caracter empleando la palabra clave **USING DELIMITERS**. Los caracteres dentro de los campos de datos que resulten coincidir con el delimitador serán encerrados entre comillas.

Ha de hacerse primero un *select access* en cualquier tabla cuyos valores sean leídos por **COPY**, y *insert or update access* en la tabla en la que se vayan a insertar los valores. El servidor necesita los permisos Unix adecuados sobre cualquier fichero que vaya a leerse o escribirse con este comando.

la palabra clave **USING DELIMITERS** especifica un caracter que se usará para delimitar entre columnas. Si se especifican varios caracteres en la cadena delimitadora, solo se usará el primer caracter.

Sugerencia: No confunda **COPY** con la instrucción `\copy` de `psql`.

COPY no invoca regla ni acciones por defecto en las columnas. Sin embargo, puede invocar procedimientos disparados.

COPY detiene las operaciones en el primer error. Esto no produce problemas en el caso de **COPY FROM**, pero el destino, por supuesto, será parcialmente modificado en el caso de un **COPY TO**. **VACUUM** puede usarse para limpiar tras una copia fallida.

Debido a que el directorio de trabajo del servidor de Postgres no es normalmente el mismo que el directorio de trabajo del usuario, el resultado de copiar el fichero "foo" (sin añadir información de la ruta) puede dar lugar a resultados inesperados para el usuario inadvertido. En este caso, en lugar de foo, acabamos con \$PGDATA/foo. Por lo general, debería usarse la ruta completa tal como se vería desde el servidor, al especificar los ficheros a copiar.

Los ficheros usados como argumentos para **COPY** deben residir o ser accesible por parte de la máquina servidor de base de datos, en los discos locales o en un sistema de ficheros de red.

Cuando se emplea una conexión TCP/IP, y se especifica un fichero objetivo, dicho fichero se escribirá en la máquina donde se esté ejecutando el servidor, no en la máquina del usuario.

File Formats

Text Format

Cuando se usa **COPY TO** sin la opción **BINARY**, el fichero generado tendrá cada fila (instancia) en una sola línea, con cada una de las columnas (atributo) separada por el caracter delimitador. Los caracteres delimitadores internos (los caracteres internos que coincidan con el delimitador) se precederán del caracter barra atrás (""). Los valores de atributo son cadenas de texto generados por la función de salida asociada con cada uno de los tipos de atributo. La función de salida para un tipo no debería tratar de generar el caracter barra atrás; éste será generado por el comando **COPY**.

El formato para cada instancia es

```
<attr1><separator><attr2><separator>...<separator><attrn><newline>
```

El identificador se situa en el principio de la linea, cuando se especifica WITH OIDS

Si **COPY** envía su salida a la salida estandar en lugar de a un fichero, enviará una barra invertida ("\") y un punto, seguidos de un caracter de salto de linea en una linea separada, cuando termina su salida. Similarmente, si **COPY** está leyendo de una salida estandar, esperará una barra invertida y un punto seguidos por un fin de linea, como los tres primeros caracteres de una linea para indicar el fin del fichero. Sin embargo, **COPY** terminará (y a continuación terminará la aplicación servidor) si se encuentra un EOF antes de que se encuentre esta cadena que indica el fin de fichero.

El caracter barra invertida tiene otros significados especiales. Un caracter barra invertida literal se representa como dos barras consecutivas ("\\"). El caracter tabulador se representa con una barra invertida y un tabulador. EL caracter fin de linea se representa como una barra invertida y un fin de linea. Cuando se cargan datos de texto no generados por Postgres necesitará convertir el caracter barra invertida en un par de barras para asegurar que se carguen adecuadamente. (La secuencia "\N" siempre se interpretará como una barra invertida y un caracter "N", por compatibilidad. La solución más general es "\\N".)

Binary Format

EN el caso de **COPY BINARY**, los primeros cuatro bytes del fichero será el numero de instancias en el fichero. Si el numero es cero, el comando **COPY BINARY** leerá hasta que se encuentre el fin del fichero. En otro caso, dejará de leer cuando se lean ese numero de instancias. Los restantes datos en el fichero se ignorarán.

El formato para cada instancia en el fichero es como sigue. Nótese que este formato debe ser seguido *exactamente*. Las cantidades enteras de cuatro bytes sin signo se denominan uint32 en la tabal que sigue.

Tabla 14-1. Contenidos de un fichero binario de copy

En el principio del fichero	
uint32	numero de tuplas
Para cada tupla	
uint32	Longitud total de la tupla de datos
uint32	identificador (si se especifica)
uint32	numero de atributos nulos
[uint32,...,uint32]	numeros de atributos, contando desde cero
-	<tupla data>

Alineación de datos binarios

Sobre equipos Sun-3s, los atributos de 2 bytes se alinean en grupos de cuatro bytes. Los atributos de caracteres se alinean en grupos de un solo byte. En la mayoría de las otras máquinas, todos los atributos mayores de un byte se alinean en grupos de cuatro bytes. Nótese que los atributos de longitud variable vienen precedidos de la longitud del atributo; las matrices son simplemente cadenas continuas del elemento tipo de la matriz.

Usage

El siguiente ejemplo copia una tabla a la salida estandar, usando una barra vertical como delimitador de campo:

```
COPY country TO stdout USING DELIMITERS '|' ;
```

Para copiar datos de un fichero Unix a la tabla "country":

```
COPY country FROM '/usr1/proj/bray/sql/country_data' ;
```

Ha aquí un ejemplo de datos adecuados para ser copiados a una tabla desde `stdin` (dado que tienen la secuencia de terminación en la última línea):

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
...
ZM      ZAMBIA
ZW      ZIMBABWE
\.
```

Los mismos datos, como salida en formato binario en una máquina Linux/i586. Los datos se muestran tras ser filtrados con el comando Unix `od -c`. La tabla tiene tres campos; el primero es `char(2)` y el segundo es `text`. Todas las filas tienen un valor null en el tercer campo. Nótese como el campo `char(2)` está relleno con nulos hasta alcanzar los cuatro bytes y el campo de texto es precedido por su longitud:

```
355  \0  \0  \0 027  \0  \0  \0 001  \0  \0  \0 002  \0  \0  \0
006  \0  \0  \0  A  F  \0  \0 017  \0  \0  \0  A  F  G  H
    A  N  I  S  T  A  N 023  \0  \0  \0 001  \0  \0  \0 002
    \0  \0  \0 006  \0  \0  \0  A  L  \0  \0  \v  \0  \0  \0  A
    L  B  A  N  I  A 023  \0  \0  \0 001  \0  \0  \0 002  \0
    \0  \0 006  \0  \0  \0  D  Z  \0  \0  \v  \0  \0  \0  A  L
    G  E  R  I  A
...
    \n  \0  \0  \0  Z  A  M  B  I  A 024  \0
    \0  \0 001  \0  \0  \0 002  \0  \0  \0 006  \0  \0  \0  Z  W
    \0  \0  \f  \0  \0  \0  Z  I  M  B  A  B  W  E
```

Compatibility

SQL92

No existe la sentencia **COPY** en SQL 92.

CREATE AGGREGATE

Nombre

CREATE AGGREGATE — Define una nueva función de agregado

Synopsis

```
CREATE AGGREGATE name [ AS ] ( BASETYPE = data_type
    [ , SFUNC1 = sfunc1, STYPE1 = sfunc1_return_type ]
    [ , SFUNC2 = sfunc2, STYPE2 = sfunc2_return_type ]
    [ , FINALFUNC = ffunc ]
    [ , INITCOND1 = initial_condition1 ]
    [ , INITCOND2 = initial_condition2 ] )
```

Entradas

name

El nombre de la función de agregado a crear.

data_type

El tipo de dato fundamental sobre el que opera esta función de agregado.

sfunc1

La función de estado de transición que ha de llamarse para cada campo no nulo desde la columna fuente. Toma una variable del tipo *sfunc1_return_type* como primer argumento y el campo como segundo argumento.

sfunc1_return_type

El tipo devuelto de la primera función de transición.

sfunc2

La función de estado de transición que ha de llamarse para cada campo no nulo de la columna origen. Toma una variable de tipo *sfunc2_return_type* como argumento unico y devuelve una variable del mismo tipo.

sfunc2_return_type

EL tipo devuelto por la segunda función de transición.

ffunc

La función final llamada tras convertir todos los campos de entrada. Esta función debe recibir dos argumentos de los tipos *sfunc1_return_type* y *sfunc2_return_type*.

initial_condition1

El valor inicial para el argumento de la primera función de transición.

initial_condition2

El valor inicial del argumento de la segunda función de transición.

Outputs

CREATE

Mensaje devuelto si el comando se completa satisfactoriamente.

Description

CREATE AGGREGATE permite a un usuario o programador extender la funcionalidad de Postgres definiendo nuevas funciones de agregado. Algunas funciones de agregado para tipos base como `min(int4)` y `avg(float8)` están ya disponibles en la distribución base. Si se definen nuevos tipos o se necesita una función de agregado que no se proporciona, puede usarse el comando **CREATE AGGREGATE** para proporcionar las características deseadas.

Una función de agregados puede requerir hasta tres funciones, dos funciones de transición de estado, *sfunc1* y *sfunc2*:

```
sfunc1( internal-state1, next-data_item ) --> next-internal-state1
sfunc2( internal-state2 ) --> next-internal-state2
```

y una función final de cálculo, *ffunc*:

```
ffunc(internal-state1, internal-state2) --> aggregate-value
```

Postgres crea hasta dos variables temporales (referidas aquí como *temp1* y *temp2*) para mantener resultados intermedios usados como argumentos por las funciones de transición.

Estas funciones de transición han de tener las siguientes propiedades:

- Los argumentos de *sfunc1* deben ser *temp1* del tipo *sfunc1_return_type* y *column_value* de tipo *data_type*. El valor devuelto debe ser del tipo *sfunc1_return_type* y será usado como primer argumento en la próxima llamada a *sfunc1*.
- El argumento y valor devuelto de *sfunc2* ha de ser *temp2* del tipo *sfunc2_return_type*.
- Los argumentos para la función de cálculo final ha de ser *temp1* y *temp2* y su valor devuelto debe ser un tipo base de Postgres (no necesariamente *data_type* que ha sido especificado por `BASETYPE`).
- `FINALFUNC` debe ser especificado si y solo si ambas funciones de transición de estado son especificadas.

Una función de agregado puede requerir solo una o dos condiciones iniciales, una para cada función de transición. Estas se especifican y almacenan en la base de datos como campos de tipo text.

Notes

Use **DROP AGGREGATE** para desechar funciones de agregado.

Es posible especificar funciones de agregado que tengan diversas combinaciones de funciones de estado y funciones finales. Por ejemplo, la función de agregado `count` requiere SFUNC2 (una función de incremento) pero no SFUNC1 o FINALFUNC, mientras que la función de agregado `sum` requiere SFUNC1 (una función de adición) pero no SFUNC2 ni FINALFUNC y la función de agregado `avg` requiere tanto las dos funciones de estado como una FINALFUNC (una función de división) para producir su resultado. En cualquier caso, al menos una de las funciones de estado debe ser definida, y cualquier SFUNC2 debe tener el correspondiente INITCOND2.

Usage

Vease el capítulo de las funciones de agregado en la Guía del Programador (*PostgreSQL Programmer's Guide*) para ejemplos de uso más completos.

Compatibilidad

SQL92

CREATE AGGREGATE es una extensión del lenguaje de Postgres. No existe la orden **CREATE AGGREGATE** en SQL92.

CREATE DATABASE

Nombre

CREATE DATABASE — Crea una nueva base de datos

Synopsis

```
CREATE DATABASE name [ WITH LOCATION = 'dbpath' ]
```

Inputs

name

Le nombre de la base de datos a crear.

dbpath

Una ubicación alternativa para almacenar la nueva base de datos en el sistema de archivos. Ver más adelante posibles problemas.

Outputs

CREATE DATABASE

Mensaje devuelto si la orden se completa satisfactoriamente.

ERROR: user 'username' is not allowed to create/drop databases

Ha de tener el privilegio especial CREATEDB para crear bases de datos. Ver *CREAR USUARIO*.

ERROR: createdb: database "name" already exists

Esto ocurre si una base de datos llamada *name* ya existe.

ERROR: Single quotes are not allowed in database names.

ERROR: Single quotes are not allowed in database paths.

La base de datos *name* y *dbpath* no pueden contener comillas simples. Esto es imprescindible para que los comandos de shell que crean el directorio de la base de datos puedan ejecutarse de modo seguro.

ERROR: The path 'xxx' is invalid.

La expansión del camino especificado *dbpath* ha fallado (ver más abajo el comando). Compruebe la ruta que introdujo o asegúrese de que la variable de entorno a la que ha hecho referencia existe.

ERROR: createdb: May not be called in a transaction block.

Si tiene una transacción de bloques explícita en ejecución no puede llamar a **CREATE DATABASE**. Primero ha de terminarse la transacción.

ERROR: Unable to create database directory 'xxx'.

ERROR: Could not initialize database directory.

Estos mensajes están más bien relacionados con insuficientes permisos sobre el directorio de datos, insuficiente espacio en el disco, u otros problemas en el sistema de ficheros. El usuario bajo el que está corriendo el servidor de base de datos debe tener acceso a la localización especificada.

Description

CREATE DATABASE crea una nueva base de datos PostgreSQL. El creador pasa a ser el propietario de la nueva base de datos.

Puede especificarse una localización alternativa para, por ejemplo, almacenar la base de datos en un disco diferente. La ruta debe haber sido preparada con la orden *initlocation*.

Si la ruta contiene una barra, la parte delantera se interpreta como una variable de entorno, que debe ser conocida por el proceso servidor. De esta forma el administrador de la base de datos puede ejercer control sobre las localizaciones que pueden ser creadas. (Una elección de usuario puede ser, por ejemplo, 'PGDATA2'.) Si el servidor está compilado con `ALLOW_ABSOLUTE_DBPATHS` (cosa que no se hace por defecto), se permiten también los nombres de ruta absolutos, identificados por una barra al principio (p. ej. `'/usr/local/pgsql/data'`).

Notas

CREATE DATABASE es una extensión del lenguaje de Postgres.

Use `drop_database` para eliminar la base de datos.

El programa `createdb` es un script shell construido alrededor de este comando, y que se incluye por cortesía.

Existen aspectos sobre seguridad e integridad de los datos implicados en el uso de localizaciones alternativas para las bases de datos especificados con nombres de ruta absolutos, y por defecto solo una variable de entorno conocida por el proceso servidor puede ser especificada para una localización alternativa. Vea la Guía del administrador para más información.

Uso

Para crear una nueva base de datos:

```
olly=> create database lusiadas;
```

para crear una nueva base de datos en un area alternativa `~/private_db`:

```
$ mkdir private_db
$ initlocation ~/private_db
Creating Postgres database system directory /home/olly/private_db/base

$ psql olly
Welcome to psql, the PostgreSQL interactive terminal.
(Please type \copyright to see the distribution terms of PostgreSQL.)

Type \h for help with SQL commands,
      \? for help on internal slash commands,
      \q to quit,
      \g or terminate with semicolon to execute query.
olly=> CREATE DATABASE elsewhere WITH LOCATION = '/home/olly/private_db';
CREATE DATABASE
```

Compatibilidad

SQL92

No existe el comando **CREATE DATABASE** en SQL92. El comando equivalente en el SQL estandar es **CREATE SCHEMA**.

CREATE FUNCTION

Nombre

CREATE FUNCTION — Defines a new function

Synopsis

```
CREATE FUNCTION name ( [ ftype [, ...] ] )
    RETURNS rtype
    [ WITH ( attribute [, ...] ) ]
    AS definition
    LANGUAGE 'langname'
```

```
CREATE FUNCTION name ( [ ftype [, ...] ] )
    RETURNS rtype
    [ WITH ( attribute [, ...] ) ]
    AS obj_file , link_symbol
    LANGUAGE 'C'
```

Inputs

name

The name of a function to create.

ftype

The data type of function arguments. The input types may be base or complex types, or *opaque*. *opaque* indicates that the function accepts arguments of an invalid type such as `char *`.

rtype

The return data type. The output type may be specified as a base type, complex type, *setof type*, or *opaque*. The *setof* modifier indicates that the function will return a set of items, rather than a single item.

attribute

An optional piece of information about the function, used for optimization. The only attribute currently supported is `iscachable`. `iscachable` indicates that the function always returns the same result when given the same input values (i.e., it does not do database lookups or otherwise use information not directly present in its parameter list). The optimizer uses `iscachable` to know whether it is safe to pre-evaluate a call of the function.

definition

A string defining the function; the meaning depends on the language. It may be an internal function name, the path to an object file, an SQL query, or text in a procedural language.

obj_file, link_symbol

This form of the **AS** clause is used for dynamically-linked, C language functions when the function name in the C language source code is not the same as the name of the SQL function. The string *obj_file* is the name of the file containing the dynamically loadable object, and *link_symbol*, is the object's link symbol which is the same as the name of the function in the C language source code.

langname

may be 'C', 'sql', 'internal' or 'plname', where 'plname' is the name of a created procedural language. See *CREATE LANGUAGE* for details.

Outputs**CREATE**

This is returned if the command completes successfully.

Description

CREATE FUNCTION allows a Postgres user to register a function with a database. Subsequently, this user is treated as the owner of the function.

Notes

Refer to the chapter in the *PostgreSQL Programmer's Guide* on extending Postgres via functions for further information on writing external functions.

Use **DROP FUNCTION** to drop user-defined functions.

Postgres allows function "overloading"; that is, the same name can be used for several different functions so long as they have distinct argument types. This facility must be used with caution for `internal` and C-language functions, however.

Two `internal` functions cannot have the same C name without causing errors at link time. To get around that, give them different C names (for example, use the argument types as part of the C names), then specify those names in the **AS** clause of **CREATE**

FUNCTION. If the AS clause is left empty then **CREATE FUNCTION** assumes the C name of the function is the same as the SQL name.

When overloading SQL functions with C-language functions, give each C-language instance of the function a distinct name, and use the alternative form of the **AS** clause in the **CREATE FUNCTION** syntax to ensure that overloaded SQL functions names are resolved to the correct dynamically linked objects.

A C function cannot return a set of values.

Usage

To create a simple SQL function:

```
CREATE FUNCTION one() RETURNS int4
    AS 'SELECT 1 AS RESULT'
    LANGUAGE 'sql';
SELECT one() AS answer;

    answer
-----
1
```

This example creates a C function by calling a routine from a user-created shared library. This particular routine calculates a check digit and returns TRUE if the check digit in the function parameters is correct. It is intended for use in a CHECK constraint.

```
CREATE FUNCTION ean_checkdigit(bpchar, bpchar) RETURNS bool
    AS '/usr1/proj/bray/sql/funcs.so' LANGUAGE 'c';

CREATE TABLE product (
    id          char(8) PRIMARY KEY,
    eanprefix   char(8) CHECK (eanprefix ~ '[0-9]{2}-[0-9]{5}')
                REFERENCES brandname(ean_prefix),
    eancode     char(6) CHECK (eancode ~ '[0-9]{6}'),
    CONSTRAINT ean    CHECK (ean_checkdigit(eanprefix, eancode))
);
```

This example creates a function that does type conversion between the user defined type complex, and the internal type point. The function is implemented by a dynamically loaded object that was compiled from C source. For Postgres to find a type conversion function automatically, the sql function has to have the same name as the return type, and overloading is unavoidable. The function name is overloaded by using the second form of the **AS** clause in the SQL definition

```
CREATE FUNCTION point(complex) RETURNS point
    AS '/home/bernie/pgsql/lib/complex.so', 'complex_to_point'
    LANGUAGE 'c';
```

The C decalaration of the function is:

```
Point * complex_to_point (Complex *z)
{
```

```

Point *p;

p = (Point *) malloc(sizeof(Point));
p->x = z->x;
p->y = z->y;

return p;
}

```

Compatibility

SQL92

CREATE FUNCTION is a Postgres language extension.

SQL/PSM

Nota: PSM stands for Persistent Stored Modules. It is a procedural language and it was originally hoped that PSM would be ratified as an official standard by late 1996. As of mid-1998, this has not yet happened, but it is hoped that PSM will eventually become a standard.

SQL/PSM CREATE FUNCTION has the following syntax:

```

CREATE FUNCTION name
  ( [ [ IN | OUT | INOUT ] type [, ...] ] )
  RETURNS rtype
  LANGUAGE 'langname'
  ESPECIFIC routine
  SQL-statement

```

CREATE GROUP

Nombre

CREATE GROUP — Crea un grupo nuevo

Synopsis

```

CREATE GROUP name
  [ WITH
    [ SYSID gid ]
    [ USER username [, ...] ] ]

```


Entradas

name

El nombre del grupo.

gid

La clausula `SYSID` puede ser usada para elegir el numero id del grupo PostgreSQL del grupo nuevo. El uso de esta clausula es opcional.

En caso de no especificar el numero id del grupo, se asignara el numero mayor ya asignado mas uno, empezando por 1.

username

Una lista de los usuarios a incluir en el grupo. Los usuarios tienen que existir antes de incluirlos en el grupo.

Salidas

`CREATE GROUP`

Mensaje que sera devuelto siempre que la orden termina con exito.

Descripcion

`CREATE GROUP` permite crear un grupo nuevo en la base de datos. Consulte la guia del administrador para informaciones sobre el uso de grupos para prueba de autenticidad. Esta orden solamente podra ser ejecutada por un usuario administrativo.

Use *MODIFICAR GRUPO* para cambiar la pertenencia de un grupo y *DROP GROUP* para borrar un grupo.

Uso

Crear un grupo vacio:

```
CREATE GROUP staff
```

Crear un grupo con miembros:

```
CREATE GROUP marketing WITH USER jonathan, david
```

Compatibilidad

SQL92

En las especificaciones de SQL92 no existe la instrucción **CREATE GROUP**. El concepto de los Roles es similar al concepto de grupos.

CREATE INDEX

Nombre

CREATE INDEX — Construir un índice secundario.

Synopsis

```
CREATE [ UNIQUE ] INDEX nombre_indice ON tabla
    [ USING nombre_acceso ] ( columna [ nombre_operador] [, ...] )
CREATE [ UNIQUE ] INDEX nombre_indice ON tabla
    [ USING nombre_acceso ] ( nombre_funcion( r">columnale> [, ... ] ) nombre_operador )
```

Entradas

UNIQUE

Proboca que el sistema compruebe si existen valores duplicados en la tabla cuando se crea el índice (si ya existen datos) y cada vez que se añaden datos. Los intentos de insertar o actualizar datos duplicados generarán un error.

nombre_indice

El nombre del índice que se debe crear.

tabla

El nombre de la tabla para la que se quiere crear un índice.

nombre_acceso

El nombre del método de acceso que se utilizará para el índice. El método de acceso de defecto es BTREE. Postgres proporciona tres métodos de acceso para índices secundarios.

BTREE

una implementación de los btrees de alta concurrencia de Lehman-Yao.

RTREE

Implementa rtrees estándar utilizando el algoritmo de partición cuadrática de Guttman.

HASH

Una implementación de las dispersiones lineales de Litwin.

columna

El nombre de una columna de la tabla.

nombre_operador

Una clase de operadores asociada. Vea más abajo para obtener más detalles.

nombre_función

Una función definida por el usuario, que devuelve un valor que puede ser indexado.

Salidas**CREATE**

El mensaje devuelto si el índice se ha creado con éxito.

ERROR: Cannot create index: 'index_name' already exists.

Se presenta este error si es imposible crear el índice.

Description

CREATE INDEX construye un índice *nombre_indice* en la *tabla* especificada.

Sugerencia: Los índices se utilizan principalmente para incrementar el rendimiento de una base de datos. Sin embargo, un uso inapropiado de los índices dará lugar a una base de datos más lenta.

En la primera sintaxis mostrada antes, los campos claves para el índice se especifican como nombres de columna; una columna puede tener también una clase de operadores asociada. Una clase de operadores se utiliza para especificar los operadores que se utilizarán para un índice particular. Por ejemplo, un índice btree sobre enteros de cuatro_bytes debería utilizar la clase `int4_ops`; esta clase de operadores incluye funciones de comparación para enteros de cuatro_bytes. La clase de operadores de defecto es la apropiada para ese tipo de datos.

En la segunda sintaxis mostrada antes, se definía un índice como resultado de una función definida por el usuario *nombre_funcion* aplicada a uno o más atributos de una única clase. Estos *índices funcionales* pueden utilizarse para conseguir un acceso rápido a datos basados en operadores que normalmente requerirían algún tipo de transformación para aplicarlos a la base de datos.

Postgres proporciona métodos de acceso btree, rtree y hash para los índices secundarios. El método de acceso btree es una implementación de los btrees de alta concurrencia de Lehman-Yao. El método de acceso rtree implementa el algoritmo de partición cuadrática estándar de Guttman. El método de acceso hash es una implementación de las dispersiones lineales de Litwin. Mencionamos los algoritmos utilizados solamente para indicar que todos estos métodos de acceso son completamente dinámicos, y no necesitan ser optimizados periódicamente (como es el caso, por ejemplo, de los métodos de acceso de hash estático).

Notas

El optimizador de consultas de Postgres considerará que está usando índices btree en un barrido, siempre que un atributo indexado esté involucrado en una comparación que utilice uno de los siguientes: `<`, `<=`, `=`, `>=`, `>`

Ambas clases de caja (box) soportan índices en el tipo de datos `box` en Postgres. La diferencia entre ellas es que `bigbox_ops` escala coordenadas de caja hacia abajo, para impedir excepciones de punto flotante que se pudiesen producir en multiplicaciones, sumas y restas de coordenadas de punto flotante muy grandes. Si el campo al cual se unen sus rectángulos es de alrededor de 20.000 unidades cuadradas o mayor, debería usted utilizar `bigbox_ops`. La clase de operadores `poly_ops` soporta índices rtree en datos poligonales.

El optimizador de consultas de Postgres considerará que está utilizando un índice rtree siempre que un atributo indexado esté involucrado en una comparación que utilice uno de los siguientes: `<<`, `&<`, `&>`, `>>`, `@`, `~`, `=`, `&&`

El optimizador de consultas de Postgres considerará que está utilizando un índice hash siempre que un atributo del índice esté involucrado en una comparación que utilice el operador `=`.

Actualmente, sólo el método de acceso BTREE soporta índices multi-columna. Se pueden especificar hasta 7 claves.

Utilice `DROP INDEX` para eliminar un índice.

La clase de operadores `int24_ops` se puede utilizar para construir índices sobre datos `int2`, y realizar comparaciones contra datos `int4` en cualificaciones de consultas. Similarmente, `int42_ops` soporta índices sobre datos `int4` que deben ser comparados con datos `int2` en las consultas.

La siguiente lista select devuelve todos los nombres de operador:

```
SELECT am.amname AS acc_name,
       opc.opcname AS ops_name,
       opr.oprname AS ops_comp
FROM   pg_am am, pg_amop amop,
       pg_opclass opc, pg_operator opr
WHERE  amop.amopid = am.oid AND
       amop.amopclaid = opc.oid AND
       amop.amopopr = opr.oid
ORDER BY acc_name, ops_name, ops_comp
```

Usage

Para crear un índice en el campo `título` en la tabla `películas`:

```
CREATE UNIQUE INDEX índice_título
    ON películas (título);
```

Compatibilidad.

SQL92

CREATE INDEX es una extensión del lenguaje de Postgres.

No hay un comando **CREATE INDEX** en SQL92.

CREATE LANGUAGE

Nombre

CREATE LANGUAGE — Define un nuevo lenguaje para funciones

Synopsis

```
CREATE [ TRUSTED ] PROCEDURAL LANGUAGE 'langname'
    HANDLER call_handler
    LANCOMPILER 'comment'
```

Entradas

TRUSTED

TRUSTED especifica que el manipulador para el lenguaje es seguro; es decir, que no ofrece a un usuario no privilegiado nuevas funcionalidades sobrepasando las restricciones de acceso. Si esta palabra es omitida entonces al registrar el lenguaje, sólo usuarios con privilegio de superusuario Postgres podrán utilizar este lenguaje para crear nuevas funciones (como el lenguaje 'C').

langname

El nombre del nuevo lenguaje procedimental. No se diferencian mayúsculas de minúsculas en el nombre del lenguaje. Un lenguaje procedimental no puede redefinir uno de los lenguajes incorporados de Postgres. Postgres.

HANDLER *call_handler*

call_handler es el nombre de una función previamente registrada que será llamada para ejecutar los procedimientos PL.

comment

El argumento `LANCOMPILER` es la cadena que será insertada en el atributo `LANCOMPILER` de la nueva entrada `pg_language`. Actualmente Postgres no utiliza este atributo para ningún fin.

Salidas

`CREATE`

Este mensaje es devuelto si el lenguaje es creado con éxito.

`ERROR: PL handler function funcname() doesn't exist`

Este error es devuelto si la función *funcname*() no es encontrada.

Descripción

Utilizando **CREATE LANGUAGE**, un usuario Postgres puede registrar un nuevo lenguaje en Postgres. A continuación, las funciones y procedimientos "trigger" pueden ser definidos en este nuevo lenguaje. El usuario debe tener privilegios de superusuario Postgres para registrar un nuevo lenguaje.

Escritura de manipuladores PL

El manipulador de llamadas para un lenguaje procedimental debe ser escrito en un lenguaje compilado como 'C' y registrado en Postgres como una función sin argumentos y devolviendo el tipo `opaque`, un contenedor para tipos no definidos o especificados... Esto evita que el manipulador de llamadas sea llamado directamente como una función desde consultas.

Sin embargo, los argumentos deben ser suministrados en la llamada cuando una función PL o procedimiento trigger en el lenguaje ofrecido por el manipulador sea ejecutado.

- Cuando es llamado por el gestor de triggers, el único argumento es el ID del objeto tomada de la entrada de procedimientos `pg_proc`. Toda la demás información del gestor de triggers es encontrada en el puntero global `CurrentTriggerData`.
- Cuando es llamado desde el gestor de funciones, los argumentos son el ID del objeto de la entrada `pg_proc` del procedimiento, el número de argumentos entregados a la función PL, los argumentos en una estructura `FmgrValues` y un puntero a un booleano donde la función informa si el valor de retorno es el valor `NULL` de SQL.

Es responsabilidad del manipulador de llamadas obtener la entrada `pg_proc` y analizar el argumento y tipos de retorno del procedimiento llamado. La cláusula **AS** del **CREATE FUNCTION** del procedimiento estará basada en el atributo `prosrc` de la tabla `pg_proc`. Esto puede ser el texto fuente en el lenguaje procedimental mismo (co-

mo en PL/Tcl), una ruta a un fichero o cualquier otra cosa que le indique al handler que hacer en detalle.

Notas

Utilice **CREATE FUNCTION** para crear una función.

Utilice **DROP LANGUAGE** para eliminar lenguajes de procedimiento.

Remítase a la tabla `pg_language` para más información:

Table = pg_language			
Field		Type	Length
lanname	name		32
lancompiler	text		var
<pre> lanname lancompiler -----+----- internal n/a lisp /usr/ucb/liszt C /bin/cc sql postgres </pre>			

Ya que el manipulador (call handler) para un lenguaje de procedimientos debe ser registrado en Postgres en el lenguaje 'C', hereda todas las capacidades y restricciones de las funciones de 'C'.

Actualmente, las definiciones para un lenguaje de procedimientos no pueden ser modificadas una vez que han sido creadas.

Uso

Esta es una plantilla para un manipulador en 'C':

```

#include "executor/spi.h"
#include "commands/trigger.h"
#include "utils/elog.h"
#include "fmgr.h" /* for FmgrValues struct */
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"

Datum
plsample_call_handler(
    Oid      prooid,
    int      pronargs,
    FmgrValues *proargs,
    bool     *isNull)
{
    Datum      retval;
    TriggerData *trigdata;

```

```

    if (CurrentTriggerData == NULL) {
        /*
         * Llamado como una función
         */

        retval = ...
    } else {
        /*
         * Llamado como un procedimiento "trigger"
         */
        trigdata = CurrentTriggerData;
        CurrentTriggerData = NULL;

        retval = ...
    }

    *isNull = false;
    return retval;
}

```

Solamente unos pocos miles de líneas de código tienen que ser añadidas en vez de los puntos para completar el 'PL call handler' Vea **CREATE FUNCTION** para información sobre como compilarlo en un módulo cargable.

Los siguientes comandos entonces registran el lenguaje de procedimientos de muestra:

```

CREATE FUNCTION plsample_call_handler () RETURNS opaque
    AS '/usr/local/pgsql/lib/plsample.so'
    LANGUAGE 'C';
CREATE PROCEDURAL LANGUAGE 'plsample'
    HANDLER plsample_call_handler
    LANCMPILER 'PL/Sample';

```

Compatibilidad

SQL92

CREATE LANGUAGE es una extensión de Postgres. No existe una sentencia **CREATE LANGUAGE** en SQL92.

CREATE OPERATOR

Nombre

CREATE OPERATOR — Define un nuevo operador de usuario

Synopsis

```
CREATE OPERATOR name ( PROCEDURE = func_name
    [, LEFTARG = type1 ] [, RIGHTARG = type2 ]
    [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
    [, RESTRICT = res_proc ] [, JOIN = join_proc ]
    [, HASHES ] [, SORT1 = left_sort_op ] [, SORT2 = right_sort_op ] )
```

Entradas

name

El operador a definir. Véanse más abajo los caracteres permitidos.

func_name

La función utilizada para implementar este operador.

type1

El tipo de la parte izquierda del operador, si procede. Esta opción debería ser omitida para un operador unario por la derecha.

type2

El tipo para la parte derecha del operador, si procede. Esta opción debería ser omitida para un operador unario por la izquierda.

com_op

El conmutador para este operador.

neg_op

El negador para este operador.

res_proc

La función estimadora de restricción selectiva para este operador.

join_proc

*****The join selectivity estimator function for this operator. *****La función estimador de ????

HASHES

Indica que este operador soporta un algoritmo "hash-join".

left_sort_op

Operador que ordena el tipo de dato de la parte izquierda de este operador.

right_sort_op

Operador que ordena el tipo de dato de la parte derecha de este operador.

Salidas

CREATE

Mensaje devuelto si el operador es creado con éxito.

Description

CREATE OPERATOR define un nuevo operador, *name*. El usuario que define el operador se convierte en su propietario.

El operador *name* es una secuencia de hasta treinta y dos (32) caracteres con cualquiera combinación de lo siguiente:

+ - * / < > = ~ ! @ # % ^ & | ' ? \$:

Nota: No se permite ningún carácter alfabético en un nombre de operador. Esto permite a Postgres analizar la entrada SQL en elementos sin requerir espacio entre cada elemento.

El operador "!=" es convertido a "<>" en la entrada, por lo que son en consecuencia equivalentes.

Por lo menos uno de LEFTARG o RIGHTARG deben ser definidos. Para operadores binarios, ambos deberían ser definidos. Para operadores unarios por la derecha, solamente LEFTARG debería ser definido, mientras que en operadores unarios por la izquierda solamente RIGHTARG debería ser definido.

También, el procedimiento *func_name* debe haber sido previamente definido utilizando **CREATE FUNCTION** y debe ser definido para aceptar el número correcto de argumentos (bien uno o dos).

El operador conmutador debería ser identificado si existe uno, para que Postgres pudiese invertir el orden de los operandos si lo desea. Por ejemplo, el operador *area-menor-que*, <<<, debería probablemente tener un operador conmutador *area-mayor-que* >>>. De esta forma, el optimizador de consultas podría convertir libremente:

```
"0,0,1,1"::box >>> MYBOXES.description
```

a

```
MYBOXES.description <<< "0,0,1,1"::box
```

Esto permite la ejecución de código para utilizar siempre la última representación y simplifica algo el optimizador.

De forma similar, si existe un operador negador entonces debería ser identificado. Supongamos que un operador, area-igual, `===`, existe, y también un operador area-no-igual, `!==`. El negador permite al optimizador simplificar

```
NOT MYBOXES.description === "0,0,1,1"::box
```

a

```
MYBOXES.description !== "0,0,1,1"::box
```

Si el nombre de un operador conmutador es suministrado, Postgres lo busca en el catálogo. Si es encontrado e no tiene aún un conmutador él mismo, entonces la entrada del conmutador es actualizada para tener el recién creado operador como su conmutador. Esto se aplica al negador, también.

Esto es para permitir la definición de dos operadores que son conmutadores de los negadores de cada uno de los otros. El primer operador debería ser definido sin un conmutador o negador (como sea apropiado). Cuando el segundo operador es definido, se debe nombrar el primero como el conmutador o negador. El primero será actualizado como un efecto lateral. (En Postgres 6.5, esto también funciona para simplemente que ambos operadores se refieran al otro).

Los siguientes tres especificadores están presentes para auxiliar al optimizador de consultas al realizar uniones ("joins"). Postgres siempre puede evaluar una unión (i.e., procesando una cláusula con dos variables de tuplas separadas por un operador que retorne un booleano) por substitución iterativa [WONG76]. Además, Postgres es capaz de utilizar un algoritmo "hash-join" siguiendo las líneas de [SHAP86]; sin embargo, debe saber si esta estrategia es aplicable. Es algoritmo "hash-join" actual es solamente correcto para operadores que representan tests de igualdad; además la igualdad del tipo de dato debe significar igualdad a nivel de bits de la representación del tipo. (Por ejemplo, un tipo de dato que contiene bits no utilizados que no tienen repercusión para tests de igualdad podría no ser usado en el "hash-join"). El indicador HASHES indica al optimizador de consultas que un hash join puede ser utilizado de forma segura por este operador.

De forma parecida, los dos operadores de orden indican al optimizador de consultas si la estrategia mezclar-ordenar es utilizable y que operadores deberían ser utilizados para ordenar las clases de los dos operadores. Los operadores de orden deberían ser suministrados solamente para un operador de igualdad, y deberían referirse a operadores menor-que para los tipos de la parte izquierda y derecha respectivamente.

Si otras estrategias de unión son consideradas prácticas, Postgres cambiará el optimizador en tiempo de ejecución para utilizarlas y requerirán especificación adicional cuando un operador sea definido. Afortunadamente, la comunidad investigadora inventa nuevas estrategias de unión infrecuentemente, y la generalidad añadida de estrategias definidas por el usuario no merece la complejidad resultante.

Las dos últimas piezas de la especificación están presentes para que el optimizador pueda estimar los tamaños de los resultados. Si una cláusula de la forma:

```
MYBOXES.description <<< "0,0,1,1"::box
```

está presente en la cualificación, entonces Postgres puede tener que estimar la fracción de instancias en MYBOXES que satisfacen la cláusula. La función *res_proc* debe ser una función registrada (lo que significa que ya está definida utilizando **CREATE FUNCTION**), acepta argumentos del tipo correcto y devuelve un numero en punto

flotante. El optimizador simplemente llama a esta función, pasándole el parámetro "0,0,1,1" y multiplica el resultado por el tamaño de la relación para obtener el deseado número de instancias estimado.

Cuando ambos operandos del operador contienen variables de instancia, el optimizador debe estimar el tamaño de la unión resultante. La función `join_proc` retornara otro número decimal que será multiplicado por las cardinalidades de las dos clases envueltas en el cómputo del tamaño esperado.

La diferencia entre la función

```
my_procedure_1 (MYBOXES.description, "0,0,1,1"::box)
```

y el operador

```
MYBOXES.description === "0,0,1,1"::box
```

es que Postgres intenta optimizar operadores y puede decidir utilizar un índice para restringir el espacio de búsqueda cuando aparecen operadores. Sin embargo, no se intenta optimizar funciones, y son ejecutadas mediante fuerza bruta. Además, las funciones pueden tener cualquier número de argumentos mientras que los operadores están restringidos a uno o dos.

Notes

Refiérase al capítulo sobre operadores en la *PostgreSQL User's Guide* para más información. Refiérase a **DROP OPERATOR** para borrar operadores definidos por el usuario de una base de datos.

Utilización Usage

El siguiente comando define un nuevo operador, `area-igualdad`, para el tipo de dato `BOX`.

```
CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    PROCEDURE = area_equal_procedure,
    COMMUTATOR = ===,
    NEGATOR = !==,
    RESTRICT = area_restriction_procedure,
    JOIN = area_join_procedure,
    HASHES,
    SORT1 = <<,
    SORT2 = <<
);
```

Compatibility

SQL92

CREATE OPERATOR is a Postgres extension. There is no **CREATE OPERATOR** statement in SQL92.

CREATE RULE

Nombre

CREATE RULE — Define una nueva regla

Synopsis

```
CREATE RULE name AS ON event
  TO object [ WHERE condition ]
  DO [ INSTEAD ] [ action | NOTHING ]
```

Inputs

name

El nombre de la regla a crear.

event

Evento puede ser `select`, `update`, `delete` o `insert`.

object

Object puede ser *table* o *table.column*.

condition

Cualquiera clausula SQL WHERE. `new` o `current` pueden aparecer en lugar de una variable de instancia*** siempre que una variable de instancia es admisible en SQL.

action

Cualquiera clausula SQL. `new` o `current` pueden aparecer en lugar de una variable de instancia*** siempre que una variable de instancia sea admisible en SQL.

Salidas

CREATE

Mensaje devuelto si la regla es creada con éxito.

Description

El Postgres *rule system* permite que una action alternativa sea realizada en updates, inserts o deletes en tablas o clases. Actualmente se utilizan reglas para implementar vistas de tablas.

El significado de una regla es que cuando una instancia individual es accedida, actualizada, insertada o borrada, existe una instancia actual (para consultas, actualizaciones y borrados) y una nueva instancia (para actualizaciones y añadidos). Si el *event* especificado en la clausula ON y la *condition* especificada en la clausula WHERE son verdaderas para la instancia actual la parte *action* de la regla es ejecutada. Antes, sin embargo, los valores de los campos de la instancia actual y/o la nueva instancia son sustituidos por *current.attribute-name* y *new.attribute-name*.

La parte *action* de la regla se ejecuta con el mismo identificador de comando y transacción que el comando de usuario que causó la activación.

Notas

Es pertinente la precaución con reglas de SQL. Si el mismo nombre de clase o variable de instancia aparece en el *event*, la *condition* y la parte *action* de la regla, son considerados todos diferentes tuplas. De forma más precisa, *new* y *current* son las únicas tuplas que son compartidas entre cláusulas. Por ejemplo, las siguientes dos reglas tienen la misma semántica.

```
ON UPDATE TO emp.salary WHERE emp.name = "Joe"
DO UPDATE emp ( ... ) WHERE ...
```

```
ON UPDATE TO emp-1.salary WHERE emp-2.name = "Joe"
DO UPDATE emp-3 ( ... ) WHERE ...
```

Cada regla puede tener el tag opcional INSTEAD. Sin este tag, la *action* sera realizada en adición al comando de usuario cuando el *event* en la parte *condition* de la regla aparezcan. Alternativamente, la parte *action* será realizada en lugar del comando del usuario. En este último caso la *instead of the user command*. In this later case, the *action* puede ser la palabra clave NOTHING.

Cuando se elige entre los sistemas de reescritura y reglas de instancia para una aplicación particular de una regla, recuerdese que en el sistema de reescritura, *current* se refiere a la relación y algunos cualificadores mientras que en el sistema de instancias se refiere a una instancia (tupla).

Es muy importante notar que el sistema de reescritura nunca detectará ni procesará reglas circulares. Por ejemplo, aunque cada una de las siguientes dos reglas con aceptadas por Postgres, el comando de recogida causará la caída de Postgres :

Ejemplo 14-1. Ejemplo de combinación circular de reglas.

```
CREATE RULE bad_rule_combination_1 AS
    ON SELECT TO emp
    DO INSTEAD SELECT TO toyemp;

CREATE RULE bad_rule_combination_2 AS
    ON SELECT TO toyemp
    DO INSTEAD SELECT TO emp;
```

Este intento de obtención de datos desde EMP provocará la caída de Postgres.

```
SELECT * FROM emp;
```

Es necesario tener permiso de definición de reglas en una clase para poder definir una regla en el. Se debe utilizar el comando **GRANT** y **REVOKE** para modificar estos permisos.

El objeto en una regla SQL no puede ser una referencia a un array y no puede tener parámetros.

Aparte del campo "oid", los atributos del sistema no pueden ser referenciados en ningún lugar en una regla. Entre otras cosas esto significa que las funciones de instancias (por ejemplo `foo(emp)` donde `emp` es una clase) no pueden ser llamadas en ningún lugar dentro de una regla.

El sistema almacena el texto de la regla y los planes de consulta como atributos de texto. Esto implica que la creación de reglas puede fallar si la regla más sus varias internas representaciones exceden algún valor que es del orden de una página.

Uso

Hacer que Sam obtenga el mismo ajuste de salario que Joe:

```
CREATE RULE example_1 AS
    ON UPDATE emp.salary WHERE current.name = "Joe"
    DO UPDATE emp (salary = new.salary)
    WHERE emp.name = "Sam";
```

Al mismo tiempo que Joe recibe un ajuste de salario, el evento será verdadero y la instancia actual de Joe y la nueva instancia propuesta están disponibles para las rutinas de ejecución. Por lo tanto, este nuevo salario es sustituido en la parte de acción de la regla que es subsiguientemente ejecutada. Esto propaga el salario de Joe a Sam.

Hacer que Bill obtenga el salario de Joe cuando es accedido:

```
CREATE RULE example_2 AS
    ON SELECT TO EMP.salary
    WHERE current.name = "Bill"
    DO INSTEAD
    SELECT (emp.salary) from emp
    WHERE emp.name = "Joe";
```

Denegar a Joe el acceso al salario de empleados en el departamento de calzado (`current_user` devuelve el nombre del usuario actual):

```
CREATE RULE example_3 AS
  ON SELECT TO emp.salary
  WHERE current.dept = "shoe" AND current_user = "Joe"
  DO INSTEAD NOTHING;
```

Crear una vista de empleados trabajando en el departamento de juguetes.

```
CREATE toyemp(name = char16, salary = int4);

CREATE RULE example_4 AS
  ON SELECT TO toyemp
  DO INSTEAD
  SELECT (emp.name, emp.salary) FROM emp
  WHERE emp.dept = "toy";
```

Todos los nuevos empleados deben hacer 5.000 o menos.

```
CREATE RULE example_5 AS
  ON INSERT TO emp WHERE new.salary > 5000
  DO UPDATE NEWSET salary = 5000;
```

Compatibility

SQL92

El comando **CREATE RULE** es una extensión de Postgres No existe la sentencia **CREATE RULE** en SQL92.

CREATE SEQUENCE

Nombre

CREATE SEQUENCE — Crea una nueva secuencia de generador de numeros

Synopsis

```
CREATE SEQUENCE seqname [ INCREMENT increment ]
  [ MINVALUE minvalue ] [ MAXVALUE maxvalue ]
  [ START start ] [ CACHE cache ] [ CYCLE ]
```


Entradas

seqname

El nombre de una secuencia que sera creada.

increment

La clausula INCREMENT *increment* es opcional. Un valor positivo hara una secuencia ascendente, uno negativo hara una secuencia descendente. El valor por omision es uno (1).

minvalue

La clausula opcional MINVALUE *minvalue* determina el valor minimo que una secuencia puede generar. El valor por omision es 1 y -2147483647 para secuencias ascendentes y descendentes, respectivamente.

maxvalue

Utilice la clausula opcional MAXVALUE *maxvalue* para determinar el valor maximo para una secuencia. Por omision son 2147483647 y -1 para secuencias ascendentes y descendentes, respectivamente.

start

La clausula opcional START *start* habilita la secuencia para que comience en cualquier lugar. El valor de inicio por omision es *minvalue* para secuencias ascendentes y *maxvalue* para las descendentes.

cache

La opcion CACHE *cache* permite que los numeros de la secuencia sean alojados (preallocated) y almacenados en memoria para un acceso mas rapido. El valor minimo es 1 (solo se puede generar un valor cada vez, i.e. sin cache) y es tambien el valor por omision.

CYCLE

La palabra clave (keyword) CYCLE puede ser utilizada para permitir a la secuencia continuar cuando el valor de *maxvalue* o el de *minvalue* ha sido alcanzado por una secuencia ascendente o descendente respectivamente. Si el limite es alcanzado, el siguiente numero generado sera cualquiera que para *minvalue* o *maxvalue* sea tomado como apropiado.

Outputs

CREATE

Mensaje devuelto si el comando es ejecutado con exito.

ERROR: Relation '*seqname*' already exists

Si la secuencia especificada ya existe.

ERROR: DefineSequence: MINVALUE (*start*) can't be >= MAXVALUE (*max*)

Si el valor de inicio especificado esta fuera de rango.

```
ERROR: DefineSequence: START value (start) can't be < MINVALUE (min)
```

Si el valor de inicio especificado esta fuera de rango.

```
ERROR: DefineSequence: MINVALUE (min) can't be >= MAXVALUE (max)
```

Si el valor minimo y maximo son inconsistentes.

Descripcion

CREATE SEQUENCE introducira una nueva secuencia generadora de numeros dentro de la actual base de datos. Esto implica la creacion e inicializacion de una nueva tabla de una linea con el nombre *seqname*. La secuencia generadora sera propiedad del usuario que ejecuta el comando.

Depsues de que se crea una secuencia, puede utilizar la funcion `nextval(seqname)` par aobtener una nuevo numero de la secuencia. La funcion `currval('seqname')` puede ser utilizada para determinar el numerp devuelto por la ultima llamada a `nextval(seqname)` desde la secuencia especificada en la sesion en curso. La funcion `setval('seqname', newvalue)` puede ser utilizada para configurar el valor actual de la secuencia especificada. La siguiente llamada a `nextval(seqname)` devolvera el valor dado mas la secuencia de incremento.

Utilice una consulta (query) como

```
SELECT * FROM sequence_name;
```

para obtener los parametros de una secuencia. Aparte de obtener los parametros originales, puede utilizar

```
SELECT last_value FROM sequence_name;
```

para obtener el ultimo valor asignado por cualquier proceso en el servidor (backend). to obtain the last value allocated by any backend. parametros que puedes utilizar

Se utliza bajo nivel de bloque para habilitar multiples llamadas simultaneas a un generador.

Atención

Se pueden obtener resultados inesperados si una configuración de cache mayor que uno es utilizada por un objeto secuencia que será usado concurrentemente por múltiples procesos en el servidor (backends). Cada proceso en el servidor (backend) asignará valores de secuencia "cache" sucesivas durante un acceso al objeto secuencia e incrementará el último valor (`last_value`) del objeto secuencia en conformidad con esto. De este modo, el siguiente uso de `cache=1` de `nextval` dentro de ese proceso en el servidor (backend) devolverá simplemente los valores asignados sin tocar el objeto compartido. Así pues, los números asignados pero no utilizados en la sesión en curso se perderán. Mas aun, aunque se garantice por múltiples procesos en el servidor (backends) la asignación de distintos valores de secuencia, los valores pueden ser generados fuera de secuencia cuando los procesos en el servidor (backends) son tenidos en cuenta. (Por ejemplo, con una configuración de cache de 10, el proceso A puede reservar valores 1..10 y devolver un `nextval=1`, entonces el proceso B puede reservar valores 11..20 y devolver un `nextval=11` antes de que el proceso A ha generado un `nextval=2`.) Así, con una configuración de cache de uno es seguro asumir que los valores de `nextval` serán generados secuencialmente; con una cache configurada mayor que uno solo podrías asumir que los valores de `nextvalue` serán todos distintos, no que serán todos generados de un modo puramente secuencial. También, `last_value` reflejará el último valor reservado por cualquier proceso en el servidor (backend), tanto si ya ha sido devuelto por `nextval` como si no.

Notas

Remítase a estado **DROP SEQUENCE** para eliminar una secuencia.

Cada proceso en el servidor (backend) utiliza su propia cache para almacenar números asignados. Los números que están en la cache pero no son utilizados en la sesión en curso se perderán, dando como resultado "vacíos" en la secuencia.

Uso

Crea una secuencia ascendente llamada `serial`, comenzando en 101:

```
CREATE SEQUENCE serial START 101;
```

Seleccione el siguiente número de esta secuencia

```
SELECT NEXTVAL ('serial');
```

```
nextval
-----
      114
```

Utilice esta secuencia en una **INSERT**:

```
INSERT INTO distributors VALUES (NEXTVAL('serial'),'nothing');
```

Configurar el valor de la secuencia despues de un COPY FROM:

```
CREATE FUNCTION distributors_id_max() RETURNS INT4
AS 'SELECT max(id) FROM distributors'
LANGUAGE 'sql';
BEGIN;
COPY distributors FROM 'input_file';
SELECT setval('serial', distributors_id_max());
END;
```

Compatibilidad

SQL92

CREATE SEQUENCE es una extension de lenguaje Postgres. No hay estado **CREATE SEQUENCE** en SQL92.

CREATE TABLE

Nombre

CREATE TABLE — Crea una nueva tabla

Synopsis

```
CREATE [ TEMPORARY | TEMP ] TABLE table (
    column type
    [ NULL | NOT NULL ] [ UNIQUE ] [ DEFAULT value ]
    [column_constraint_clause | PRIMARY KEY } [ ... ] ]
    [, ... ]
    [, PRIMARY KEY ( column [, ...] ) ]
    [, CHECK ( condition ) ]
    [, table_constraint_clause ]
    ) [ INHERITS ( inherited_table [, ...] ) ]
```

Entradas

TEMPORARY

Se crea la tabla sólo para esta sesión, y es eliminada automáticamente con el fin de la sesión. Las tablas permanentes existentes con el mismo nombre no son visibles mientras la table temporal existe.

table

El nombre de una nueva clase o tabla a crear.

column

El nombre de un campo.

type

El tipo del campo. Puede incluir especificadores de array. Consulte la *PostgreSQL User's Guide* para más información sobre tipos y arrays.

DEFAULT *value*

Un valor por defecto para el campo. Consulte la cláusula DEFAULT para más información.

column_constraint_clause

La cláusula opcional de restricciones (constraint) especifica una lista de restricciones de integridad o comprueba que las nuevas inserciones o actualizaciones deben satisfacer para que la inserción o la actualización tenga éxito. Cada restricción debe evaluarse a una expresión booleana. Aunque SQL92 requiere la *column_constraint_clause* para referirse a ese campo solamente, Postgres permite que múltiples campos sean referenciados dentro de un único campo constraint. Consulte la cláusula constraint para más información.

table_constraint_clause

La cláusula opcional CONSTRAINT especifica una lista de restricciones de integridad que las nuevas inserciones o las actualizaciones deberán satisfacer para que una sentencia insert o update tenga éxito. Cada restricción debe ser evaluada a una expresión booleana. Se pueden referenciar múltiples campos con una única restricción. Sólo se puede definir una única cláusula PRIMARY KEY por tabla; PRIMARY KEY *column* (una restricción de tabla) and PRIMARY KEY (una restricción de campo) son mutuamente excluyentes. Consulte la cláusula de restricción de tabla para más información.

INHERITS *inherited_table*

La cláusula opcional INHERITS especifica una colección de nombres de tabla de las cuales esta tabla hereda todos los campos. Si algún campo heredado aparece más de una vez, Postgres informa de un error. Postgres permite automáticamente a la tabla creada heredar funciones de las tablas superiores a ella en la jerarquía de herencia.

Aside: La herencia de funciones se realiza siguiendo las convenciones del Common Lisp Object System (CLOS).

Salidas

CREATE

Mensaje devuelto si la table se ha creado con éxito.

ERROR

Mensaje devuelto si la creación de la tabla falla. Este mensaje viene normalmente acompañado por algún texto explicativo, como: `ERROR: Relation 'table' already exists` que ocurre en tiempo de ejecución, si la tabla especificada ya existe en la base de datos.

ERROR: DEFAULT: type mismatched

Si el tipo de datos o el valor por defecto no corresponde al tipo de datos de la definición del campo.

Description

CREATE TABLE introducirá una nueva clase o tabla en la base de datos actual. La tabla será poseída por el usuario que introduce la sentencia.

Cada *type* puede ser un tipo simple, un tipo complejo (set) o un tipo array. Cada atributo puede ser especificado para ser no nulo, y puede tener un valor por defecto, especificado por la *Cláusula DEFAULT*.

Nota: Al igual que en la versión 6.0 de Postgres, constant array dimensions within an attribute are not enforced. Esto cambiará probablemente en las versiones futuras.

La cláusula opcional **INHERITS** especifica una colección de nombres de clases de los cuales esta clase hereda automáticamente todos los campos. Si cualquier nombre de campo heredado aparece más de una vez, Postgres informa de un error. Postgres permite automáticamente a la clase creada heredar funciones de clases superiores en la jerarquía de herencia. La herencia de funciones se hace siguiendo las convenciones del Common Lisp Object System (CLOS).

Cada nueva tabla o clase *table* es creada automáticamente como tipo. Por tanto, una o más instancias de la clase son automáticamente un tipo y pueden ser usadas en otras *MODIFICAR TABLA* sentencias **CREATE TABLE**.

The new table is created as a heap with no initial data. Una tabla no puede tener más de 1600 campos (realmente, esto viene limitado por el hecho que el máximo tamaño de una tupla debe ser menor que 8192 bytes), pero este límite puede ser configurado a un tamaño menor en algunos sitios. Una tabla no puede tener el mismo nombre que una tabla de catálogo del sistema.

Cláusula DEFAULT

`DEFAULT value`

Entradas

value

Los posibles valores para la expresión DEFAULT (valor por defecto) son:

- un literal
- una función de usuario
- una función niladic

Salidas

Ninguna.

Descripción

La cláusula DEFAULT asigna un valor por defecto a un campo (a través de una definición de campo en la sentencia CREATE TABLE). El tipo de dato de un valor por defecto debe corresponder al tipo de dato de la definición del campo.

Una operación de inserción (INSERT) que incluya un campo sin un valor especificado por defecto asignará el valor NULL al campo si no se le proporciona un valor explícito. Si el valor por defecto es un *literal* significa que es un valor constante. Si el valor por defecto es una *niladic-function* o una *user-function* significa que dicho valor es el de la función especificada en el momento de la inserción.

Hay dos tipos de funciones niladic:

niladic USER

CURRENT_USER / USER

Consulte las funciones CURRENT_USER

SESSION_USER

todavía no soportadas

SYSTEM_USER

todavía no soportadas

niladic datetime

CURRENT_DATE

Consulte las funciones CURRENT_DATE

CURRENT_TIME

Consulte las funciones CURRENT_TIME

CURRENT_TIMESTAMP

Consulte la función CURRENT_TIMESTAMP

En la versión actual (v6.5), Postgres evalúa todas las expresiones por defecto en el momento en que la tabla es definida. Por tanto, las funciones que son "non-cacheable" como CURRENT_TIMESTAMP pueden no producir el efecto deseado. Para el caso particular de tipos date/time, se puede trabajar sobre este comportamiento usando "DEFAULT TEXT 'now'" en lugar de "DEFAULT 'now'" o "DEFAULT CURRENT_TIMESTAMP". Esto fuerza Postgres a considerar la constante como un tipo string y así convertirla al valor timestamp en tiempo de ejecución.

Uso

Para asignar un valor constante como valor por defecto para los campos did and number, y una cadena al campo did:

```
CREATE TABLE video_sales (
    did      VARCHAR(40) DEFAULT 'luso films',
    number   INTEGER DEFAULT 0,
    total    CASH DEFAULT '$0.0'
);
```

Para asignar una secuencia existente como valor por defecto para el campo did, y un literal para el campo name:

```
CREATE TABLE distributors (
    did      DECIMAL(3) DEFAULT NEXTVAL('serial'),
    name     VARCHAR(40) DEFAULT 'luso films'
);
```

Column CONSTRAINT Clause

```
[ CONSTRAINT name ] { [
    NULL | NOT NULL ] | UNIQUE | PRIMARY KEY | CHECK constraint } [, ...]
```

Entradas

name

Un nombre arbitrario dado a la restricción de integridad. Si no se especifica *name*, se genera de los nombres de la tabla y campos, lo que asegura unicidad para *name*.

NULL

El campo puede contener valores NULL. Ésta es la opción por defecto.

NOT NULL

El campo no puede contener valores NULL. Esto equivale a la restricción de campo CHECK (*column* NOT NULL).

UNIQUE

El campo debe contener un valor único. En Postgres esto es forzado por medio de la creación implícita de un índice único sobre la tabla.

PRIMARY KEY

Este campo es una clave primaria, lo que implica que la unicidad es forzada por el sistema y que otras tablas pueden confiar en este campo como identificador único para los registros. Consulte PRIMARY KEY para más información.

constraint

La definición de la restricción.

Descripción

La cláusula opcional de restricción (CONSTRAINT) especifica restricciones o verifica qué deben cumplir las nuevas inserciones o las actualizaciones para que una operación de inserción o de actualización tenga éxito. Cada restricción debe evaluarse a una expresión booleana. Con una única restricción se pueden referenciar múltiples atributos. El uso de PRIMARY KEY como restricción de tabla es mutuamente incompatible con el uso de PRIMARY KEY como restricción de campo.

Una restricción es una regla con nombre: un objeto SQL que ayuda a definir conjuntos de valores válidos poniendo límites a los resultados de las operaciones INSERT, UPDATE or DELETE sobre una tabla.

Existen dos maneras de definir restricciones de integridad: restricciones de tabla, que veremos más adelante, y restricciones de campo, que pasamos a ver.

Una restricción de campo es una restricción de integridad definida como parte de la definición de campo, y lógicamente se convierte en una restricción de tabla nada más ser creada. Las restricciones de campo disponibles son:

PRIMARY KEY
REFERENCES
UNIQUE
CHECK
NOT NULL

Nota: Postgres todavía no soporta (en su versión 6.5) restricciones de integridad especificadas por REFERENCES. Se acepta la sintaxis pero se ignora la cláusula (disponible, en cambio, a partir de la versión 7.0)

Restricción NOT NULL

```
[ CONSTRAINT name ] NOT NULL
```

La restricción NOT NULL especifica una regla que obliga que un campo contenga únicamente valores no nulos. Ésta es únicamente una restricción de campo, y no se permite como restricción de tabla..

Salidas

status

```
ERROR: ExecAppend: Fail to add null value in not null attribute "column".
```

Este error ocurre en tiempo de ejecución cuando se intenta insertar un valor nulo en un campo que contiene la restricción NOT NULL.

Descripción

Uso

Definir dos restricciones de campo NOT NULL en la tabla `distributors`, una de las cuales se nombra:

```
CREATE TABLE distributors (
    did      DECIMAL(3) CONSTRAINT no_null NOT NULL,
    name     VARCHAR(40) NOT NULL
);
```

Restricción UNIQUE

```
[ CONSTRAINT name ] UNIQUE
```

Entradas

CONSTRAINT *name*

Una etiqueta arbitraria dada a una restricción.

Salidas

status

```
ERROR: Cannot insert a duplicate key into a unique index.
```

Este error ocurre en tiempo de ejecución si se intenta insertar un valor duplicado en un campo.

Descripción

La restricción UNIQUE especifica una regla que obliga a un grupo de uno o más campos de una tabla a contener valores únicos.

Las definiciones de campo de las columnas especificadas no tienen porqué incluir una restricción NOT NULL constraint para ser incluidos en una restricción UNIQUE. Tener más de un valor nulo en un campo sin la restricción NOT NULL, no viola la restricción UNIQUE. (This deviates from the SQL92 definition, but is a more sensible convention. See the section on compatibility for more details.).

Cada restricción de campo UNIQUE debe nombrar un campo que es distinto del conjunto de campos nombrados por cualquier otra restricción UNIQUE o PRIMARY KEY definidas por la tabla.

Nota: Postgres crea automáticamente un índice único por cada restricción UNIQUE, para asegurar la integridad de los datos. Vea CREATE INDEX para más información.

Uso

Define una restricción de campo UNIQUE para la tabla distributors. Las restricciones de campo UNIQUE solo son definidas sobre un campo de la tabla:

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40) UNIQUE
);
```

lo que equivale a la siguiente restricción de tabla:

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40),
    UNIQUE(name)
);
```

La restricción CHECK

```
[ CONSTRAINT name ] CHECK
    ( condition [, ...] )
```

Entradas

name

Un nombre arbitrario dado a la restricción.

condition

Cualquier expresión condicional válida que se evalúe a un resultado booleano.

Outputs

status

```
ERROR: ExecAppend: rejected due to CHECK constraint "table_column".
```

Este error ocurre en tiempo de ejecución si alguien intenta insertar un valor ilegal en un campo sujeto a una restricción CHECK.

Descripción

La restricción CHECK especifica una restricción sobre los valores permitidos en un campo. La restricción CHECK también se permite como restricción de tabla.

Las restricciones de campo CHECK de SQL92 sólo pueden ser definidas sobre un campo de la tabla, y solamente pueden referirse a un campo. Postgres no tiene esta restricción.

Restricción PRIMARY KEY

```
[ CONSTRAINT name ] PRIMARY KEY
```

Entradas

CONSTRAINT *name*

Un nombre arbitrario para la restricción.

Salidas

ERROR: No se puede insertar un valor duplicado en un índice único.

Esto ocurre en tiempo de ejecución si alguien intenta insertar un valor duplicado en una columna sujeta a una restricción PRIMARY KEY.

Descripción

La restricción de campo PRIMARY KEY especifica que un campo de una tabla solamente puede contener valores únicos (no duplicados) y no nulos . La definición de la columna especificada no tiene que incluir una restricción explícita NOT NULL para ser incluida en una restricción PRIMARY KEY.

Sólo se puede especificar una única clave primaria (PRIMARY KEY) por tabla.

Notas

Postgres crea automáticamente un índice único para asegurar la integridad de los datos. (Vea la sentencia CREATE INDEX)

La restricción de clave primaria (PRIMARY KEY) debe nombrar un conjunto de campos que no sean contenidos por ninguna otra restricción UNIQUE definidos por la misma tabla, ya que produciría una duplicación de índices equivalentes y una sobrecarga adicional en tiempo de ejecución. Sin embargo, Postgres no lo deshabilita específicamente.

Cláusula CONSTRAINT para tablas

```
[ CONSTRAINT name ] { PRIMARY KEY | UNIQUE } ( column [, ...] )
[ CONSTRAINT name ] CHECK ( constraint )
```

Entradas

CONSTRAINT *name*

Un nombre arbitrario dado a una restricción de integridad.

column [, ...]

El nombre de los campos para los que definimos un índice único y, para la PRIMARY KEY, una restricción NOT NULL.

CHECK (*constraint*)

Una expresión booleana a ser evaluado como la restricción.

Salidas

Las posibles salidas para la cláusula de restricción de tablas son las mismas que para las partes correspondientes de la cláusula restricción de campo.

Descripción

Una restricción de tabla es una restricción de integridad definida sobre uno o más campos de una tabla base. Las cuatro variaciones de restricciones de tabla son:

UNIQUE
CHECK
PRIMARY KEY
FOREIGN KEY

Nota: Postgres todavía no soporta (en su versión 6.5) las restricciones de integridad FOREIGN KEY. El compilador entiende la sintaxis de FOREIGN KEY, pero solo imprime un aviso e ignora la cláusula. Las claves ajenas pueden ser parcialmente emuladas por medio de triggers (Consulte la sentencia CREATE TRIGGER).

Restricción UNIQUE

```
[ CONSTRAINT name ] UNIQUE ( column [, ...] )
```

Entradas

CONSTRAINT *name*

Un nombre arbitrario dado a una restricción.

column

Un nombre de un campo en una tabla.

Salidas

status

ERROR: Cannot insert a duplicate key into a unique index.

Este error ocurre en tiempo de ejecución si se intenta insertar un valor duplicado en un campo.

Descripción

La restricción UNIQUE especifica una regla en la que un grupo de uno o más campos de una tabla puede contener solo valores únicos. El comportamiento de la restricción de tabla UNIQUE es el mismo que para la restricción de campo, con la posibilidad adicional de aplicarlo a más de un campo.

Consulte la sección sobre la restricción de campo UNIQUE para más detalles.

Uso

Define una restricción de tabla UNIQUE en la tabla distributors:

```
CREATE TABLE distributors (
    did      DECIMAL(03),
    name     VARCHAR(40),
    UNIQUE(name)
);
```

Restricción PRIMARY KEY

```
[ CONSTRAINT name ] PRIMARY KEY ( column [, ...] )
```

Entradas

CONSTRAINT *name*

Un nombre arbitrario para la restricción.

column [, ...]

Los nombres de uno o más campos en la tabla.

Salidas

status

ERROR: Cannot insert a duplicate key into a unique index.

Esto ocurre en tiempo de ejecución si alguien intenta insertar un valor duplicado en un campo sujeto a una restricción de clave primaria (PRIMARY KEY).

Descripción

La restricción PRIMARY KEY especifica una regla en la que un grupo de uno o más campos de una tabla puede contener sólo valores únicos (no duplicados) y no nulos. Las definiciones de campo de los campos especificados no necesita incluir una restricción NOT NULL para ser incluida en una restricción PRIMARY KEY.

La restricción de tabla PRIMARY KEY es similar a la respectiva restricción de campo, con la posibilidad de extenderla with the additional capability of encompassing multiple columns.

Consulte la sección sobre la restricción de campo PRIMARY KEY para más información.

Uso

Crea las tablas films y distributors:

```
CREATE TABLE films (
    code        CHARACTER(5) CONSTRAINT firstkey PRIMARY KEY,
    title       CHARACTER VARYING(40) NOT NULL,
    did         DECIMAL(3) NOT NULL,
    date_prod   DATE,
    kind        CHAR(10),
    len         INTERVAL HOUR TO MINUTE
);

CREATE TABLE distributors (
    did         DECIMAL(03) PRIMARY KEY DEFAULT NEXTVAL('serial'),
    name        VARCHAR(40) NOT NULL CHECK (name <> "")
);
```

Crea una tabla con un array de 2 dimensiones:

```
CREATE TABLE array (
    vector INT[][]
);
```

Define una restricción de tabla UNIQUE para la tabla films. Las restricciones de tabla UNIQUE pueden ser definidas sobre uno o más campos de la tabla:

```
CREATE TABLE films (
    code        CHAR(5),
    title       VARCHAR(40),
    did         DECIMAL(03),
    date_prod   DATE,
    kind        CHAR(10),
    len         INTERVAL HOUR TO MINUTE,
    CONSTRAINT production UNIQUE(date_prod)
);
```


Define una restricción de campo CHECK:

```
CREATE TABLE distributors (
    did      DECIMAL(3) CHECK (did > 100),
    name     VARCHAR(40)
);
```

Define una restricción de tabla CHECK:

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40)
    CONSTRAINT con1 CHECK (did > 100 AND name > " )
);
```

Define una restricción de tabla PRIMARY KEY para la tabla films. Las restricciones de tabla PRIMARY KEY pueden ser definidas sobre uno o más campos de la tabla:

```
CREATE TABLE films (
    code     CHAR(05),
    title    VARCHAR(40),
    did      DECIMAL(03),
    date_prod DATE,
    kind     CHAR(10),
    len      INTERVAL HOUR TO MINUTE,
    CONSTRAINT code_title PRIMARY KEY(code,title)
);
```

Define una restricción de campo PRIMARY KEY para la tabla distributors. Las restricciones de campo PRIMARY KEY solamente se pueden definir para un campo de la tabla (los siguientes dos ejemplos serían equivalentes) :

```
CREATE TABLE distributors (
    did      DECIMAL(03),
    name     CHAR VARYING(40),
    PRIMARY KEY(did)
);
```

```
CREATE TABLE distributors (
    did      DECIMAL(03) PRIMARY KEY,
    name     VARCHAR(40)
);
```

Notas

CREATE TABLE/INHERITS es una extensión al lenguaje de Postgres.

Compatibilidad

SQL92

Además de la tabla temporal visible localmente, SQL92 define una sentencia CREATE GLOBAL TEMPORARY TABLE, y opcionalmente una cláusula ON COMMIT:

```
CREATE GLOBAL TEMPORARY TABLE table ( column type [
    DEFAULT value ] [ CONSTRAINT column_constraint ] [, ...] )
    [ CONSTRAINT table_constraint ] [ ON COMMIT { DELETE | PRESERVE } ROWS ]
```

Para tablas temporales, la sentencia CREATE GLOBAL TEMPORARY TABLE nombra una nueva tabla visible a otros clientes y define los campos de la tabla y las restricciones.

La cláusula opcional ON COMMIT de CREATE TEMPORARY TABLE especifica si la tabla temporal debe vaciarse de registros cada vez que se ejecuta un COMMIT o no. Si se omite la cláusula ON COMMIT, se asume la opción por defecto, ON COMMIT DELETE ROWS.

Para crear una tabla temporal:

```
CREATE TEMPORARY TABLE actors (
    id          DECIMAL(03),
    name        VARCHAR(40),
    CONSTRAINT actor_id CHECK (id < 150)
) ON COMMIT DELETE ROWS;
```

Cláusula UNIQUE

SQL92 especifica algunas posibilidades adicionales para UNIQUE:

Definición de restricción de tabla:

```
[ CONSTRAINT name ] UNIQUE ( column [, ...] )
    [ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
    [ [ NOT ] DEFERRABLE ]
```

Definición de restricción de campo:

```
[ CONSTRAINT name ] UNIQUE
    [ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
    [ [ NOT ] DEFERRABLE ]
```

cláusula NULL

La restricción NULL (realmente no es una restricción) es una extensión Postgres a SQL92 incluída por simetría con la cláusula NOT NULL. Como es el valor por defecto para cualquier campo, su presencia es redundante.

```
[ CONSTRAINT name ] NULL
```

cláusula NOT NULL

SQL92 especifica alguna posibilidad adicional para NOT NULL:

```
[ CONSTRAINT name ] NOT NULL
    [ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
    [ [ NOT ] DEFERRABLE ]
```

cláusula CONSTRAINT

SQL92 especifica alguna posibilidad adicional para restricciones, y también define restricciones de assertions y de dominio.

Nota: Postgres todavía no soporta ni dominios ni assertions.

Una assertion es un tipo especial de restricción de integridad y comparte el mismo espacio de nombres con otras restricciones. Sin embargo, una assertion no es necesariamente dependiente de una particular tabla base como son las restricciones, así que SQL-92 proporciona la sentencia CREATE ASSERTION como un método alternativo para definir una restricción:

```
CREATE ASSERTION name CHECK ( condition )
```

Las restricciones de dominio se definen con las sentencias CREATE DOMAIN o ALTER DOMAIN:

Restricción de dominio:

```
[ CONSTRAINT name ] CHECK constraint
    [ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
    [ [ NOT ] DEFERRABLE ]
```

Definición de restricciones de tabla:

```
[ CONSTRAINT name ] { PRIMARY KEY ( column, ... ) | FOREIGN KEY constraint | UNIQUE constraint | CHECK constraint }
    [ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
    [ [ NOT ] DEFERRABLE ]
```

Definición de restricciones de campo:

```
[ CONSTRAINT name ] { NOT NULL | PRIMARY KEY | FOREIGN KEY constraint | UNI-
QUE | CHECK constraint }
[ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
[ [ NOT ] DEFERRABLE ]
```

Una definición de restricción de integridad (CONSTRAINT) puede contener un atributo o cláusula DEFERRABLE y /o una cláusula del modo inicial de restricción, en cualquier orden.

NOT DEFERRABLE

significa que la restricción debe ser comprobada después de la ejecución de cada sentencia SQL.

DEFERRABLE

significa que la verificación del cumplimiento de la restricción puede ser aplazado hasta más tarde, pero no más tarde que el final de la actual transacción.

El modo de restricción para cada restricción tiene siempre un valor inicial por defecto que se establece para la restricción al principio de la transacción.

INITIALLY IMMEDIATE

significa que, desde el principio de la transacción, la restricción debe ser comprobada después de la ejecución de cada sentencia SQL.

INITIALLY DEFERRED

significa que, como se está al principio de la transacción, la comprobación de la restricción puede ser aplazada hasta más tarde, pero no más tarde que en el final de la actual transacción. O sea, que la restricción puede ser incumplida por alguna sentencia SQL en un punto intermedio de la transacción, pero no al final de la misma.

Cláusula CHECK

SQL92 especifica alguna capacidad adicional para CHECK tanto en la restricciones de tabla como en la de campo.

definición de restricción de tabla:

```
[ CONSTRAINT name ] CHECK ( VALUE condition )
[ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
[ [ NOT ] DEFERRABLE ]
```

definición de restricción de campo:

```
[ CONSTRAINT name ] CHECK ( VALUE condition )
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

cláusula PRIMARY KEY

SQL92 especifica algunas posibilidades adicionales para la PRIMARY KEY:

Definición de restricciones de tabla:

```
[ CONSTRAINT name ] PRIMARY KEY ( column [, ...] )
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

Definición de restricciones de campo:

```
[ CONSTRAINT name ] PRIMARY KEY
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

CREATE TABLE AS

Nombre

CREATE TABLE AS — Crea una nueva tabla

Synopsis

```
CREATE TABLE table [ (column [, ...] ) ]
    AS select_clause
```

Inputs

table

El nombre de una nueva tabla a ser creada.

column

El nombre de una columna. Se pueden especificar múltiples nombres de columna usando una lista de nombres de columna delimitada por comas.

select_clause

Una sentencia de consulta válida. Refiérase a **SELECT** para hallar una descripción de la sintaxis permitida.

Salidas

Refiérase a **CREATE TABLE** y a **SELECT** para hallar un sumario de posibles mensajes de salida.

Descripción

CREATE TABLE AS permite a una tabla ser creada a partir del contenido de una tabla existente. Equivale en funcionamiento a: *SELECT INTO*, pero quizás con una sintaxis más directa.

CREATE TRIGGER**Nombre**

CREATE TRIGGER — Crea un nuevo disparador

Synopsis

```
CREATE TRIGGER name
  { BEFORE | AFTER } { event
  [OR ...] } ON table
  FOR EACH { ROW | STATEMENT } EXECUTE PROCEDURE
  func
  ( arguments )
```

Entradas*name*

El nombre de un disparador existente.

table

El nombre de una tabla.

event

Uno entre INSERT, DELETE o UPDATE.

funcname

Una función suministrada por el usuario.

Salidas

CREATE

Se devuelve este mensaje si el disparador se ha creado con éxito.

Descripción

CREATE TRIGGER introducir un nuevo disparador en la base de datos actual. El disparador se asocia con la relación *relname* y ejecutar la función especificada *funcname*.

Se puede especificar que el disparador se dispare de cualquiera de estas dos formas: antes (BEFORE) de que la operación sea intentada en un registro (antes de que las restricciones se comprueben y **INSERT**, **UPDATE** o **DELETE** sean intentados) o después (AFTER) de que la operación haya sido intentada (por ejemplo después de que las restricciones sean comprobadas y de que **INSERT**, **UPDATE** o **DELETE** hayan sido completados). Si el disparador se pone en marcha antes del evento, éste puede saltar la operación para el registro actual o cambiar el registro que estaba insertándose (sólo para las operaciones **INSERT** y **UPDATE**). Si el disparador se dispara después del evento, todos los cambios, incluyendo la última inserción, actualización o borrado, son "visibles" para el disparador.

Refiérase a los capítulos de SPI y Triggers en la guía *PostgreSQL Programmer's Guide* para más información.

Notas

CREATE TRIGGER es una extensión del lenguaje Postgres.

Sólo el propietario relacionado puede crear un disparador en esta relación.

Hasta la versión actual (v6.4), las sentencias de disparadores no están implementadas.

Refiérase a **DROP TRIGGER** para obtener información sobre como borrar disparadores.

Uso

Comprueba si el código de distribuidor especificado existe en la tabla de distribuidores antes de añadir o actualizar una fila en los films de la tabla:

```
CREATE TRIGGER if_dist_exists
  BEFORE INSERT OR UPDATE ON films FOR EACH ROW
```

```
EXECUTE PROCEDURE check_primary_key ('did', 'distributors', 'did');
```

Antes de cancelar un distribuidor o de actualizar su código, borra cada referencia en los films de la tabla:

```
CREATE TRIGGER if_film_exists
  BEFORE DELETE OR UPDATE ON distributors FOR EACH ROW
  EXECUTE PROCEDURE check_foreign_key (1, 'CASCADE', 'did', 'films', 'did');
```

Compatibilidad

SQL92

No hay **CREATE TRIGGER** en SQL92.

El segundo ejemplo explicado anteriormente puede implementarse también usando una restricción de FOREIGN KEY (clave foránea) como en:

```
CREATE TABLE distributors (
  did      DECIMAL(3),
  name     VARCHAR(40),
  CONSTRAINT if_film_exists
  FOREIGN KEY(did) REFERENCES films
  ON UPDATE CASCADE ON DELETE CASCADE
);
```

En cualquier caso, las claves foráneas todavía no están implementadas (hasta la versión 6.5) en Postgres.

CREATE TYPE

Nombre

CREATE TYPE — Define un nuevo tipo de datos base

Synopsis

```
CREATE TYPE typename
( INPUT = input_function,
  OUTPUT = output_function
  , INTERNALLENGTH = { internallength
  | VARIABLE }
  [ , EXTERNALLENGTH = { externallength
```



```

    | VARIABLE } ]
[ , DEFAULT = "default" ]
[ , ELEMENT = element ]
[ , DELIMITER = delimiter ]
[ , SEND = send_function ]
[ , RECEIVE = receive_function ]
[ , PASSEDBYVALUE ] )

```

Entradas

typename

El nombre del tipo a ser creado.

internallength

Un valor literal, el cual especifica la longitud interna del nuevo tipo.

externallength

Un valor literal, el cual especifica la longitud externa del nuevo tipo.

input_function

El nombre de una función, creada mediante **CREATE FUNCTION**, la cual convierte los datos desde su forma externa a la forma interna de tipo.

output_function

El nombre de una función, creada mediante **CREATE FUNCTION**, la cual convierte los datos desde su forma interna a una forma conveniente para ser mostrados.

element

El tipo creado es un array; esto especifica el tipo de los elementos del array.

delimiter

El carácter delimitador para el array.

default

El texto por defecto que se mostrar para indicar "datos no presentes"

send_function

El nombre de una función, creada mediante **CREATE FUNCTION**, la cual convierte los datos de este tipo a una forma adecuada para ser transmitidos a otra máquina.

receive_function

El nombre de una función, creada mediante **CREATE FUNCTION**, la cual convierte los datos de este tipo a una forma adecuada para su transmisión desde otra máquina a la forma interna.

Salidas

```
CREATE
```

Mensaje que se devuelve si el tipo ha sido creado con éxito.

Descripción

CREATE TYPE permite al usuario registrar un nuevo tipo de datos de usuario con Postgres para ser usado en la base de datos actual. El usuario que define un tipo se convierte en su propietario. *typename* es el nombre de del nuevo tipo y debe ser único dentro de los tipos definidos para esta base de datos.

CREATE TYPE necesita el registro de dos funciones (usando `create function`) antes de definir el tipo. La representación de un nuevo tipo base está determinada por *input_function*, la cual convierte la representación externa del tipo a una representación interna, utilizable por los operadores y funciones definidas por el tipo. Naturalmente *output_function* ejecuta la transformación inversa. Ambas funciones, la de entrada y la de salida deben ser declaradas para asumir uno o dos argumentos de tipo "opaque".

Los nuevos tipos de datos base pueden ser de longitud fija, en cuyo caso *internallength* es un entero positivo, o también pueden ser de longitud variable, en cuyo caso, Postgres asume que el nuevo tipo tiene el mismo formato que el tipo de datos suministrado por Postgres, "text". Para indicar que un tipo es de longitud variable, se debe especificar *internallength* como `VARIABLE`. La representación externa se especifica de forma similar usando la palabra clave *externallength*.

Para indicar que un tipo es un array y para indicar que un tipo tiene elementos de array, indique el tipo del elemento del array usando la palabra clave `element`. Por ejemplo, para definir un array de enteros de cuatro bytes ("int4"), especifique

```
ELEMENT = int4
```

Para indicar el delimitador a ser usado en arrays de este tipo, *delimiter* se puede fijar a un carácter específico. El delimitador por defecto es la coma (",").

Opcionalmente, hay un valor por defecto disponible en caso de que un usuario quiera algún patrón de bit específico para expresar "datos no presentes." Especifique el valor por defecto con la palabra clave `DEFAULT`.

BEGIN RATIONALE: Cómo el usuario especifica este patrón de bit y lo asocia con el hecho de que los datos no estén presentes>END RATIONALE:

Los argumentos opcionales *send_function* y *receive_function* son usados cuando el programa de aplicación que demanda los servicios Postgres reside en una máquina diferente. En este caso, la máquina en la cual se ejecuta Postgres puede usar un formato para el tipo de datos diferente del usado en la máquina remota. En este caso es conveniente convertir los items de datos a una forma estándar cuando se envíen desde el servidor al cliente y convertirlos del formato estándar al específico de la máquina cuando el servidor recibe los datos desde el cliente. Si estas funciones no están especificadas, se asume que el formato interno del tipo es aceptable en todas las arquitecturas de máquina relevantes. Por ejemplo, los caracteres simples no se tienen que convertir si se pasa desde un Sun-4 a un DECstation, pero muchos otros tipos sí.

El flag opcional, `PASSEDBYVALUE`, indica que los operadores y funciones que usan este tipo de datos deben pasar los argumentos preferentemente por valor que por referencia. Dese cuenta de que no pasar; a por valor tipos cuya representación interna es de más de cuatro bytes.

Para nuevos tipos base, un usuario puede definir operadores, funciones y conjuntos usando las facilidades apropiadas descritas en esta sección.

Tipos de Array

Existen dos funciones generalizadas incorporadas, `array_in` y `array_out` para la creación rápida de tipos de array de longitud variable. Estas funciones operan en arrays de cualquier tipo Posgres existente.

Tipos de objetos grandes

Un tipo Posgres regular sólo puede ser de longitud 8192 bytes. Si necesita un tipo mayor debe crear un tipo de objeto grande. El interface para estos tipos está ampliamente explicado en *The PostgreSQL Programmer's Guide*. La longitud de todos los tipos de objeto grande es siempre `VARIABLE`.

Ejemplos

Este comando crea el tipo de datos `box` y después usa el tipo en una definición de clase:

```
CREATE TYPE box (INTERNALLENGTH = 8,
    INPUT = my_procedure_1, OUTPUT = my_procedure_2);
CREATE TABLE myboxes (id INT4, description box);
```

Este comando crea un tipo array de longitud variable con elementos enteros:

```
CREATE TYPE int4array (INPUT = array_in, OUTPUT = array_out,
    INTERNALLENGTH = VARIABLE, ELEMENT = int4);
CREATE TABLE myarrays (id int4, numbers int4array);
```

Este comando crea un tipo de objeto grande y lo usa en una definición de clase:

```
CREATE TYPE bigobj (INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE);
CREATE TABLE big_objs (id int4, obj bigobj);
```

Notas

Los nombres de tipos no pueden empezar por el carácter guión bajo ("`_`") y sólo pueden tener una longitud de 31 caracteres. Esto es debido a que Posgres crea sin infor-

mar un tipo array para cada tipo base con un nombre que consiste en el nombre del tipo base precedido de un guión bajo.

Refiérase a **DROP TYPE** para borrar un tipo existente.

Vea también **CREATE FUNCTION**, **CREATE OPERATOR** y el capítulo de Objetos Grandes, 'Large Objects', en *PostgreSQL Programmer's Guide*.

Compatibilidad

SQL3

CREATE TYPE es un elemento de SQL3.

CREAR USUARIO

Nombre

CREAR USUARIO — Creando un nuevo usuario de base de datos

Synopsis

```
CREAR USUARIO nombre de usuario
[ CON
  [ SYSID uid ]
  [ PASSWORD 'palabra clave' ] ]
[ CREARDB | NOCREARDB ] [ CREARUSUARIO | NOCREARUSUARIO ]
[ EN EL GRUPO nombre de grupo [, ...] ]
[ VALIDO HASTA 'abstime' ]
```

Entradas

nombre de usuario

El nombre del usuario.

uid

La orden **SYSID** puede ser usada para escoger el identificador de usuario PostgreSQL del usuario que se esta creando. No es nada necesario que corresponda a los identificadores de usuarios de UNIX, pero algunas personas eligen mantener los números iguales.

Si no se especifica, se usará por defecto el número más alto asignado más uno.

palabra clave

Pide la palabra clave del usuario. Si no va a usar autenticación por palabra clave puede omitir esta opción, de otra manera el usuario no será capaz de conectar con el servidor de autenticación de palabras clave. Mire en `pg_hba.conf(5)` o la Guía del administrador para más detalles de como usar mecanismos de autenticación.

CREATEDB
NOCREATEDB

Estas órdenes definen la capacidad de un usuario para crear bases de datos. Si se especifica **CREATEDB**, el usuario definido tendrá permiso para crear sus propias bases de datos. Usando **NOCREATEDB** se denegará a un usuario la capacidad de crear bases de datos. Si se omite esta orden, **NOCREATEDB** se usa por defecto.

CREATEUSER
NOCREATEUSER

Estas ordenes determinan si a un usuario se le permitirá crear nuevos usuarios. Esta opción hará del usuario un superusuario que podrá pasar por encima de todas las restricciones de acceso. Si se omite esta orden se cogerá la orden de **NOCREATEUSER** como valor por defecto del usuario.

nombre de grupo

El nombre de un grupo dentro del cual se coloca al usuario como un nuevo miembro.

abstime

La orden **VALIDO HASTA** pone un valor absoluto a la fecha en la que la palabra clave del usuario pierde su validez. Si se omite esta orden el login valdrá para siempre.

Resultados

CREAR USUARIO

Mensaje devuelto si el comando se completa satisfactoriamente.

Descripción

CREAR USUARIO añadirá un nuevo usuario a un ejemplo de PostgreSQL. Véase la Guía del Administrador para más información sobre el manejo de usuarios y la autenticación. Debe ser un superusuario de bases de datos para usar este comando.

Use **MODIFICAR USUARIO** para cambiar la palabra clave y los privilegios de un usuario, y **DROP USER** para eliminar a un usuario. Use **MODIFICAR GRUPO** para añadir o eliminar a un usuario de otros grupos. PostgreSQL viene con un script *createuser* que tiene la misma funcionalidad que este comando (de hecho, llama a este comando) pero puede ser ejecutado desde la línea de comandos.

Modo de uso

Crear un usuario sin palabra clave:

```
CREAR USUARIO jonathan
```

Crear un usuario con palabra clave:

```
CREAR USUARIO david CON PALABRA CLAVE 'jw8s0F4'
```

Crear un usuario con una palabra clave, cuya cuenta es válida hasta el final del 2001. Notese que un segundo dentro del año 2002 la cuenta no es valida:

```
CREAR USUARIO miriam CON PALABRA CLAVE 'jw8s0F4' VALIDA HASTA '1 En 2002'
```

crear una cuenta con la que el usuario pueda crear bases de datos:

```
CREAR USUARIO manuel CON PALABRA CLAVE 'jw8s0F4' CREARDB
```

Compatibilidad

SQL92

No existe la orden **CREATE USER** en SQL92.

CREAR VISTA

Nombre

CREAR VISTA — Construir una tabla virtual

Synopsis

```
CREAR VISTA vista COMO SELECCIONADO query
```

Entradas

vista

El nombre de la vista que se va a crear.

consulta

Una consulta en SQL indica las columnas y filas de la vista.

Dirijase a la orden **SELECCIONAR** para más información sobre los argumentos válidos.

Resultados

CREADA

El mensaje recibido si la vista se crea satisfactoriamente.

ERROR: Relación 'view' ya existe

Este error ocurre si la vista especificada ya existe en la base de datos.

AVISO creado: el nombre atribuido "column" tiene un caracter desconocido

La vista será creada teniendo una columna con un carácter desconocido si usted no lo especifica. Por ejemplo, el siguiente comando da un error:

```
CREAR VISTA vista COMO SELECCIONADO 'Hola Mundo'
```

mientras que este comando no lo hace:

```
CREAR VISTA vista COMO SELECCIONADO 'Hola Mundo'::texto
```

Descripción

CREAR VISTA definirá una vista de una tabla o class. Esta vista no se materializa físicamente. Específicamente, una consulta reescrita genera automáticamente una regla para mantener las operaciones ejecutadas en la vista.

Notas

Normalmente, las vista son de sólo lectura.

Use la orden **TIRAR VISTA** para deshacerse de la vista.

Modo de uso

Crear una vista conteniendo todas las películas de Comedia:

```
CREAR VISTA clases COMO
  SELECCIONAR *
  DESDE películas
  DONDE clase = 'Comedia';

SELECCIONAR * DESDE clases;
```

codigo	título	did	date_prod	Clase	Dur
UA502	Bananas	105	13-07-1971	Comedia	01:22
C_701	There's a Girl in my Soup	107	11-06-1970	Comedia	01:36

Compatibilidad

SQL92

SQL92 especifica algunas capacidades específicas para la orden **CREAR VISTA** :

```

CREAR VISTA view [ columna [, ...] ]
    COMO SELECCIONADO expresión [ COMO nombre de columna ] [, ...]
    DESDE tabla [ DONDE condición ]
    [ CON [ CASCADA | LOCAL ] COMPROBAR OPCION ]

```

Las clausulas opcionales para todos los comandos SQL92 son:

COMPROBAR OPCION

Esta opción es para hacer vistas renovables. Todos los INSERTAR Y RENOVAR en la vista serán comprobados para asegurar que los datos satisfacen las condiciones definidas en la tabla. Si no lo cumplen, la renovación no será ejecutada.

LOCAL

Comprobar la integridad de esta vista.

CASCADA

Comprobar la integridad de esta vista y cualquier vista dependiente. CASCADA se asume si ni CASCADA ni LOCAL son especificadas.

DECLARE

Nombre

DECLARE — Define un cursor para acceso a una tabla

Synopsis

```

DECLARE cursorname [ BINARY ] [ INSENSITIVE ] [ SCROLL
] CURSOR FOR query [ FOR { READ
ONLY | UPDATE [ OF column [, ...]

```


]]

Inputs

cursorname

El nombre del cursor a ser usado en subsecuentes operaciones FETCH..

BINARY

Provoca que el cursor traiga datos en formato binario en vez de formato texto.

INSENSITIVE

SQL92 palabra clave indicando que los datos recuperados del cursor no deben ser afectados por actualizaciones desde otros procesos o cursores. Ya que la operación de los cursores ocurre dentro de las transacciones, en Postgres este siempre es el caso. Esta palabra clave no tiene efecto.

SCROLL

SQL92 palabra clave indicando que los datos deben ser recuperados en múltiples filas por cada operación FETCH. Ya que esto es siempre permitido por Postgres esta palabra clave no tiene efecto.

query

Una consulta SQL la cual proveera las filas a ser gobernadas por el cursor. Referirse al comando SELECT para mayor información acerca de los argumentos válidos.

READ ONLY

SQL92 palabra clave indicando que el cursor sera usado en modo solo lectura. Ya que este es el único modo de acceso de cursor disponible en Postgres esta palabra clave no tiene efecto.

UPDATE

SQL92 palabra clave indicando que el cursor sera usado para actualizar tablas. Ya que la actualización de cursores no esta actualmente soportada en Postgres esta palabra clave provoca un mensaje de error informativo.

column

Columna(s) a ser actualizadas. Ya que la actualización de cursores no esta actualmente soportada en Postgres la clausula UPDATE provoca un mensaje de error informativo.

Outputs

SELECT

El mensaje devuelto si el SELECT es ejecutado exitosamente.

```
NOTICE BlankPortalAssignName: portal"cursorname" already exists
```

Este error ocurre si *cursorname* ya esta declarado.

```
ERROR: Named portals may only be used in begin/endtransaction blocks
```

Este error ocurre si el cursor no esta declarado dentro de un transaction block.

Description

DECLARE permite a un usuario crear cursores, los cuales pueden ser usados para recuperar un pequeño número de filas a la vez provenientes de una consulta mas extensa. Los cursores pueden devolver datos ya sea en formato de texto o en foemato binario *FETCH*.

Los cursores comunes retornan datos en formato texto, ya sea ASCII u otro esquema de codificacion dependiendo en como el Postgres backend fue creado. Ya que los datos estan guardados nativamente en formato binario, el sistema debe hacer una conversión para producir formato texto. Ademas, los formatos de texto son a menudo mayores en tamaño que sus correspondientes en formato binario. Una vez que la información viene en formato texto, la aplicación cliente podria necesitar convertirlos a un formato binario para manipularlos. los cursores BINARY devuelven los datos en una representación binaria nativa.

Como ejemplo, si una consulta devuelve un valor de uno desde una columna integer, usted obtendria un string de '1' con un cursor default mientras que con un cursor binario usted obtendria un valor 4-byte igual a un control-A ('^A').

Los cursores BINARY deben ser usados cuidadosamente. Aplicaciones de usuario tales como *psql* no son conscientes de los cursores binarios y esperan que los datos vengan en formato texto.

La representación de los string es neutral respecto a la arquitectura, mientras que la representación binaria puede diferir entre diferentes arquitecturas de máquinas y *Postgres no resuelve el ordenamiento de bytes o las cuestiones de representacion para los cursores binarios*. Por consiguiente, si su máquina cliente y su máquina servidor usa diferentes representaciones (e.g. "big-endian" contra "little-endian"), probablemente usted no deseara sus datos devueltos en formato binario. Sin embargo, los cursores binarios pueden ser un poco más eficientes ya que hay menos overhead debido a la conversión en la transferencias de datos del servidor al cliente.

Sugerencia: Si usted pretende mostrar los datos en ASCII, recuperarlos en ASCII le ahorraran un poco de esfuerzo del lado cliente.

Notes

Los cursores solo estan disponibles en las transacciones. Usar para *BEGIN*, *COMMIT* y *ROLLBACK* para definir un transaction block.

En SQL92 los cursores estan disponibles solo en aplicaciones SQL (ESQL) embebidas. EL Postgres backend no implementa un comando explicito **OPEN cursor** ; un cursor se considera abierto cuando este es declarado. Sin embargo, *ecpg*, el preprocesador embebido de SQL para Postgres, soporta la convención de cursores SQL92 , incluyendo aquellos que involucran los comandos **DECLARE** y **OPEN**.

Uso

Para declarar un cursor:

```
DECLARE liahona CURSOR FOR SELECT * FROMfilms;
```

Compatibilidad

SQL92

SQL92 permite cursores solo en SQL embebido y en módulos. Postgres permite cursores para ser usados en forma interactiva. SQL92 permite cursores embebidos o modulares para actualizar información de la base de datos. Todos los cursores Postgres son de solo lectura. La palabra clave BINARY es una extensión de Postgres.

DELETE

Nombre

DELETE — Borra filas de una tabla

Synopsis

```
DELETE FROM table [ WHERE condition ]
```

Inputs

table

El nombre de una tabla existente.

condition

Esta es una consulta SQL de selección la cual devuelve las filas a ser borradas.

Referirse al comando SELECT para una mayor descripción de la clausula WHERE.

Outputs

`DELETE count`

Mensaje devuelto si los items son borrados exitosamente. El valor *count* es la cantidad de filas borradas.

Si *count* es 0, ninguna fila fue borrada.

Description

DELETE borra las filas que satisfacen la clausula **WHERE** de la tabla especificada.

Si la *condicion* (clausula **WHERE**) esta ausente, el efecto es borrar todas las filas de la tabla. El resultado es una tabla valida, pero vacia.

Sugerencia: *TRUNCATE* es una extensión de Postgres el cual provee un mecanismo más rápido para borrar todas las filas de una tabla.

Para modificar la tabla usted debe poseer acceso de escritura a la misma, asi como acceso de lectura a cualquier tabla cuyos valores son leidos en la *condicion*.

Uso

Borra todos los films excepto los musicales:

```
DELETE FROM films WHERE kind <> 'Musical';
SELECT * FROM films;
```

code	title	did	date_prod	kind	len
UA501	West Side Story	105	1961-01-03	Musical	02:32
TC901	The King and I	109	1956-08-11	Musical	02:13
WD101	Bed Knobs and Broomsticks	111		Musical	01:57

(3 rows)

Borra completamente la tabla films:

```
DELETE FROM films;
SELECT * FROM films;
```

code	title	did	date_prod	kind	len
------	-------	-----	-----------	------	-----

(0 rows)

Compatibility

SQL92

SQL92 permite un comando DELETE posicionado:

```
DELETE FROM table WHERE CURRENT OF cursor
```

donde *cursor* corresponde a un cursor abierto. En Postgres los cursores interactivos son de solo-lectura.

DROP AGGREGATE

Nombre

DROP AGGREGATE — Elimina la definición de una función agregada

Synopsis

```
DROP AGGREGATE name type
```

Entradas

name

El nombre de una función de agregado existente.

type

El tipo de una función de agregado existente. (Véase la *PostgreSQL User's Guide* para más información sobre los tipos de datos).

*BEGIN RATIONALE: Esto debería ser una referencia cruzada más que un punto de un capítulo*END RATIONALE:

Salidas

DROP

Mensaje devuelto si el comando se ejecuta satisfactoriamente.

```
WARN RemoveAggregate: aggregate 'agg' for 'type' does not exist
```

Este mensaje aparece si la función agregada especificada no existe en la base de datos.

Descripción

DROP AGGREGATE eliminará todas las referencias a la definición de una función de agregado existente. Para ejecutar esta orden el usuario actual debe ser el propietario del agregado.

Notas

Use *CREATE AGGREGATE* para crear funciones de agregado.

Uso

Para eliminar el agregado `myavg` de tipo `int4`:

```
DROP AGGREGATE myavg int4;
```

Compatibilidad

SQL92

No existe la sentencia **DROP AGGREGATE** en SQL92; la sentencia es una extensión de lenguaje de Postgres.

DROP DATABASE

Nombre

`DROP DATABASE` — Elimina una base de datos existente

Synopsis

```
DROP DATABASE name
```

Entradas

name

El nombre de una base de datos existente que se desea eliminar.

Salidas

`DROP DATABASE`

Este mensaje se devuelve si la orden se ejecuta satisfactoriamente.

`ERROR: user 'username' is not allowed to create/drop databases`

Debe tener el privilegio especial `CREATEDB` para eliminar bases de datos. Ver *CREAR USUARIO*.

`ERROR: dropdb: cannot be executed on the template database`

La base de datos `template1` no puede ser eliminada. No es conveniente hacerlo.

`ERROR: dropdb: cannot be executed on an open database`

NO puede conectarse a la base de datos que quiere eliminar. En su lugar, ha de conectar a `template1` o cualquier otra base de datos, y ejecutar el comando de nuevo.

`ERROR: dropdb: database 'name' does not exist`

Este mensaje ocurre si la base de datos especificada no existe.

`ERROR: dropdb: database 'name' is not owned by you`

Debe ser el propietario de la base de datos. Ser el propietario normalmente significa que también la ha creado.

`ERROR: dropdb: May not be called in a transaction block.`

Ha de completar primero la transacción en progreso antes de poder ejecutar este comando.

`NOTICE: The database directory 'xxx' could not be removed.`

la base de datos fué eliminada (a menos que haya aparecido otro mensaje de error), pero el directorio donde se almacenaban los datos no pudo ser eliminado. Debe borrarlo manualmente.

Descripción

`DROP DATABASE` elimina las entradas de catálogo de una base de datos existente y borra el directorio que contiene los datos. Solamente puede ser ejecutado por el propietario de la base de datos (normalmente quien la creó).

Notas

Esta orden no puede ser ejecutada mientras se está conectado a la base de datos objetivo. Por lo tanto, puede ser más conveniente usar el shell script *dropdb*, que emplea este comando.

Véase Refer to *CREATE DATABASE* para más información sobre como crear una base de datos.

Compatibilidad

SQL92

La sentencia **DROP DATABASE** es una extensión de lenguaje de Postgres; no existe ese comando en SQL92.

DROP FUNCTION

Nombre

`DROP FUNCTION` — Elimina una función de usuario escrita en C

Synopsis

```
DROP FUNCTION name ( [ type [, ...] ] )
```

Entradas

name

El nombre de una función existente.

type

El tipo de los parámetros de la función.

Salidas

`DROP`

Mensaje devuelto si la orden se completa satisfactoriamente.

`WARN RemoveFunction: Function "name" ("types") does not exist`

Este mensaje se obtiene si la función especificada no existe en la base de datos actual.

Descripción

DROP FUNCTION eliminará las referencias a una función C existente. Para ejecutar esta orden el usuario debe ser el propietario de la función. Los tipos de argumentos de entrada de la función han de especificarse, dado que solo la función con el nombre dado, y los tipos de argumentos dados se eliminará.

Notas

Véase *CREATE FUNCTION* para más información sobre la creación de funciones de agregado.

No se hacen comprobaciones para verificar los tipos de datos, operadores o método de acceso relacionados con la función que ha de eliminarse.

Uso

Esta orden elimina la función raíz cuadrada:

```
DROP FUNCTION sqrt(int4);
```

Compatibilidad

SQL92

DROP FUNCTION es una extensión de lenguaje de Postgres.

SQL/PSM

SQL/PSM es un estandar propuesto para habilitar la extensibilidad de la funciones. La sentencia DROP FUNCTION de SQL/PSM tienen la siguiente sintaxis:

```
DROP [ SPECIFIC ] FUNCTION name { RESTRICT | CASCADE }
```

DROP GROUP

Nombre

DROP GROUP — Elimina un grupo

Synopsis

```
DROP GROUP name
```

Entradas

name

El nombre de un grupo existente.

Salidas

```
DROP GROUP
```

El mensaje devuelto si es grupo es eliminado satisfactoriamente.

Descripción

DROP GROUP elimina el grupo especificado de la base de datos. Los usuarios del grupo no se eliminan.

Use *CREATE GROUP* para añadir nuevos grupos, y *MODIFICAR GRUPO* para cambiar la pertenencia a un grupo.

Uso

Para eliminar un grupo:

```
DROP GROUP staff;
```

Compatibilidad

SQL92

No existe el comando **DROP GROUP** en SQL92.

DROP INDEX

Nombre

`DROP INDEX` — Elimina un índice de la base de datos

Synopsis

```
DROP INDEX index_name
```

Entradas

index_name

El nombre del índice a eliminar.

Salidas

`DROP`

El mensaje devuelto si el índice es eliminado satisfactoriamente.

```
ERROR: index "index_name" nonexistent
```

Este mensaje tiene lugar si *index_name* no es un índice de la base de datos.

Descripción

`DROP INDEX` elimina un índice existente del sistema de base de datos. Quien ejecute este comando, ha de ser el propietario del índice.

Notas

`DROP INDEX` es una extensión del lenguaje de Postgres.

Véase Refer to *CREATE INDEX* para más información sobre como crear índices.

Uso

Este comando eliminará el índice `title_idx`:

```
DROP INDEX title_idx;
```

Compatibilidad

SQL92

SQL92 define comandos con los que acceder a una base de datos relacional genérica. Los índices son una característica dependiente de la implementación, por lo que no existe comandos o de finiciones específicos para los índices en el lenguaje SQL92.

DROP LANGUAGE

Nombre

`DROP LANGUAGE` — Elimina un lenguaje procedural definido por el usuario

Synopsis

```
DROP PROCEDURAL LANGUAGE 'name'
```

Entradas

name

El nombre de un lenguaje procedural existente.

Salidas

`DROP`

Este mensaje es devuelto si el lenguaje es eliminado satisfactoriamente.

`ERROR: Language "name" doesn't exist`

Este mensaje tiene lugar si el lenguaje llamado *name* no se encuentra en la base de datos.

Descripción

`DROP PROCEDURAL LANGUAGE` eliminará la definición del lenguaje procedural llamdo *name*, previamente registrado.

Notas

La sentencia **DROP PROCEDURAL LANGUAGE** es una extensión de lenguaje de Postgres.

Véase Refer to *CREATE LANGUAGE* para más información sobre como crear lenguajes procedurales.

No se realiza ninguna comprobación acerca de si existen funciones o procedimientos desencadenados por eventos escritos en este lenguaje. Para re-habilitarlos sin tener que eliminar y recrear todas las funciones, el tributo `pg_proc's prolang` de las funciones ha de ser ajustado para el nuevo identificador de objeto de la entrada `pg_language` del lenguaje procedural nuevamente creado.

Uso

Este comando elimina el lenguaje PL/Sample:

```
DROP PROCEDURAL LANGUAGE 'plsample';
```

Compatibilidad

SQL92

No existe el comando **DROP PROCEDURAL LANGUAGE** en SQL92.

DROP OPERATOR

Nombre

DROP OPERATOR — Quita un operador de la base de datos

Synopsis

```
DROP OPERATOR id ( type | NONE [,...] )
```

Entradas

id

El identificador de un operador existente.

type

El tipo de los parámetros de la función.

Salidas

DROP

Mensaje devuelto si la operación es exitosa.

```
ERROR: RemoveOperator: binary operator 'oper' taking 'type' and 'type2'
does not exist
```

Este mensaje se muestra si el operador binario especificado no existe.

```
ERROR: RemoveOperator: left unary operator 'oper' taking 'type' does not
exist
```

Este mensaje se muestra si el operador unario izquierdo especificado no existe.

```
ERROR: RemoveOperator: right unary operator 'oper' taking 'type' does
not exist
```

Este mensaje se muestra si el operador unario derecho especificado no existe.

Description

DROP OPERATOR quita un operador de la base de datos. Para ejecutar este comando usted debe ser el propietario del operador.

La calidad de derecho o izquierdo de un operador unario izquierdo o derecho, respectivamente, puede ser especificada como **NONE**.

Notas

La declaración **DROP OPERATOR** es una extensión de lenguaje de Postgres.

Consulte *CREATE OPERATOR* por información sobre cómo crear operadores.

Es responsabilidad del usuario remover cualquier método de acceso y clases de operador que dependan del operador que se quitó.

Utilización

Quita el operador de potencia a^n para `int4`:

```
DROP OPERATOR ^ (int4, int4);
```

Quita el operador unario izquierdo de negación (`b !`) para expresiones booleanas:

```
DROP OPERATOR ! (none, bool);
```

Quita el operador unario derecho de factorial (! i) para int4:

```
DROP OPERATOR ! (int4, none);
```

Compatibilidad

SQL92

No existe un comando **DROP OPERATOR** en SQL92.

DROP RULE

Nombre

DROP RULE — Quita una regla existente de la base de datos

Synopsis

```
DROP RULE name
```

Entradas

name

El nombre de una regla existente para quitar.

Salidas

DROP

Mensaje devuelto en caso de que la operación sea exitosa.

```
ERROR: RewriteGetRuleEventRel: rule "name" not found
```

Este mensaje se muestra si la regla especificada no existe.

Descripción

DROP RULE quita una regla del sistema de reglas de Postgres especificado. Postgres dejará de aplicarla inmediatamente y quitará su definición de los catálogos del sistema.

Notas

La declaración **DROP RULE** es una extensión de lenguaje de Postgres.

Consulte **CREATE RULE** para información sobre cómo crear reglas.

Una vez que se quita una regla, el acceso a la información histórica que la regla haya escrito puede desaparecer.

Utilización

Para quitar la regla de reescritura `newrule`:

```
DROP RULE newrule;
```

Compatibilidad

SQL92

No existe **DROP RULE** en SQL92.

DROP SEQUENCE

Nombre

`DROP SEQUENCE` — Quita una secuencia existente

Synopsis

```
DROP SEQUENCE name [, ...]
```


Entradas

name

El nombre de una secuencia.

Salidas

DROP

Mensaje devuelto si la secuencia se elimina exitosamente.

WARN: Relation "*name*" does not exist.

Este mensaje se muestra si la secuencia especificada no existe.

Descripción

DROP SEQUENCE quita una secuencia generadora de números de la base de datos. Con la actual implementación de las secuencias como tablas especiales, trabaja igual que la declaración **DROP TABLE**.

Notas

La declaración **DROP SEQUENCE** es una extensión de lenguaje de Postgres.

Consulte la declaración **CREATE SEQUENCE** para obtener información sobre cómo crear una secuencia.

Utilización

Para quitar la secuencia `serial` de la base de datos:

```
DROP SEQUENCE serial;
```

Compatibilidad

SQL92

No existe **DROP SEQUENCE** en SQL92.

DROP TABLE

Nombre

`DROP TABLE [Eliminar Tabla]` — Elimina tablas de una base de datos

Synopsis

`DROP TABLE nombre [, ...]`

Entradas

nombre

El nombre de una tabla vista existente para eliminarla.

Salidas

DROP

El mensaje devuelto si el comando concluyo exitosamente.

ERROR Relation "*nombre*" Does Not Exist!

Si la tabla o vista especificada no existe en la base de datos.

Descripción

DROP TABLE elimina tablas y vistas de una base de datos. Solo su propietario (owner) puede destruir una tabla o vista. Una tabla puede ser vaciada de sus filas, pero no destruida, usando **DELETE**.

Si una tabla a ser destruida tiene un índice secundario, este debe ser removido primero. La remoción de solo un índice secundario no afecta el contenido de la tabla subyacente.

Notas

Consultar en **CREATE TABLE** y **ALTER TABLE** para información sobre como crear o modificar tablas.

Uso

Para destruir dos tablas, cintas y **distribuidores**:

```
DROP TABLE cintas, distribuidores;
```

Compatibilidad

SQL92

SQL92 especifica algunas capacidades adicionales a DROP TABLE:

```
DROP TABLE table { RESTRICT | CASCADE }
```

RESTRICT

Asegura que solo una tabla sin vistas dependientes o restricciones de integridad pueda ser destruida.

CASCADE

Cualquier vista o restricción de integridad sería también eliminada.

Sugerencia: Por el momento, para eliminar una vista dependiente se debe eliminar esta explícitamente.

DROP TRIGGER

Nombre

DROP TRIGGER — Borra la definición de un disparador

Synopsis

```
DROP TRIGGER nombre ON tabla
```

Entradas

nombre

El nombre de un disparador existente.

tabla

El nombre de una tabla.

Salidas

DROP

Mensaje devuelto si el disparador se borró correctamente.

ERROR: DropTrigger: there is no trigger name on relation "*table*"

Este mensaje se da cuando el disparador especificado no existe.

Descripción

DROP TRIGGER borrará todas las referencias existentes a la definición de un disparador. Para poder ejecutar este comando el usuario actual debe ser el propietario del disparador.

Notas

DROP TRIGGER es una extensión del lenguaje de Postgres.

Consulte **CREATE TRIGGER** para obtener información acerca de cómo crear disparadores (triggers).

Utilización

Destruye el disparador `if_dist_exists` en la tabla `films`:

```
DROP TRIGGER if_dist_exists ON films;
```

Compatibilidad

SQL92

No existe ninguna declaración **DROP TRIGGER** en SQL92.

DROP TYPE

Nombre

`DROP TYPE` — Retira un tipo, definido por el usuario, de los catálogos del sistema

Synopsis

```
DROP TYPE tipo
```

Entradas

tipo

El nombre del tipo catalogado.

Salidas

`DROP`

El mensaje que se obtiene si el comando ha sido ejecutado con éxito.

```
ERROR: RemoveType: type 'tipo' does not exist
```

Este mensaje ocurre si el tipo dado no ha sido encontrado.

Descripción

DROP TYPE retira un tipo, definido por el usuario, de los catálogos del sistema.

Un tipo puede ser retirado únicamente por su dueño.

Notas

La cláusula `DROP TYPE` es una extensión del lenguaje Postgres.

Consulte el comando **CREATE TYPE** para obtener información sobre como crear tipos.

Es responsabilidad del autor retirar cualquier operador, función, agregado, método de acceso, subtipo y clase que usen el tipo que ha sido borrado.

Si se retira un tipo predefinido, el comportamiento del servidor será impredecible.

Uso

Para retirar el tipo `caja`:

```
DROP TYPE caja;
```

Compatibilidad

SQL3

`DROP TYPE` es una cláusula de SQL3.

DROP USER

Nombre

`DROP USER` — Retira un usuario

Synopsis

```
DROP USER nombre
```

Entradas

nombre

El nombre de un usuario existente.

Salidas

```
DROP USER
```

El mensaje que se obtiene si el usuario ha sido retirado con éxito.

```
ERROR: DROP USER: user "nombre" does not exist
```

Este mensaje ocurre si no ha sido encontrado el usuario dado.

```
DROP USER: user "nombre" owns database "base_datos", cannot be removed
```

Deberá eliminar primero la base de datos perteneciente al usuario, o modificar su propietario, antes de poder retirar al usuario.

Descripción

DROP USER retira de la base de datos el usuario dado. No retira tablas, vistas u otros objetos que pertenezcan al usuario. Si el usuario es dueño de una base de datos, se producirá un error.

Use *CREAR USUARIO* para adicionar nuevos usuarios, y *MODIFICAR USUARIO* para modificar las propiedades de un usuario. PostgreSQL viene con un guión *dropuser* que tiene la misma función de este comando (de hecho, invoca este comando) pero que puede ser ejecutado desde la shell.

Uso

Para eliminar la cuenta de un usuario:

```
DROP USER juan;
```

Compatibilidad

SQL92

No existe comando **DROP USER** en SQL92.

DROP VIEW

Nombre

DROP VIEW — Retira una vista definida en una base de datos

Synopsis

```
DROP VIEW nombre
```

Entradas

nombre

El nombre de la vista definida.

Salidas

DROP

El mensaje que se obtiene si el comando ha sido ejecutado con éxito.

ERROR: RewriteGetRuleEventRel: rule "_RETnombre" not found

Este mensaje ocurre si la vista dada no existe en la base de datos.

Descripción

DROP VIEW retira una vista definida en una base de datos. Para poder ejecutar este comando, deberá ser el dueño de la vista.

Notas

La cláusula **DROP TABLE** de Postgres también elimina vistas.

Consulte **CREATE VIEW** para una explicación de como se crean vistas.

Uso

Este comando retirará la vista llamada *variedades*:

```
DROP VIEW variedades;
```

Compatibilidad

SQL92

SQL92 especifica algunas funcionalidades adicionales para **DROP VIEW**:

```
DROP VIEW vista { RESTRICT | CASCADE }
```


Entradas**RESTRICT**

Asegura que sean destruidas únicamente vistas sin otras listas dependientes y sin restricciones de integridad.

CASCADE

Cualquier vista que se refiera a esta será también eliminada, al igual que cualquier restricción de integridad.

Notas

Actualmente, para retirar una vista referida en una base de datos Postgres esta debe ser eliminada explícitamente.

END**Nombre**

END — Lleva a cabo la transacción actual

Synopsis

END [WORK | TRANSACTION]

Entradas

WORK
TRANSACTION

Palabras clave opcionales. No tienen ningún efecto.

Salidas

COMMIT

Es el mensaje que se devuelve si la transacción se ha llevado a cabo correctamente.

```
NOTICE: COMMIT: no transaction in progress
```

Se da cuando no hay ninguna transacción en curso.

Descripción

END es un sinónimo de PostgreSQL para *COMMIT*.

Notas

Las palabras clave *WORK* y *TRANSACTION* son "ruidosas" y pueden ser omitidas.

Use *ROLLBACK* para abortar una transacción.

Utilización

Para hacer que todos los cambios sean permanentes:

```
END WORK;
```

Compatibilidad

SQL92

END es una extensión de PostgreSQL que proporciona una funcionalidad equivalente a *COMMIT*.

EXPLAIN

Nombre

EXPLAIN — Muestra el plan de ejecución de la sentencia

Synopsis

```
EXPLAIN [ VERBOSE ] consulta
```

Entradas

VERBOSE

Bandera para mostrar el plan detallado de la consulta.

consulta

Cualquier *consulta*.

Salidas

NOTICE: QUERY PLAN: *plan*

Plan de consulta explícito del backend Postgres.

EXPLAIN

Bandera enviada luego de mostrarse el plan.

Descripción

Este comando muestra el plan de ejecución que el planificador Postgres genera para la consulta dada. El plan de ejecución muestra la manera en que serán escaneadas las tablas referenciadas — ya sea escaneo secuencial plano, escaneo por índice, etc. — y si se referencian varias tablas, los algoritmos de unión que serán utilizados para agrupar las tuplas requeridas para cada tabla de entrada.

La parte más crítica de la presentación es el costo estimado de ejecución de la consulta, que es la suposición del planificador sobre el tiempo que tomará correr la consulta (medido en unidades de captura de páginas de disco). En realidad se muestran dos números: el tiempo inicial que toma devolverse la primer tupla, y el tiempo total para devolver todas las tuplas. Para la mayoría de las consultas lo que importa es el tiempo total, pero en algunos casos como una sub-consulta EXISTS el planificador escogerá el menor tiempo inicial en vez del menor tiempo total (ya que en todo caso el ejecutor se detendrá después de obtener la primer tupla). También, si Ud. limita el número de tuplas a devolver con una cláusula LIMIT, el planificador realiza una interpolación apropiada entre los dos costos finales para estimar cuál de los planes es realmente el menos costoso.

La opción VERBOSE emite la representación interna completa del árbol del plan, en vez de un resumen (y lo envía al archivo log del postmaster también). Usualmente esta opción es únicamente útil para la corrección de errores (debug) de Postgres.

Notas

Existe escasa documentación en Postgres con respecto a la utilización por parte del optimizador de la información de costos. Información general sobre la estimación de costos para la optimización de las consultas puede encontrarse en libros de textos de bases de datos. Refiérase a los capítulos sobre índices y el optimizador genético de consultas de la *Guía del Programador* para mayor información.

Uso

Para mostrar un plan de consulta para una consulta simple sobre una tabla con una única columna de tipo `int4` y 128 filas:

```
EXPLAIN SELECT * FROM foo;
NOTICE: QUERY PLAN:

Seq Scan on foo (cost=0.00..2.28 rows=128 width=4)

EXPLAIN
```

Para la misma tabla con un índice para lograr una condición *equijoin* en la consulta, **EXPLAIN** mostrará un plan distinto:

```
EXPLAIN SELECT * FROM foo WHERE i = 4;
NOTICE: QUERY PLAN:

Index Scan using fi on foo (cost=0.00..0.42 rows=1 width=4)

EXPLAIN
```

Y para terminar, para la misma tabla con un índice para lograr una condición *equijoin* en la consulta, **EXPLAIN** mostrará lo siguiente para una consulta que utilice una función de agregación:

```
EXPLAIN SELECT sum(i) FROM foo WHERE i = 4;
NOTICE: QUERY PLAN:

Aggregate (cost=0.42..0.42 rows=1 width=4)
-> Index Scan using fi on foo (cost=0.00..0.42 rows=1 width=4)
```

Nótese que los números específicos mostrados, y aún la estrategia de consulta seleccionada, pueden variar entre dos versiones de Postgres debido al mejoramiento del planificador.

Compatibilidad

SQL92

No existe una sentencia **EXPLAIN** definida en SQL92.

FETCH

Nombre

FETCH — Selecciona filas usando un cursor

Synopsis

```
FETCH [ selector ] [ count ] { IN | FROM } cursor
FETCH [ RELATIVE ] [ { [ # | ALL | NEXT | PRIOR ] } ] FROM ] cursor
```

Entradas

selector

selector define la dirección de FETCH. Puede ser una de las siguientes:

FORWARD

selecciona la(s) siguiente(s) filas. Es el valor por defecto si se omite *selector*.

BACKWARD

selecciona la(s) fila(s) anterior(es).

RELATIVE

Palabra sin significado (Noise word), para compatibilidad con SQL92.

count

count determina cuántas filas hay que seleccionar. Puede ser uno de los siguientes:

#

Un entero con signo que especifica cuántas filas hay que seleccionar. Dese cuenta de que un entero negativo es equivalente a cambiar el sentido de FORWARD y BACKWARD.

ALL

Devuelve todas las filas restantes.

NEXT

Equivalente a especificar un "count" de **1**.

PRIOR

Equivalente a especificar un "count" de **-1**.

cursor

El nombre de un cursor abierto.

Salidas

FETCH retorna el resultado de la consulta definida por el cursor especificado. Si la consulta falla serán mostrados los siguientes mensajes:

NOTICE: PerformPortalFetch: portal "*cursor*" not found

Si el *cursor* no está previamente declarado. El cursor debe ser declarado dentro de un bloque de operación (transaction block).

NOTICE: FETCH/ABSOLUTE not supported, using RELATIVE

Postgres no soporta el posicionamiento absoluto de los cursores.

ERROR: FETCH/RELATIVE at current position is not supported

SQL92 permite devolver de forma repetida el cursor en su "posición actual" usando la sintaxis

```
FETCH RELATIVE 0 FROM cursor
```

Postgres actualmente no soporta este concepto, de hecho, el valor cero está reservado para indicar que todas las filas deben ser devueltas y es equivalente a especificar la palabra clave ALL. Si se ha usado la palabra clave RELATIVE, Postgres asume que el usuario desea un comportamiento como en SQL92 y devuelve este mensaje de error.

Description

FETCH permite a un usuario devolver filas usando un cursor. El número de filas devueltas está especificado mediante #. Si el número de filas restantes en el cursor es menor a than #, sólo serán seleccionadas las disponibles. Sustituyendo la palabra clave ALL en lugar de un número provocará que sean devueltas todas las filas restantes en el cursor. Las instancias pueden ser seleccionadas en ambas direcciones hacia adelante y hacia atrás (FORWARD y BACKWARD). La dirección por defecto es FORWARD.

Sugerencia: Se permite especificar números negativos en el contador. Un número negativo es equivalente a modificar el sentido de las palabras clave FORWARD y BACKWARD. Por ejemplo, **FORWARD -1** es igual a **BACKWARD 1**.

Notas

Dese cuenta de que las palabras clave FORWARD y BACKWARD son extensiones Postgres. La sintaxis SQL92 también es soportada, especificada en la segunda forma del comando. Véanse más abajo detalles y temas de compatibilidad.

Una vez todas las filas se han seleccionado, todos los demás accesos de fetch no devuelven filas.

Postgres no soporta la característica de actualizar los datos en un cursor, ya que volver a mapear las actualizaciones del cursor en las tablas base no es posible por regla general, como sucede también en las actualizaciones de las vistas (VIEW). Por consiguiente, los usuarios deben explicitar comandos UPDATE para sustituir los datos.

Los cursores sólo sólo se deberían usar dentro de transacciones, ya que los datos que almacenan abarcan múltiples consultas de usuario.

Usar *MOVE* para modificar la posición del cursor. *DECLARE* definirá un cursor. Refiérase a *BEGIN*, *COMMIT*, y a *ROLLBACK* para mayor información acerca de las transacciones.

Uso

Los siguientes ejemplos recorren una tabla usando un cursor. The following examples traverse a table using a cursor.

```
-montar y usar un cursor:
-
BEGIN WORK;
    DECLARE liahona CURSOR
        FOR SELECT * FROM films;

-seleccionar las primeras cinco filas en el cursor liahona:
-
    FETCH FORWARD 5 IN liahona;
```

code	title	did	date_prod	kind	len
BL101	The Third Man	101	1949-12-23	Drama	01:44
BL102	The African Queen	101	1951-08-11	Romantic	01:43
JL201	Une Femme est une Femme	102	1961-03-12	Romantic	01:25
P_301	Vertigo	103	1958-11-14	Action	02:08
P_302	Becket	103	1964-02-03	Drama	02:28

```

-Seleccionar la fila anterior:
-
    FETCH BACKWARD 1 IN liahona;
```

code	title	did	date_prod	kind	len
P_301	Vertigo	103	1958-11-14	Action	02:08

```

- cerrar el cursor y commit work:
-
    CLOSE liahona;
COMMIT WORK;
```

Compatibilidad

SQL92

Nota: El uso no embebido de los cursores es una extensión Postgres. La sintaxis y el uso de los cursores está siendo comparada en contraposición a la forma embebida de los cursores definida en SQL92.

SQL92 permite el posicionamiento absoluto del cursor para FETCH y también la localización de los resultados en variables explícitas.

```
FETCH ABSOLUTE #
      FROM cursor
      INTO :variable [, ...]
```

ABSOLUTE

El cursor debe ser posicionado al número de fila absoluto especificado. Todos los números de filas en Postgres son números relativos, por lo tanto no se soporta esta característica.

:*variable*

Variable(s) objetivo del host.

GRANT

Nombre

GRANT — otorga privilegios de acceso a un usuario, un grupo o a todos los usuarios

Synopsis

```
GRANT privilege [, ...] ON object [, ...]
      TO { PUBLIC | GROUP group | username }
```


Entradas

privilege

Los posibles privilegios son:

SELECT

Acceso a todas las columnas de una tabla / vista específica.

INSERT

Inserta datos en todas las columnas de una tabla específica.

UPDATE

Actualiza todas las columnas de una tabla específica.

DELETE

Elimina filas de una tabla específica.

RULE

Define las reglas de la tabla (vista) (ver sentencia CREATE RULE).

ALL

Otorga todos los privilegios-

object

El nombre de un objeto al que se quiere conceder el acceso. Los posibles objetos son:

- tabla
- vista
- secuencia
- índice

PUBLIC

Una abreviación para representar a todos los usuarios.

GROUP *group*

Un *grupo* al que se otorgan privilegios. En la actual versión, el grupo debe haber sido creado explícitamente como se describe más adelante.

username

El nombre de un usuario al que se quiere conceder privilegios. PUBLIC es una abreviatura para representar a todos los usuarios.

Salidas

CHANGE

Mensaje devuelto se la acción se ha realizado satisfactoriamente.

```
ERROR: ChangeAcl: class "object" not found
```

Mensaje devuelto si el objeto especificado no está disponible o si es imposible dar los privilegios a grupo o usuarios especificado.

Descripción

GRANT permite al creador de un objeto el dar permisos específicos a todos los usuarios (PUBLIC) o a un cierto usuario o grupo. Usuarios distintos al creador pueden no tener permisos de acceso a menos que el creador se los conceda, una vez que el objeto ha sido creado.

Una vez que un usuario tiene privilegios sobre un objeto, tiene posibilidad de ejecutar ese privilegio. No hay necesidad de conceder privilegios al creador de un objeto; el creador obtiene automáticamente TODOS los privilegios, y puede también eliminar el objeto.

Notas

Actualmente, para conceder privilegios en Postgres a solo algunas columnas, he de crear una vista que contenga las columnas deseadas, y conceder privilegios sobre esa vista.

Use **psql \z** para obtener más información sobre los permisos de los objetos existentes:

```
Database      = lusitania
+-----+-----+
|  Relacion    |      Conceder/Eliminar Permisos      |
+-----+-----+
| mytable      | {"=rw","miriam=arwR","group todos=rw"} |
+-----+-----+
Leyenda:
      uname=arwR - se conceden privilegios a un usuario
      group gname=arwR - se conceen privilegios al un GRUPO
              =arwR - se conceden privilegios a PUBLIC

              r - SELECT
              w - UPDATE/DELETE
              a - INSERT
              R - RULE
      arwR - ALL
```

Sugerencia: Actualmente, para crear un GRUPO ha de insertar los datos manualmente en la tabla `pg_group` como sigue:

```
INSERT INTO pg_group VALUES ('todos');
CREATE USER miriam IN GROUP todos;
```

Véase la sentencia **REVOKE** para ver como eliminar los privilegios de acceso.

Uso

Concede privilegios de inserción a todos los usuarios de la tabla 'films':

```
GRANT INSERT ON films TO PUBLIC;
```

Concede todos los privilegios al usuario 'manuel' sobre la vista 'kinds':

```
GRANT ALL ON kinds TO manuel;
```

Compatibilidad

SQL92

La sintaxis de SQL92 para GRANT permite establecer derechos sobre columnas individuales, y permite establecer el privilegio de conceder el mismo privilegio a otros:

```
GRANT privilege [, ...]
    ON object [ ( column [, ...] ) ] [, ...]
    TO { PUBLIC | username [, ...] } [ WITH GRANT OPTION ]
```

Los campos son compatibles con los de la implementación de Postgres, con las siguientes incorporaciones:

privilege

SQL92 permite privilegios adicionales a los mencionados:

SELECT

REFERENCES

Permitido para hacer referencia a alguna o todas las columnas de una tabla/vista específica en limitaciones de integridad.

USAGE

Permitido para usar un dominio, un conjunto de caracteres, cotejo o traducción. Si un objeto especifica algo que no sea una tabla/vista, *privilegio* ha de especificar solo USAGE.

object

[TABLE] *table*

SQL92 permite adicionalmente la palabra clave no funcional TABLE.

CHARACTER SET

Se permite usar el juego de caracteres especificado.

COLLATION

Se permite usar la secuencia de cotejo especificada.

TRANSLATION

Se permite usar la conversión de juego de caracteres especificada.

DOMAIN

Se permite usar el dominio especificado.

WITH GRANT OPTION

Se permite conceder el mismo privilegio a otros.

INSERT**Nombre**

INSERT — Inserta filas nuevas en una tabla

Synopsis

```
INSERT INTO table [ ( column [, ...] ) ]
    { VALUES ( expression [, ...] ) | SELECT query }
```

Entradas

table

El nombre de una tabla existente.

column

El nombre de una columna en *table*.

expression

Una expresión o un valor válidos a asignar en *column*.

query

Una consulta válida. Vea la instrucción SELECT para una mejor descripción de argumentos válidos.

Salidas

```
INSERT oid 1
```

Mensaje devuelto si solo se ha insertado una fila. *oid* es el número OID de la fila insertada.

```
INSERT 0 #
```

Mensaje devuelto si se ha insertado más de una fila. *#* es el número de filas insertadas.

Descripción

INSERT permite la inserción de nuevas filas en una clase o una tabla. Se puede insertar una fila a la vez o varias como el resultado de una consulta. Las columnas en el resultado pueden ser listadas en cualquier orden.

Cada columna que no esté presente en la lista de origen será insertada usando el valor por defecto, que puede ser tanto un valor por defecto declarado **DEFAULT** o bien **NULL**. Postgres rechazará la nueva columna si se inserta un **NULL** en una columna declarada como **NOT NULL**.

Si la expresión para cada columna no es del tipo de datos correcto, se intentará una coerción de tipos automáticamente.

Debe tener privilegios de inserción en la tabla para añadir en ella, así como privilegios de selección en cualquier tabla especificada en una cláusula **WHERE**.

Uso

Inserta una fila en la tabla *films*:

```
INSERT INTO films VALUES
('UA502','Bananas',105,'1971-07-13','Comedy',INTERVAL '82 minute');
```

En este segundo ejemplo la columna *date_prod* se omite y entonces tendrá el valor por defecto de **NULL**:

```
INSERT INTO films (code, title, did, date_prod, kind)
VALUES ('T_601', 'Yojimbo', 106, DATE '1961-06-16', 'Drama');
```

Inserta una fila simple en la tabla *distributors*; note que solo se especifica la columna *name*, de forma que la columna omitida *did* será asignada con su valor por defecto.

```
INSERT INTO distributors (name) VALUES ('British Lion');
```

Inserta varias filas en la tabla *films* desde la tabla *tmp*:

```
INSERT INTO films SELECT * FROM tmp;
```

Inserción en arrays (vea *The PostgreSQL User's Guide* para mayor información sobre los arrays):

```
- Crea un tablero de juego vacío de 3x3 para cruz y raya
- (todos estos queries generan el mismo efecto)
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1, '{{"","",""},{{{"",""}}}');
INSERT INTO tictactoe (game, board[3][3])
    VALUES (2, '{{}}');
INSERT INTO tictactoe (game, board)
    VALUES (3, '{{"",""},{{"",""},{{"",""}}}');
```

Compatibilidad

SQL92

INSERT es totalmente compatible con SQL92. Las posibles limitaciones en las características de la cláusula *query* están documentadas en *SELECT*.

LISTEN

Nombre

LISTEN — Recibir aviso de la notificación de una condición

Synopsis

```
LISTEN nombre
```

Entradas

nombre

Nombre de la condición de notificación.

Salidas

`LISTEN`

Mensaje devuelto cuando se completa exitosamente el registro.

`NOTICE Async_Listen: We are already listening on nombre`

Si este backend ya fue registrado para ser avisado cuando se notifica esa condición.

Descripción

LISTEN registra al backend Postgres para recibir aviso de la notificación de una condición *nombre*.

Cada vez que el comando **NOTIFY** *nombre* es invocado, ya sea por este backend u otro conectado a la misma base de datos, todos los backends que están registrados para ser avisados de la notificación de esa condición, reciben el aviso, y en su momento cada uno de ellos notificará a su aplicación frontend. Véase el tratamiento de **NOTIFY** para mayor información.

Un backend puede anular su registro de recepción de aviso de una condición de notificación dada a través del comando **UNLISTEN**. Asimismo, todos los registros de recepción de avisos se anulan automáticamente cuando finaliza el proceso backend.

El método mediante el cual la aplicación frontend detecta los eventos de notificación depende de la interfaz de programación de aplicaciones Postgres utilizada. Con la librería básica libpq, la aplicación envía **LISTEN** como un comando SQL ordinario, y entonces llama periódicamente a la rutina `PQnotifies` para averiguar si se ha recibido algún evento de notificación. Otras interfaces como libpqctl proporcionan métodos de alto nivel para el manejo de eventos de notificación; de hecho, con libpqctl el programador de aplicaciones no debe enviar **LISTEN** o **UNLISTEN** directamente. Véase la documentación de la librería utilizada para mayores detalles.

NOTIFY contiene un tratamiento más extenso de la utilización de **LISTEN** y **NOTIFY**.

Notas

nombre puede ser cualquier cadena válida como nombre; no es necesario que sea igual al nombre de una tabla existente. Si *nombre* se encierra entre comillas, ni siquiera es necesario que sea un nombre válido, sino cualquier cadena de hasta 31 caracteres de largo.

En algunas versiones previas de Postgres, *nombre* debía ser encerrado entre comillas cuando no se correspondía con el nombre de una tabla existente, aunque fuera sintácticamente correcto como nombre. Actualmente no es requerido.

Uso

Configura y ejecuta una secuencia recepción de aviso/notificación desde `psql`:

```
LISTEN virtual;
```

```

NOTIFY virtual;

ASYNC NOTIFY of 'virtual' from backend pid '11239' received

```

Compatibilidad

SQL92

El comando **LISTEN** no existe en SQL92.

LOAD

Nombre

LOAD — Carga dinamicamente un fichero objeto

Synopsis

```
LOAD 'nombrefichero'
```

Parametros de Entrada

nombrefichero

Nombre del fichero para cargar dinamicamente.

Outputs

LOAD

Mensaje devuelto en caso de suceso en la operacion.

```
ERROR: LOAD: could not open file 'nombrefichero'
```

Mensaje devuelto si el fichero especificado no es encontrado. El fichero debe ser visible *alPostgres backend*, y debe ser enviado con su apropiado camino completo (path), para no obtener este tipo de error.

Descripción

Carga un fichero objeto (o ".o") en el espacio de direccionamiento Postgres. Una vez que el fichero es cargado en memoria, todas las funciones de ese fichero pueden ser llamadas. Esta función es usada para soporte de tipos y funciones definidas por el usuario.

Si un fichero no es cargado usando **LOAD**, el fichero será cargado automáticamente la primera vez que una función sea llamada por el Postgres. **LOAD** Puede ser usado para recargar un fichero objeto si este ha sido editado y recompilado. Por el momento, únicamente son soportados ficheros objeto que son creados con el lenguaje C.

Notas

Funciones que se encuentran en ficheros objeto no deberían llamar a otras funciones en otros ficheros objeto que fueron cargados por medio del comando **LOAD**. Por ejemplo, todas las funciones en el fichero A pueden llamar a otras funciones que se encuentran en las librerías standard o math, o en las del propio Postgres. Estas no deberían llamar funciones definidas en otro fichero cargado B. Esto es así porque si B es recargado, el cargador del Postgres no está preparado para realocar las llamadas desde las funciones en A en el nuevo espacio de direccionamiento de B. Si B no es recargado, entonces no habrá problemas.

Ficheros objeto deben ser compilados para contener código sin dependencia de posición. Por ejemplo, en estaciones DEC, debe usar `/bin/cc` con la opción `-G 0` cuando compila ficheros objeto para ser cargados.

Si está pensando en portar Postgres a una nueva plataforma, **LOAD** debe trabajar de forma tal que soporte ADTs.

Uso

Carga el fichero `/usr/postgres/demo/circle.o`:

```
LOAD '/usr/postgres/demo/circle.o'
```

Compatibilidad

SQL92

No existe el comando **LOAD** en SQL92.

LOCK

Nombre

LOCK — Explícitamente bloquea una tabla dentro de una transacción

Synopsis

```
LOCK [ TABLE ] name
LOCK [ TABLE ] name IN [ ROW | ACCESS ] { SHARE | EXCLUSIVE } MODE
LOCK [ TABLE ] name IN SHARE ROW EXCLUSIVE MODE
```

Entradas

name

El nombre de una tabla existente para bloquear.

ACCESS SHARE MODE

Nota: A este modo de bloqueo se accede automáticamente sobre tablas que estan siendo consultadas. Postgres libera automáticamente los bloqueos accedidos ACCESS SHARE despues de que se haya hecho la sentencia.

Este es el modo de bloqueo menos restrictivo el cual entra en conflicto sólo con el modo ACCESS EXCLUSIVE . Se pretende proteger una tabla que está siendo consultada de sentencias concurrentes **ALTER TABLE**, **DROP TABLE** y **VACUUM** sobre la misma tabla.

ROW SHARE MODE

Nota: Se accede automáticamente por cualquier declaración **SELECT FOR UPDATE**.

Conflictos con los modos de bloqueo EXCLUSIVE y ACCESS EXCLUSIVE.

ROW EXCLUSIVE MODE

Nota: Se accede automáticamente por cualquier sentencia **UPDATE**, **DELETE**, **INSERT**.

Conflictos con los modos SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE ACCESS EXCLUSIVE. Generalmente significa que una transacción actualiza o inserta algunas tuplas en una tabla.

SHARE MODE

Nota: Se accede automáticamente por cualquier sentencia **CREATE INDEX**

Conflictos con los modos ROW EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE y ACCESS EXCLUSIVE . Este modo protege una tabla contra actualizaciones concurrentes.

SHARE ROW EXCLUSIVE MODE

Conflictos con los modos ROW EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE y ACCESS EXCLUSIVE. Este modo es más restrictivo que el modo SHARE debido a que sólo puede soportar este bloqueo una transacción por vez .

EXCLUSIVE MODE

Conflictos con los modos ROW SHARE, ROW EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE y ACCESS EXCLUSIVE modes. Este modo es aún más restrictivo que éste de SHARE ROW EXCLUSIVE; bloquea todas las consultas concurrentes SELECT FOR UPDATE .

ACCESS EXCLUSIVE MODE

Nota: Se accede automáticamente por las sentencias **ALTER TABLE, DROP TABLE, VACUUM** .

Este es el modo de bloqueo más restrictivo y es incompatible con todos los demás modos de bloqueo y protege una tabla bloqueada de cualquier otra operación concurrente.

Nota: Este modo de bloqueo se accede también por un **LOCK TABLE** sin cualificar. (i.e. el comando sin una opción de bloqueo explícita).

Salidas

```
LOCK TABLE
```

El bloqueo se activó con éxito.

```
ERROR name: La tabla no existe.
```

Mensaje devuelto si el *nombre* no existe.

Description

Postgres siempre usa el modo de bloqueo menos restrictivo cuando le es posible. **LOCK TABLE** toma medidas para cuando se pueda necesitar un modo de bloqueo mas restrictivo.

Por ejemplo, una aplicación ejecuta una transacción en el nivel de aislamiento READ COMMITTED y necesita asegurar la existencia de datos en una tabla para la duración de la transacción. Para ello tú podrías usar el modo de bloqueo SHARE sobre la tabla antes de la consulta. Esto protegerá los datos de cambios concurrentes y proporcionará cualquier otra operación de escritura sobre la tabla con datos en su verdadero estado actual, porque el modo de bloqueo SHARE es incompatible con cualquier ROW EXCLUSIVE accedido por los que escriben, y **LOCK TABLE "tabla" en sentencia IN SHARE MODE** esperará hasta que se produzca o se "baje" cualquier operación de escritura concurrente.

Nota: Para leer datos en su verdadero estado actual cuando ejecutas una transacción en el nivel de aislamiento SERIALIZABLE tienes que ejecutar una declaración LOCK TABLE antes de la ejecución de cualquier sentencia DML, cuando la transacción define qué cambios concurrentes serán visibles por ellos mismos.

Además de los requerimientos precedentes, si una transacción va a cambiar datos en una tabla entonces se debería acceder al modo SHARE ROW EXCLUSIVE para evitar condiciones de punto muerto cuando dos transacciones coincidentes intentan bloquear la tabla en modo SHARE y entonces intentan cambiar datos en esta tabla, ambas (implícitamente) accediendo al modo de bloqueo ROW EXCLUSIVE que es incompatible con el bloqueo SHARE .

Para continuar con los puntos muertos (cuando dos transacciones se esperan la una a la otra) tema tratado arriba, deberías seguir dos reglas generales para evitar condiciones de punto muerto :

- Las transacciones tienen que acceder a bloqueos de los mismos objetos en el mismo orden.

Por ejemplo, si una aplicación actualiza la fila R1 y después actualiza la fila R2 (en la misma transacción) entonces la segunda aplicación no debería actualizar la fila R2 si ello va a actualizar la fila R1 más tarde (en una transacción simple). En cambio, debería actualizar la fila R1 y R2 en el mismo orden como en la primera aplicación.

- Las transacciones deberían procurarse dos modos de bloqueo conflictivos sólo si uno de ellos es auto-conflictivo (i.e. podría ser soportado por sólo una transacción cada vez). Si estan involucrados modos de bloqueo múltiples, entonces las transacciones deberían siempre acceder primero al modo más restrictivo.

Un ejemplo para esta regla se dió antes cuando se discutió el uso del modo SHARE ROW EXCLUSIVE mejor que el modo SHARE.

Nota: Postgres no detecta puntos muertos "bajará" una transacción a la espera para resolver el punto muerto.

Notas

LOCK es una extension del lenguaje Postgres.

Excepto para los modos de bloqueo ACCESS SHARE/EXCLUSIVE , todos los demás modos de bloqueo de Postgres y las sentencias **LOCK TABLE** son compatibles con aquellos presentes en Oracle.

LOCK funciona sólo dentro de transacciones.

Uso

Illustrate a SHARE lock on a primary key table when going to perform inserts into a foreign key table:

```
BEGIN WORK;
LOCK TABLE películas IN SHARE MODE;
SELECT id FROM películas
    WHERE name = 'Star Wars: Episodio I - La amenaza fantasma';
- Haz ROLLBACK si el registro no fue devuelto
INSERT INTO comentarios_usuario_películas VALUES
    (_id_, 'GUAY! Llevaba tanto tiempo esperándola!');
COMMIT WORK;
```

Toma un bloqueo SHARE ROW EXCLUSIVE clave de tabla primaria cuando vayas a hacer una operación de borrado:

```
BEGIN WORK;
LOCK TABLE películas IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM comentarios_usuario_películas WHERE id IN
    (SELECT id FROM películas WHERE clasificación < 5);
DELETE FROM películas WHERE clasificación < 5;
COMMIT WORK;
```

Compatibilidad

SQL92

No hay **LOCK TABLE** en SQL92, que usa en cambio **SET TRANSACTION** para especificar niveles de concurrencia en transacciones. Nosotros también la tenemos; ver *SET* para más detalles.

MOVE

Nombre

MOVE — Mueve la posición del cursor

Synopsis

```
MOVE [ selector ] [ count ]
    { IN | FROM } cursor
    FETCH [ RELATIVE ] [ { [ # | ALL | NEXT | PRIOR ] } ] FROM ] cur-
sor
```

Descripción

MOVE permite al usuario mover la posición del cursor un número específico de filas. **MOVE** funciona como el comando **FETCH**, pero sólo posiciona el cursor y no devuelve filas.

Ir a *FETCH* para detalles de sintaxis y uso.

Notes

MOVE es una extensión del language Postgres.

Ir a *FETCH* para una descripción de los argumentos válidos. Ir a *DECLARE* par definir un cursor. Ir a *BEGIN*, *COMMIT*, y *ROLLBACK* para más información acerca de transacciones.

Usage

Configurar y usar un cursor:

```
BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;
-Saltarse las 5 primeras filas:
MOVE FORWARD 5 IN liahona;
MOVE
-Fetch la 6ª fila en el cursor liahona:
```

```

FETCH 1 IN liahona;
FETCH

      code |title |did| date_prod|kind      |len
      ----+-----+-----+-----+-----+-----
      P_303|48 Hrs|103|1982-10-22|Action    | 01:37
      (1 row)
- cierra el cursor liahona and commit work:
CLOSE liahona;
COMMIT WORK;

```

Compatibility

SQL92

No hay sentencia SQL92 **MOVE**. En cambio, SQL92 permite one to **FETCH** filas de una posición absoluta del cursor, moviendo implícitamente el cursor a una posición correcta.

NOTIFY

Nombre

NOTIFY — Señala todos los "fronends" y "backends" a la escucha de una condición notify.

Synopsis

```
NOTIFY name
```

Entradas

notifyname

Notifica la condición a ser señalada.

Salidas

```
NOTIFY
```

Acuse de recibo de que el comando notify ha sido ejecutado.

Eventos Notify

Los eventos son repartidos a los "frontends" que están a la escucha; el cómo y si cada aplicación "frontend" reacciona depende de su programación.

Descripción

El comando **NOTIFY** envía un evento notify a cada aplicación frontend que previamente ha ejecutado **LISTEN *notifyname*** para la condición notify específica en la base de datos en curso.

La información pasada al "frontend" para un evento notify incluye el nombre de la condición notify y el PID de la notificación del proceso "backend". Es asunto del diseñador de la base de datos el definir los nombres de las condiciones que serán usadas en una base de datos dada y que significa cada una.

Comunmente, el nombre de una condición notify es el mismo que el de alguna tabla en la base de datos, y el evento notify esencialmente significa "He cambiado ésta tabla, echale un vistazo para ver los cambios". Pero dicha asociación no es obligada por lo comandos **NOTIFY** y **LISTEN**. Por ejemplo, un diseñador de bases de datos podría usar varios nombres de condición diferentes para señalar diferentes tipos de cambios en una misma tabla.

NOTIFY provee un modo simple de señalar o un mecanismo de comunicación entre procesos (IPC interprocess communication) para el conjunto de procesos que acceden a la misma base de datos Postgres. Se pueden construir mecanismos de más alto nivel usando tablas en la base de datos para pasar datos adicionales (más allá de un mero nombre de condición) desde el notificador al o a los que estén a la escucha.

Cuando se usa **NOTIFY** para señalar la ocurrencia de cambios en una tabla en particular, una técnica útil de programación es poner **NOTIFY** en una norma que es disparada por actualizaciones de la tabla. De esta manera, la notificación es automática cuando la tabla cambia, y el programador de la aplicación no puede olvidarse de ello de forma accidental.

NOTIFY interactúa con transacciones SQL de una manera importante. Primero, si se ejecuta un **NOTIFY** dentro de una transacción, los eventos notify no son repartidos hasta y a menos que la transacción se haya hecho. Esto es adecuado, dado que si una transacción se aborta nos gustaría que todos los comandos dentro de ella no hubieran tenido efecto, incluyendo **NOTIFY**. Pero puede ser desconcertante si uno está esperando que los eventos notify se repartan inmediatamente. Segundo, si un "backend" a la escucha recibe una señal notify mientras está en una transacción, el evento notify no se repartirá al "frontend" conectado hasta justo después de que la transacción se haya completado (tanto si se ejecuta como si se aborta). De nuevo, la razón es que si un notify fuera repartido dentro de una transacción que después fue abortado, sería deseable que la notificación se deshiciera de alguna manera — pero el "backend" no puede echar marcha atrás un notify una vez que ha sido enviado al "frontend". Por tanto los eventos notify son sólo repartidos entre transacciones. El resultado de esto es que las aplicaciones que usan **NOTIFY** para señalar en tiempo real deberían tratar de mantener cortas sus transacciones.

NOTIFY se comporta como las señales Unix en un aspecto importante: si una misma condición es señalada varias veces en una sucesión rápida, los receptores pueden que sólo recibieran un evento notify para varias ejecuciones de **NOTIFY**. Por ello es mala idea depender del número de notificaciones recibidas. En cambio, usaremos **NOTIFY** para "despertar" a las aplicaciones que necesitan prestar atención a algo,

y usaremos un objeto de base de datos (tal como una secuencia) para mantener un registro de lo que ha ocurrido o cuantas veces ha ocurrido.

Es usual para un "frontend" que envía **NOTIFY** estar él mismo a la escucha del mismo nombre notify. En ese caso recibirá un evento notify , justo igual que los otros "frontends" a la escucha. Dependiendo de la lógica de la aplicación, esto podría acarrear un trabajo inútil — por ejemplo, releendo una tabla de una base de datos para encontrar la misma actualización que ése mismo frontend acababa de escribir. En Postgres 6.4 y posteriores , es posible evitar dicho trabajo extra notificando si el PID del proceso de notificación del "backend" (suministrado en el mensaje del evento notify) es el mismo que el PID del backend de uno mismo (valga la redundancia) (disponible en libpq). Cuando son el mismo, el evento notificación es la recuperación del propio trabajo de uno mismo, y puede ser ignorado. (A pesar de lo que se dijo en el párrafo precedente, esto es una técnica segura. Postgres mantiene las auto-notificaciones separadas de las notificaciones que llegan de otros "backends", de manera que no puedes perder una notificación de fuera por ignorar tus propias notificaciones. (Si alguien entiende ésto que me lo explique))

Notas

name puede ser una cadena válica com un nombre; no es necesaria una relación con el nombre de la tabla en sí. Si *name* se encierra entre dobles comillas, ni siquiera necesita un nombre sintácticamente válido, sino que puede ser cualquier cadena de hasta 31 caracteres de longitud.

En algunas versiones previas de Postgres, *name* tenía que encerrarse entre comillas dobles cuando no había relación con ningún nombre de tabla existente, incluso si sintácticamente era válido como nombre. Esto ya no es necesario.

En versiones Postgres anteriores a la 6.4, el PID de backend repartido en un mensaje notify era siempre el PID del backend del frontend de uno mismo. Por eso no se podía distinguir las notificaciones de uno mismo de las notificaciones de otros clientes en aquellas versiones.

Uso

Configura y ejecuta una secuencia listen(escucha)/notify(notificación) desde `psql`:

```
LISTEN virtual;
NOTIFY virtual;
ASYNC NOTIFY de 'virtual' desde el pide de backend '11239' recibido
```

Compatibilidad

SQL92

No hay sentencia **NOTIFY** en SQL92.

RESET

Nombre

RESET — Restaura los parámetros en tiempo de ejecución a sus valores por defecto para la sesión actual.

Synopsis

```
RESET variable
```

Entradas

variable

Refiérase a *SET* para mayor información sobre variables disponibles.

Salidas

```
RESET VARIABLE
```

Mensaje devuelto si la *variable* pudo ser restaurada exitosamente a su valor por defecto.

Descripción

RESET restaura variables a sus valores por defecto. Refiérase a *SET* para mayores detalles sobre valores permitidos y por defecto. **RESET** es una forma alternativa para

```
SET variable = DEFAULT
```

Notas

RESET es una extensión del lenguaje de Postgres

Utilice *SET* and *SHOW* para manipular el valor de las variables.

Uso

Establecer `DateStyle` (estilo de fecha) a su valor por defecto:

```
RESET DateStyle;
```

Establecer `Geqo` a su valor por defecto:

```
RESET GEQO;
```

Compatibilidad

SQL92

No existe **RESET** en SQL92.

REVOKE

Nombre

REVOKE — Revoca el privilegio de acceso a un usuario, a un grupo o a todos los usuarios.

Synopsis

```
REVOKE privilegio [, ...]
      ON objeto [, ...]
      FROM { PUBLIC | GROUP ER">gBLE> | nombre_usuario }
```

Entradas

privilegio

Los posibles privilegios son:

SELECT

Privilegio para acceder a todas las columnas de una tabla o vista específica.

INSERT

Privilegio de insertar datos en todas las columnas de una tabla específica.

UPDATE

Privilegio para actualizar todas las columnas de tabla.

DELETE

Privilegio para borrar filas de una tabla específica.

RULE

Privilegio para definir reglas en una tabla o vista. (Veáse *CREATE RULE*).

ALL

Rescinde todos los privilegios.

objeto

El nombre de un objeto sobre el que revocar el acceso. Los posibles objetos son:

- tablea
- vista
- secuencia
- índice

grupo

El nombre de un grupo al cual se revocan privilegios.

nombre_usuario

El nombre de un usuario al cual se revocan privilegios. Utilice la palabra clave **PUBLIC** para especificar todos los usuarios.

PUBLIC

Rescinde el/los privilegio(s) especificado(s) a todos los usuarios.

Salidas**CHANGE**

Mensaje devuelto si ha tenido éxito.

ERROR

Mensaje que se devuelve si el objeto no está disponible o si es imposible revocar privilegios al grupo o a los usuarios.

Descripción

REVOKE permite al creador de una objeto revocar permisos asignados anteriormente a todos los usuarios (mediante **PUBLIC**) o a un usuario o a un grupo.

Notas

Consulte el comando `psql \z` para obtener más información sobre permisos en objetos existentes:

```
Database      = lusitania
+-----+-----+
| Relation    | Grant/Revoke Permissions |
+-----+-----+
| mytable     | {"=rw","miriam=arwR","group todos=rw"} |
+-----+-----+
Legend:
  uname=arwR - privileges granted to a user
  group gname=arwR - privileges granted to a GROUP
  =arwR - privileges granted to PUBLIC

  r - SELECT
  w - UPDATE/DELETE
  a - INSERT
  R - RULE
  arwR - ALL
```

Sugerencia: Actualmente, para crear un grupo debe insertar los datos manualmente en la tabla `pg_group` de este modo:

```
INSERT INTO pg_group VALUES ('todos');
CREATE USER miriam IN GROUP todos;
```

Utilización

Revoca el privilegio de inserción a todos los usuarios en la tabla `films`:

```
REVOKE INSERT ON films FROM PUBLIC;
```

Revoca todos los privilegios al usuario `manuel` en la vista `kinds`:

```
REVOKE ALL ON kinds FROM manuel;
```

Compatibilidad

SQL92

La sintaxis de SQL92 para el comando **REVOKE** tiene capacidades adicionales para rescindir privilegios, incluso aquellos en columnas individuales en tablas:

```
REVOKE { SELECT | DELETE | USAGE | ALL PRIVILEGES } [, ...]
      ON objeto
      FROM { PUBLIC | nombre_usuario [, ...] } { RESTRICT | CASCADE }
REVOKE { INSERT | UPDATE | REFERENCES } [, ...] [ ( columna [, ...] ) ]
      ON objeto
      FROM { PUBLIC | nombre_usuario [, ...] } { RESTRICT | CASCADE }
```

Vea *GRANT* para más detalles en campos individuales.

```
REVOKE GRANT OPTION FOR privilegio [, ...]
      ON objeto
      FROM { PUBLIC | nombre_usuario [, ...] } { RESTRICT | CASCADE }
```

Rescinde a un usuario la autoridad para garantizar el privilegio especificado a otros usuarios. Véase *GRANT* para los detalles en campos individuales.

Los objetos posibles son:

```
[ TABLE ] tabla/vista
CHARACTER SET conjunto_caracteres
COLLATION colección
TRANSLATION traslación
DOMAIN dominio
```

Si user1 da un privilegio con la opción GRANT a user2 y user2 se lo da a user3, entonces user1 puede revocar este privilegio en cascada usando la palabra clave CASCADE.

Si user1 da un privilegio con GRANT a user2 y user2 se lo da a user3, entonces si user1 intenta revocar este privilegio, fallará si ha especificado la palabra clave RESTRICT.

ROLLBACK

Nombre

ROLLBACK — Interrumpe la transacción en curso

Synopsis

`ROLLBACK [WORK | TRANSACTION]`

Entrada.

Ninguna.

Salida.

`ABORT`

Mensaje devuelto si la operación es exitosa.

`NOTICE: ROLLBACK: no transaction in progress`

Si no hay transacciones en progreso actualmente.

Descripción

ROLLBACK deshace la transacción actual y provoca que todas las modificaciones originadas por la misma sean descartadas.

Notas

Utilice *COMMIT* para terminar una transacción de forma exitosa. *ABORT* es un sinónimo de **ROLLBACK**.

Usage

Para cancelar todos los cambios:

`ROLLBACK WORK;`

Compatibilidad

SQL92

SQL92 sólo especifica las dos formas siguientes: `ROLLBACK` y `ROLLBACK WORK`. De cualquier otra forma, la compatibilidad es completa.

SELECT

Nombre

SELECT — Recupera registros desde una tabla o vista.

Synopsis

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
      expression [ AS name ] [, ...]
      [ INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table ]
      [ FROM table [ alias ] [, ...] ]
      [ WHERE condition ]
      [ GROUP BY column [, ...] ]
      [ HAVING condition [, ...] ]
      [ { UNION [ ALL ] | INTERSECT | EXCEPT } select ]
      [ ORDER BY column [ ASC | DESC | USING operator ] [, ...] ]
      [ FOR UPDATE [ OF class_name [, ...] ] ]
LIMIT { count | ALL } [ { OFFSET | , } start ]
```

Inputs

expression

El nombre de una columna de la tabla o una expresión.

name

Especifica otro nombre para una columna o una expresión que utilice la cláusula AS. Este nombre se utiliza principalmente como etiqueta para la columna de salida. El nombre no puede ser utilizado en las cláusulas WHERE, GROUP BY o HAVING. Sin embargo, puede ser referenciado en cláusulas ORDER BY.

TEMPORARY

TEMP

La tabla se crea solamente para esta sesión, y es automáticamente descartada al finalizar la misma.

new_table

Si se utiliza la cláusula INTO TABLE, el resultado de la consulta se almacenará en otra tabla con el nombre indicado. La tabla objetivo (*new_table*) será creada automáticamente y no deberá existir previamente a la utilización de este comando. Consulte el comando **SELECT INTO** para más información.

Nota: La declaración **CREATE TABLE AS** también creará una nueva tabla a partir de la consulta.

table

El nombre de una tabla existente a la que se refiere la cláusula FROM.

alias

Un nombre alternativo para la tabla precedente *table*. Se utiliza para abreviar o eliminar ambigüedades en uniones dentro de una misma tabla.

condition

Una expresión booleana que da como resultado verdadero o falso (true or false). Consulte la cláusula WHERE.

column

El nombre de una columna de la tabla.

select

Una declaración de selección (select) exceptuando la cláusula ORDER BY.

Outputs

Registros

El conjunto completo de registros (filas) que resultan de la especificación de la consulta.

count

La cantidad de registros (filas) devueltos por la consulta.

Descripción

SELECT devuelve registros de una o más tablas. Los candidatos a ser seleccionados son aquellos registros que cumplen la condición especificada con WHERE; si se omite WHERE, se retornan todos los registros. (Consulte *Cláusula WHERE*.)

DISTINCT elimina registros duplicados del resultado. **ALL** (predeterminado) devolverá todos los registros, que cumplan con la consulta, incluyendo los duplicados.

DISTINCT ON elimina los registros que cumplen con todas las expresiones especificadas, manteniendo solamente el primer registro de cada conjunto de duplicados. Note que no se puede predecir cuál será "el primer registro" a menos que se utilice **ORDER BY** para asegurar que el registro eseado es el que efectivamente aparece primero. Por ejemplo:

```
SELECT DISTINCT ON (location) location, time, report
FROM weatherReports
ORDER BY location, time DESC;
```

recuperea el reporte de tiempo (weather report) más reciente para cada locación (location). Pero si no se hubiera utilizado ORDER BY para forzar el orden descendente de los valores de fecha para cada locación, se hubiesen recuperado reportes de una fecha impredecible para cada locación.

La cláusula GROUP BY permite al usuario dividir una tabla conceptualmente en grupos. (Consulte *Cláusula GROUP BY*.)

La cláusula HAVING especifica una tabla con grupos derivada de la eliminación de grupos del resultado de la cláusula previamente especificada. (Consulte *Cláusula HAVING*.)

La cláusula ORDER BY permite al usuario especificar si quiere los registros ordenados de manera ascendente o descendente utilizando los operadores de modo ASC y DESC. (Consulte *Cláusula ORDER BY*.)

El operador UNION permite que el resultado sea una colección de registros devueltos por las consultas involucradas. (Consulte *Cláusula UNION*.)

El operador INTERSECT le da los registros comunes a ambas consultas. (Consulte *Cláusula INTERSECT*.)

El operador EXCEPT le da los registros devueltos por la primera consulta que no se encuentran en la segunda consulta. (Consulte *Cláusula EXCEPT*.)

La cláusula FOR UPDATE permite a SELECT realizar un bloqueo exclusivo de los registros seleccionados.

La cláusula LIMIT permite devolver al usuario un subconjunto de los registros producidos por la consulta. (Consulte *Cláusula LIMIT*.)

Usted debe tener permiso de realizar SELECT sobre una tabla para poder leer sus valores. (Consulte las declaraciones **GRANT/REVOKE**).

Cláusula WHERE

La condición opcional WHERE tiene la forma general:

`WHERE boolean_expr`

boolean_expr puede consistir de cualquier expresión cuyo resultado sea un valor booleano. En muchos casos, esta expresión será:

`expr cond_op expr`

o

`log_op expr`

donde *cond_op* puede ser uno de: =, <, <=, >, >= or <>, un operador condicional como ALL, ANY, IN, LIKE o operador definido localmente, y *log_op* puede ser uno de: AND, OR, NOT. La comparación devuelve TRUE (verdadero) o FALSE (falso) y todas las instancias serán descartadas si la expresión resulta falsa.

Cláusula GROUP BY

GROUP BY especifica una tabla con grupos derivada de la aplicación de esta cláusula:

`GROUP BY column [, ...]`

GROUP BY condensará en una sola fila todos aquellos registros que compartan los mismos valores para las columnas agrupadas. Las funciones de agregación, si las hubiera, son computadas a través de todas las filas que conforman cada grupo, produciendo un valor separado por cada uno de los grupos (mientras que sin GROUP BY, una función de agregación produce un solo valor computado a través de todas las filas seleccionadas). Cuando GROUP BY está presente, no es válido hacer referencia a columnas no agrupadas excepto dentro de funciones de agregación, ya que habría más de un posible valor de retorno para una columna no agrupada.

Cláusula HAVING

La condición opcional HAVING tiene la forma general:

```
HAVING cond_expr
```

donde *cond_expr* cumple las mismas condiciones que las especificadas para WHERE.

HAVING especifica una tabla con grupos derivada de la eliminación de grupos, del resultado de la cláusula previamente especificada, que no cumplen con *cond_expr*.

Cada columna referenciada en *cond_expr* debe referirse precisamente (sin ambigüedades) a una columna de grupo, a menos que la referencia aparezca dentro de una función de agregación.

Cláusula ORDER BY

```
ORDER BY column [ ASC | DESC ] [, ...]
```

column puede ser tanto el nombre de una columna como un número ordinal.

Los números ordinales hacen referencia a la posición (de izquierda a derecha) de la columna. Esta característica hace posible definir un orden basado en una columna que no tiene un nombre adecuado. Esto nunca es absolutamente necesario ya que siempre es posible asignar un nombre a una columna calculada utilizando la cláusula AS, por ej.:

```
SELECT title, date_prod + 1 AS newlen FROM films ORDER BY newlen;
```

A partir de la versión 6.4 de PostgreSQL, es también posible ordenar, con ORDER BY, según expresiones arbitrarias, incluyendo campos que no aparecen en el resultado de SELECT. Por lo tanto, la siguiente declaración es legal:

```
SELECT name FROM distributors ORDER BY code;
```

Opcionalmente una puede agregar la palabra clave DESC (descendente) o ASC (ascendente) luego del nombre de cada columna en la cláusula ORDER BY. Si no se

especifica, se asume ASC de forma predeterminada. Alternativamente, puede indicarse un nombre de operador de orden específico. ASC es equivalente a USING '<' y DESC es equivalente a USING '>'.

Cláusula UNION

```
table_query UNION [ ALL ] table_query
    [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

donde *table_query* especifica cualquier declaración SELECT sin la cláusula ORDER BY.

El operador UNION permite que el resultado sea una colección de registros devueltos por las consultas involucradas. Los dos SELECTs que representan los dos operandos directos de la UNION deben producir el mismo número de columnas, y las columnas correspondientes deben ser de tipos de datos compatibles.

De forma predeterminada, el resultado de UNION no contiene registros duplicados a menos que se especifique la cláusula ALL.

Si se utilizan varios operadores UNION en la misma declaración SELECT se evalúan de izquierda a derecha. Note que la palabra clave ALL no es global, siendo aplicada solamente al par de tablas de resultado actual.

Cláusula INTERSECT

```
table_query INTERSECT table_query
    [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

donde *table_query* especifica cualquier expresión SELECT sin la cláusula ORDER BY.

El operador INTERSECT le da los registros comunes a ambas consultas. Los dos SELECTs que representan los operandos directos de la intersección deben producir el mismo número de columnas, y las columnas correspondientes deben ser de tipos de datos compatibles.

Si se utilizan varios operadores INTERSECT en la misma declaración SELECT se evalúan de izquierda a derecha, a menos que se utilicen paréntesis para modificar esto.

Cláusula EXCEPT

```
table_query EXCEPT table_query
    [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

donde *table_query* especifica cualquier expresión SELECT sin la cláusula ORDER BY.

El operador EXCEPT le da los registros devueltos por la primera consulta pero no por la segunda. Los dos SELECTs que representan los operandos directos de la intersec-

ción deben producir el mismo número de columnas, y las columnas correspondientes deben ser de tipos de datos compatibles.

Si se utilizan varios operadores INTERSECT en la misma declaración SELECT se evalúan de izquierda a derecha, a menos que se utilicen paréntesis para modificar esto.

Cláusula LIMIT

```
LIMIT { count | ALL } [ { OFFSET | , } start ]
OFFSET start
```

donde *count* especifica el máximo número de registros a devolver y *start* especifica el número de registros a saltar antes de empezar a devolver registros.

LIMIT le permite recuperar sólo una porción de los registros que se generan por el resto de la consulta. Si se especifica un número límite, no se devolverán más registros que esa cantidad. Si se da un valor de desplazamiento, esa cantidad de registros será saltada antes de comenzar a devolver registros.

Cuando se utiliza LIMIT es una buena idea utilizar la cláusula ORDER BY para colocar los registros del resultado en un orden único. De otra forma obtendrá un subconjunto impredecible de los registros de la consulta — tal vez esté buscando los registros del décimo al vigésimo, ¿pero del décimo al vigésimo en qué orden? Usted no conoce el orden a menos que utilice ORDER BY.

Ya en Postgres 7.0, el optimizador de consultas toma en cuenta a LIMIT cuando genera un plan de consulta, así que es muy factible que usted obtenga diferentes planes (abarcando diferentes criterios de ordenamiento de registros) dependiendo de los valores dados a LIMIT y OFFSET. Por lo tanto, utilizar diferentes valores para LIMIT/OFFSET para seleccionar diferentes subconjuntos del resultado de una consulta, *provocará resultados inconsistentes* a menos que usted se asegure un resultado predecible ordenando con ORDER BY. Esto no es un bug; es una consecuencia inherente al hecho de que SQL no establece ningún compromiso de entregar los resultados de una consulta en un orden en particular a menos que se utilice ORDER BY para especificar un criterio de orden explícitamente.

Uso

Para unir la tabla films con la tabla distributors:

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
FROM distributors d, films f
WHERE f.did = d.did
```

title	did	name	date_prod	kind
The Third Man	101	British Lion	1949-12-23	Drama
The African Queen	101	British Lion	1951-08-11	Romantic
Une Femme est une Femme	102	Jean Luc Godard	1961-03-12	Romantic
Vertigo	103	Paramount	1958-11-14	Action
Becket	103	Paramount	1964-02-03	Drama
48 Hrs	103	Paramount	1982-10-22	Action
War and Peace	104	Mosfilm	1967-02-12	Drama
West Side Story	105	United Artists	1961-01-03	Musical
Bananas	105	United Artists	1971-07-13	Comedy

Yojimbo	106	Toho	1961-06-16	Drama
There's a Girl in my Soup	107	Columbia	1970-06-11	Comedy
Taxi Driver	107	Columbia	1975-05-15	Action
Absence of Malice	107	Columbia	1981-11-15	Action
Storia di una donna	108	Westward	1970-08-15	Romantic
The King and I	109	20th Century Fox	1956-08-11	Musical
Das Boot	110	Bavaria Atelier	1981-11-11	Drama
Bed Knobs and Broomsticks	111	Walt Disney		Musical

Para sumar la columna `len` (duración) de todos los filmes y agrupar los resultados según la columna `kind` (tipo):

```
SELECT kind, SUM(len) AS total FROM films GROUP BY kind;
```

kind	total
-----+-----	
Action	07:34
Comedy	02:58
Drama	14:28
Musical	06:42
Romantic	04:38

Para sumar la columna `len` de todos los filmes, agrupar los resultados según la columna `kind` y mostrar los totales de esos grupos que sean menores a 5 horas:

```
SELECT kind, SUM(len) AS total
FROM films
GROUP BY kind
HAVING SUM(len) < INTERVAL '5 hour';
```

kind	total
-----+-----	
Comedy	02:58
Romantic	04:38

Los siguientes dos ejemplos muestran maneras idénticas de ordenar los resultados individuales de acuerdo con los contenidos de la segunda columna (`name`):

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;
```

did	name
---+-----	
109	20th Century Fox
110	Bavaria Atelier
101	British Lion
107	Columbia
102	Jean Luc Godard
113	Luso films
104	Mosfilm
103	Paramount
106	Toho
105	United Artists
111	Walt Disney

```

112|Warner Bros.
108|Westward

```

Este ejemplo muestra cómo obtener la union de las tablas `distributors` y `actors`, restringiendo los resultados a aquellos que comienzan con la letra W en cada tabla. No se quieren duplicados, así que la palabra clave `ALL` se omite.

distributors:	actors:
did name	id name
---+-----	--+-----
108 Westward	1 Woody Allen
111 Walt Disney	2 Warren Beatty
112 Warner Bros.	3 Walter Matthau
...	...

```

SELECT distributors.name
FROM   distributors
WHERE  distributors.name LIKE 'W%'
UNION
SELECT actors.name
FROM   actors
WHERE  actors.name LIKE 'W%'

```

```

name
-----
Walt Disney
Walter Matthau
Warner Bros.
Warren Beatty
Westward
Woody Allen

```

Compatibilidad

Extensiones

Postgres permite omitir la cláusula **FROM** de una consulta. Esta característica fue conservada del lenguaje original de consulta PostQuel:

```
SELECT distributors.* WHERE name = 'Westwood';
```

```

did|name
---+-----
108|Westward

```

SQL92

Cláusula SELECT

En el estándar SQL92, la palabra clave opcional "AS" es totalmente prescindible y puede ser omitida sin afectar el significado. El analizador sintáctico de Postgres requiere la presencia de esta palabra cuando se renombran columnas debido a las características de extensibilidad de tipos que pueden llevar a interpretaciones ambiguas en este contexto.

DISTINCT ON no es parte de SQL92. Tampoco los son LIMIT y OFFSET.

Cláusula UNION

La sintaxis de SQL92 para UNION admite una cláusula adicional CORRESPONDING BY:

```
table_query UNION [ALL]
    [CORRESPONDING [BY (column [, ...])]]
table_query
```

La cláusula CORRESPONDING BY no es soportada por Postgres.

SELECT INTO**Nombre**

SELECT INTO — Crear una nueva tabla a partir de una tabla o vista ya existente.

Synopsis

```
SELECT [ ALL | DISTINCT [ ON ( expresión [, ...] ) ] ]
    expresión [ AS nombre ] [, ...]
    [ INTO [ TEMPORARY | TEMP ] [ TABLE ] nueva_tabla ]
    [ FROM tabla [ alias ] [, ...] ]
    [ WHERE condición ]
    [ GROUP BY columna [, ...] ]
    [ HAVING condiciónn [, ...] ]
    [ { UNION [ ALL ] | INTERSECT | EXCEPT } select ]
    [ ORDER BY columna [ ASC | DESC | USING operador ] [, ...] ]
    [ FOR UPDATE [ OF Nombre_de_clase [, ...] ] ]
LIMIT { contador | ALL } [ { OFFSET | , } incio ]
```

Inputs

Todos los campos de entrada se describen en detalle en *SELECT*.

Outputs

Todos los campos de salida se describen en detalle en *SELECT*.

Descripción

SELECT INTO Crea una nueva tabla a partir del resultado de una query. Típicamente, esta query recupera los datos de una tabla existente, pero se permite cualquier query de SQL.

Nota: *CREATE TABLE AS* es funcionalmente equivalente al comando **SELECT INTO**.

SET

Nombre

SET — Fija parámetros de tiempo de ejecución para la sesión.

Synopsis

```
SET variable { TO | = } { 'value' | DEFAULT }
SET TIME ZONE { 'timezone' | LOCAL | DEFAULT }
SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }
```

Inputs (Valores de entrada)

variable

Parámetro global que se quiere fijar.

value

Nuevo valor del parámetro. Se puede utilizar el valor **DEFAULT** para especificar que se devuelve el parámetro a su valor de defecto.

Las variables posibles y los valores permitidos son:

CLIENT_ENCODING | NAMES

Fija la codificación para clientes multi-byte. Los parámetros son:

value

Fija la codificación de cliente multi-byte a: *value*. La codificación especificada debe estar soportada por el servidor.

Esta opción solo es utilizable si el soporte MULTIBYTE se autorizó durante el paso de configuración en la construcción de Postgres.

DateStyle

Fija el estilo de representación de fecha/hora. Afecta al formato de salida, y en algunos casos puede afectar a la interpretación de la entrada.

ISO

utiliza fechas y horas de estilo ISO 8601.

SQL

utiliza fechas y horas de estilo Oracle/Ingres.

Postgres

utiliza el formato tradicional de Postgres.

European

utiliza dd/mm/yyyy para la representación numérica de las fechas.

NonEuropean

utiliza mm/dd/yyyy para la representación numérica de las fechas.

German

utiliza dd.mm.yyyy para la representación numérica de las fechas.

US

igual que 'NonEuropean'

DEFAULT

recupera los valores de defecto ('US,Postgres')

La inicialización del formato de la fecha se puede hacer:

Fijando la variable de entorno PGDATESTYLE. Si PGDATESTYLE se fija en el ambiente de una aplicación

Ejecutando postmaster utilizando la opción `-o -e` se fijan las fechas a la convención Europea. Nótese

Cambiando las variables en `src/backend/utils/init/globals.c`.

Las variables de `globals.c` que se pueden cambiar son:

```
bool EuroDates = false | true
```

```
int DateStyle = USE_ISO_DATES | USE_POSTGRES_DATES | USE_SQL_DATES | USE_GERMAN_
```

SERVER_ENCODING

Fija la codificación multi-byte para el servidor.

value

Fija la codificación multi-byte para el servidor.

Esta opción sólo está disponible si se habilitó el soporte MULTIBYTE durante el paso de configuración de la construcción de Postgres.

TIMEZONE

Los valores posibles para timezone dependen de su sistema operativo. Por ejemplo, en Linux /usr/lib/zoneinfo contiene la base de datos de zonas horarias.

Aquí tiene algunos valores válidos para zonas horarias:

'PST8PDT'

situa la zona horaria de California.

'Portugal'

sitúa la zona horaria de Portugal.

'Europe/Rome'

sitúa la zona horaria de Italia.

DEFAULT

fija la zona horaria a su valor local. (el valor de la variable de entorno TZ).

Si se especifica una zona horaria invalida, será fijada a GMT (en la mayoría de sistemas en cualquier caso).

La segunda sintaxis mostrada más arriba, permite fijar la zona horaria con una sintaxis similar a **SET TIME ZONE** de SQL92. La palabra clave LOCAL es sólo un formato alternativo a DEFAULT para mantener la compatibilidad con SQL92.

Si la variable de entorno PGTZ se fija en el ambiente de la aplicación de un cliente basado en libpq (en el ambiente del frontend), libpq fijará automáticamente TIMEZONE al valor de PGTZ durante el arranque de la conexión.

TRANSACTION ISOLATION LEVEL

Fija el nivel de aislamiento para la transacción actual.

READ COMMITTED

Las consultas de la transacción actual leen sólo filas aseguradas (committed) antes de empezar una consulta. READ COMMITTED es el valor de defecto.

Nota: El estandar SQL92 requiere que se fije el valor de aislamiento de defecto a SERIALIZABLE.

SERIALIZABLE

Las consultas de la transacción llen sólo fila aseguradas antes de la primera instrucción DML (**SELECT/INSERT/DELETE/UPDATE/FETCH/COPY_TO**) que se ejecute en esta transacción.

Hay también varios parámetros internos o de optimización que se pueden especificar con el comando **SET**:

RANDOM_PAGE_COST

Fija la estimación del optimizador del coste de una página de disco leída no secuencialmente. Eso se mide como un múltiplo del coste de una lectura de página secuencial.

float8

Fija el coste de un acceso aleatorio a un página al valor punto flotante especificado.

CPU_TUPLE_COST

Fija la estimación que hará el optimizador del coste de procesar cada tupla durante una consulta. Esto se mide como una fracción del coste de una lectura secuencial de una página.

float8

Fija el coste de proceso de CPU por tupla al valor de de punto flotante especificado.

CPU_INDEX_TUPLE_COST

Fija la estimación que hará el optimizador sobre el coste de procesar cada tupla del índice durante el procesado de un barrido del índice (index scan). Se mide como una fracción del coste de una lectura secuencial de página.

float8

Fija el coste de CPU de procesado por tupla de índice al valor de punto flotante especificado.

CPU_OPERATOR_COST

Fija la estimación que hará el optimizador del coste de procesar cada operador en una cláusula WHERE. Esto se mide como una fracción del coste de un acceso secuencial a una página.

float8

Fija le coste de CPU para procesar cada operador al valor de punto flotante especificado.

EFFECTIVE_CACHE_SIZE

Fija la estimación que hará el optimizador sobre el tamaño efectivo de la caché en disco (es decir, la porción de la caché en disco del kernel que será utilizada por los ficheros de datos de Postgres). Esto se mide en páginas de disco, normalmente en piezas de 8 Kb.

float8

Fija el tamaño estimado de la caché en el valor de punto flotante especificado.

ENABLE_SEQSCAN

Habilita o inhabilita el uso por el planificador de tipos de planes de barrido secuencial. (No es posible suprimir completamente los barridos secuenciales, pero desactivando esta variable se disuade al planificador de utilizar uno de ellos si dispone de otro método utilizable).

ON

Habilita el uso de barridos secuenciales (valor de defecto).

OFF

Inhabilita el uso de barridos secuenciales.

ENABLE_INDEXSCAN

Habilita o inhabilita el uso por el planificador de tipos de planes de barrido de índices.

ON

Habilita el uso de barridos de índices (valor de defecto).

OFF

Inhabilita el uso de barridos de índices.

ENABLE_TIDSCAN

Habilita o inhabilita el uso por el planificador de tipos de planes por barrido TID.

ON

Habilita el uso de barridos TID (valor de defecto).

OFF

Inhabilita el uso de barridos TID.

ENABLE_SORT

Habilita o inhabilita el uso por el planificador pasos de ordenación explícita. (No es posible suprimir por completo las ordenaciones explícitas, pero fijando en

OFF esta variable disuade al planificador de usar uno cuando tiene otro método utilizable.)

ON

Habilita el uso de ordenaciones (valor de defecto).

OFF

Inhabilita el uso de ordenaciones.

ENABLE_NESTLOOP

Habilita o inhabilita el uso por el planificador de planes de join de bucle anidado. (No es posible suprimir por completo las joins de bucle anidado, pero fijar en OFF esta variable disuade al planificador de utilizar uno de ellos si dispone de otro método).

ON

Habilita el uso de joins de bucle anidado (valor de defecto).

OFF

Inshabilita el uso de joins de bucle anidado.

ENABLE_MERGEJOIN

Habilita o inhabilita el uso por el planificador de planes de tipo "enlace intercalado" (mergejoin).

ON

Habilita el uso de enlaces intercalados (valor de defecto).

OFF

Inhabilita el uso de enlaces intercalados.

ENABLE_HASHJOIN

Habilita o inhabilita el uso por el planificador de planes de tipo enlace hash (hashjoin).

ON

Habilita el uso de enlaces hash (valor de defecto).

OFF

Inhabilita el uso de enlaces hash.

GEQO

Fija el porcentaje de uso del algoritmo genérico del optimizador.

ON

Habilita el algoritmo genérico del optimizador para instrucciones con 11 tablas o más. (Este es también el valor de defecto DEFAULT).

ON=#

Toma un argumento entero para habilitar el algoritmo genérico para instrucciones con # o más tablas en la consulta.

OFF

Inhabilita el algoritmo genérico del optimizador.

Vea el capítulo sobre GEQO de la Guía del Programador para obtener más información sobre la optimización de la consulta.

Si la variable de entorno PGGEQO se fija en el ambiente de usuario de un cliente basado en libpq, libpq automáticamente fijará GEQO al valor de PGGEQO durante el arranque de la conexión.

KSQO

Key Set Query Optimizer (Optimizador de la Consulta Fijado por Clave) lleva al planificador de la consulta a convertir aquellas consultas cuyas cláusulas WHERE incluyan muchas cláusulas OR y AND (tales como "WHERE (a=1 AND b=2) OR (a=2 AND b=3) ...") en una consulta UNION. Este metodo puede ser más rápido que la implementación de defecto, pero no necesariamente produce exactamente el mismo resultado, puesto que UNION implícitamente añade una cláusula SELECT DISTINCT para eliminar las filas resultantes que sean idénticas. KSQO se utiliza habitualmente cuando se trabaja con productos como MicroSoft Access, que tienden a generar las consultas de esta forma.

ON

Habilita esta optimización.

OFF

Inhabilita esta optimización (valor de defecto).

DEFAULT

Equivalente a especificar **SET KSQO='OFF'**.

El algoritmo KSQO se utilizaba por ser absolutamente esencial para consultas con muchas cláusulas OR y AND, pero en Postgres 7.0 y posteriores, el planificador estandar manipula estas consultas correctamente.

Outputs

SET VARIABLE

Mensaje devuelto si se fija el valor con éxito.

WARN: Bad value for *variable* (*value*)

Si el comando falla al fijar el valor especificado.

Descripción

SET modificará los parámetros de configuración para la variable durante una sesión.

Los valores en vigor se pueden obtener utilizando el **SHOW**, y los valores pueden devolverse a su situación de defecto utilizando **RESET**. Valores y parámetros son sensibles a mayúsculas y minúsculas. Nótese que el campo “valor” siempre se especifica como una cadena de caracteres, de modo que se encierra entre comillas simples.

SET TIME ZONE cambia la asignación de zona horaria de defecto de la sesión. Una sesión SQL siempre empieza con un valor inicial de asignación de zona horaria. La instrucción **SET TIME ZONE** se utiliza para cambiar la asignación de zona horaria para la sesión SQL actual.

Notas

La instrucción **SET *variable*** es una extensión del lenguaje de Postgres.

Refierase a **SHOW** y **RESET** para mostrar o inicializar los valores actuales.

Uso

Fijar el estilo de la fecha a ISO:

```
SET DATESTYLE TO 'ISO';
```

Habilitar GEQO para consultas con 4 o más tablas:

```
SET GEQO ON=4;
```

Fijar GEQO a su valor de defecto:

```
SET GEQO = DEFAULT;
```

Fijar la zona horaria a Berkeley, California:

```
SET TIME ZONE 'PST8PDT';
SELECT CURRENT_TIMESTAMP AS ahora;
```

```
      ahora
-----
1998-03-31 07:41:21-08
```

Fijar la zona horaria para Italia:

```
SET TIME ZONE 'Europe/Rome';
SELECT CURRENT_TIMESTAMP AS ahora;
```

```
      ahora
-----
1998-03-31 17:41:31+02
```


Compatibilidad

SQL92

No hay **SET *variable*** general en SQL92 (con la excepción de **SET TRANSACTION ISOLATION LEVEL**). La sintaxis de SQL92 para **SET TIME ZONE** es ligeramente diferente, que permite sólo un único valor entero para la especificación de la zona horaria:

```
SET TIME ZONE { expresión_de_valor_del_intervalo | LOCAL }
```

SHOW

Nombre

SHOW — Muestra los parámetros en tiempo de ejecución de la sesión

Synopsis

```
SHOW palabra_clave
```

Entradas

palabra_clave

Veáse el comando **SET** para obtener más información de los argumentos disponibles.

Outputs

```
NOTICE: variable is value SHOW VARIABLE
```

Mensaje que se devuelve si todo ha ido bien.

```
NOTICE: Unrecognized variable value
```

Mensaje que se devuelve si *value* no existe.

```
NOTICE: Time zone is unknown SHOW VARIABLE
```

Si las variables de entorno TZ o PG TZ no están definidas.

Descripción

SHOW mostrará la configuración actual de un parámetro en tiempo de ejecución durante una sesión.

A estas variables se les puede asignar un valor usando la declaración **SET** y se puede restaurar su valor por defecto con la declaración **RESET**. Los parámetros y los valores son sensibles a mayúsculas y minúsculas.

Notas

SHOW es una extensión del lenguaje de Postgres.

Veáse **SET/RESET** para fijar los valores de una variable.

Utilización

Muestra el estilo de fecha (`DateStyle`):

```
SHOW DateStyle;
NOTICE:DateStyle is Postgres with US (NonEuropean) conventions
```

Muestra la configuración del optimizador genético (`geqo`):

```
SHOW GEQO;
NOTICE:GEQO is ON
```

Compatibilidad

SQL92

No hay ningún **SHOW** definido en SQL92.

TRUNCATE

Nombre

TRUNCATE — Vacía una tabla

Synopsis

```
TRUNCATE [ TABLA ] NOMBRE
```

Entradas

nombre

El nombre de la tabla a trincar.

Salidas

```
TRUNCATE
```

Mensaje retornado si la tabla ha sido vaciada (truncada) exitosamente.

Description

TRUNCATE remueve rapidamente todas las filas de una tabla. Tiene el mismo efecto que el **DELETE** pero al no recorrer la tabla resulta mas rapido. Es mas efectivo en tablas grandes.

Usage

Trincar la tabla `tablagrande`:

```
TRUNCATE TABLE tablagrande;
```

Compatibilidad

SQL92

El **TRUNCATE** no existe en SQL92.

UNLISTEN

Nombre

UNLISTEN — Deja de prestar atención a las notificaciones

Synopsis

```
UNLISTEN { nombre_notif | * }
```

Entradas

nombre_notif

Nombre de la notificación previamente registrada.

*

Se limpiarán todos los registros en escucha para este backend.

Salidas

UNLISTEN

Acuse de recibo de que la declaración se ha ejecutado.

Descripción

UNLISTEN se usa para borrar un registro **NOTIFY** existente. UNLISTEN cancela cualquier registro existente de la sesión actual de Postgres en la condición de notificación *nombre_notif*. La condición asterisco "*" cancela todos los registros "listener" de la sesión actual.

NOTIFY contiene una discusión más extensa del uso de **LISTEN** y **NOTIFY**.

Notas

nombre_clase no necesariamente ha de ser un nombre de clase válido, pero puede ser cualquier cadena (string) válida de hasta 32 caracteres de largo.

El backend no muestra errores si usted hace un UNLISTEN sobre algo al que no estuviera atendiendo (escuchando). Cada backend ejecutará automáticamente **UNLISTEN *** cuando termine.

Una restricción que se daba en versiones anteriores de Postgres, que hacía que un *nombre_clase* que no se correspondiera con la tabla en curso debía ser entrecomillada, ya no se da actualmente.

Usage

Para suscribirse a un registro existente:

```
postgres=> LISTEN virtual;
LISTEN
postgres=> NOTIFY virtual;
NOTIFY
ASYNC NOTIFY of 'virtual' from backend pid '12317' received
```

Una vez que UNLISTEN se ha ejecutado, posteriores comandos NOTIFY serán ignorados:

```
postgres=> UNLISTEN virtual;
UNLISTEN
postgres=> NOTIFY virtual;
NOTIFY
- notice no NOTIFY event is received
```

Compatibilidad

SQL92

No existe **UNLISTEN** en SQL92.

UPDATE

Nombre

UPDATE — Substituye valores de columnas en una tabla

Synopsis

```
UPDATE tabla SET columna = expresión [, ...]
    [ FROM lista ]
    [ WHERE condición ]
```

Entradas

tabla

El nombre de una tabla existente.

columna

El nombre de la columna en *tabla*.

expresión

Una expresión válida o valor a ser asignado a la columna.

lista

Es una extensión no estándar de Postgres que permite la aparición de columnas de otras tablas en la condición WHERE.

condición

Consulte la cláusula SELECT para una descripción más extensa de la cláusula WHERE.

Salidas

UPDATE

Mensaje obtenido si ha habido éxito. El símbolo # representa el número de filas que han sido actualizadas. Si # es igual a 0, ninguna fila fue actualizada.

Descripción

UPDATE cambia el valor de las columnas especificadas por todas las filas que satisfacen la condición dada. Solamente necesita indicar las columnas que serán modificadas.

Para referencias a listas se usa la misma sintaxis de *SELECT*. O sea, puede substituir un único elemento de una lista, un rango de elementos o una lista completa con una única petición.

Debe tener permiso de escribir en la tabla para poder modificarla, así como permiso de lectura de cualquier tabla cuyos valores sean mencionados en la condición WHERE.

Uso

Para cambiar la palabra "Drama" por "Dramática" en la columna categoría:

```
UPDATE películas
  SET categoría = 'Dramática'
  WHERE categoría = 'Drama';
SELECT * FROM películas WHERE categoría = 'Dramático' OR categoría = 'Drama';
```

code	título	did	fecha_prod	categoría	durac
-----+-----+-----+-----+-----					
BL101	El tercer hombre	101	1949-12-23	Dramática	01:44
P_302	Becket	103	1964-02-03	Dramática	02:28
M_401	La paz y la guerra	104	1967-02-12	Dramática	05:57

T_601 Yojimbo	106 1961-06-16 Dramática	01:50
DA101 Das Boot	110 1981-11-11 Dramática	02:29

Compatibilidad

SQL92

SQL92 define una sintaxis diferente para la cláusula UPDATE:

```
UPDATE tabla SET columna = expresión [, ...]
    WHERE CURRENT OF cursor
```

donde *cursor* identifica un cursor abierto.

VACUUM

Nombre

VACUUM — Limpia y analiza una base de datos Postgres

Synopsis

```
VACUUM [ VERBOSE ] [ ANALYZE ] [ tabla ]
VACUUM [ VERBOSE ] ANALYZE [ tabla [ (columna [, ...] ) ] ]
```

Entrada

VERBOSE

Imprime un reporte detallado de la actividad de vacuum para cada tabla.

ANALYZE

Actualiza las estadísticas de columnas usadas por el optimizador para determinar la manera más eficiente de ejecutar una consulta. Las estadísticas representan la dispersión de los datos en cada columna. Esta información es valiosa cuando hay la posibilidad de ejecución desde varios puntos.

tabla

El nombre de una tabla específica a la que se va a realizar el vacuum. El estándar es hacerlo a todas las tablas.

columna

El nombre de una columna específica a analizar. El estándar es hacerlo para todas las columnas.

Salida

VACUUM

El comando ha sido aceptado y la base de datos está siendo limpiada.

NOTICE: -Relation *tabla*-

El encabezado de reporte para *tabla*.

NOTICE: Pages 98: Changed 25, Reapped 74, Empty 0, New 0; Tup 1000: Vac 3000, Crash 0, UnUsed 0, MinLen 188, MaxLen 188; Re-using: Free/Avail. Space 586952/586952; EndEmpty/Avail. Pages 0/74. Elapsed 0/0 sec.

El análisis para la *tabla* misma.

NOTICE: Index *indice*: Página 28; Tuples 1000: Deleted 3000. Elapsed 0/0 sec.

El análisis para un índice en la tabla destino.

Descripción

VACUUM sirve para dos propósitos en Postgres como medio para reclamar almacenamiento, y también para recolectar información para el optimizador.

VACUUM abre cada clase en la base de datos, limpia los registros de transacciones ya pasadas y actualiza las estadísticas en los catálogos del sistema. Las estadísticas mantenidas incluyen el número de tuples y el número de páginas almacenadas en todas las clases.

La ejecución de **VACUUM** periódicamente aumentará la velocidad de la base de datos al procesar las consultas del usuario.

Notas

La base de datos abierta es el objetivo del comando **VACUUM**.

Recomendamos que la base de datos principal activa sea limpiada cada noche para mantener las estadísticas relativamente actualizadas. Sin embargo, la consulta **VACUUM** puede ser ejecutada en cualquier momento. Particularmente, después de copiar una clase grande en Postgres o después de borrar un gran número de registros, puede ser una buena idea emitir una consulta **VACUUM**. Esto actualizará los catálogos del sistema con todos los cambios recientes, y permitirá al organizador de consultas de Postgres tomar las mejores decisiones al planear las consultas de los usuarios.

Uso

El siguiente es un ejemplo de la ejecución del comando **VACUUM** en una tabla en la base de datos de regresión:

```
regresión=> vacuum verbose analyze onek;
NOTICE:  -Relation onek-
NOTICE:  Pages 98: Changed 25, Reapped 74, Empty 0, New 0;
          Tup 1000: Vac 3000, Crash 0, UnUsed 0, MinLen 188, MaxLen 188;
          Re-using: Free/Avail. Space 586952/586952; EndEmpty/Avail. Pa-
ges 0/74.
          Elapsed 0/0 sec.
NOTICE:  Index onek_stringul: Pages 28; Tuples 1000: Deleted 3000. Elap-
sed 0/0 sec.
NOTICE:  Index onek_hundred: Pages 12; Tuples 1000: Deleted 3000. Elap-
sed 0/0 sec.
NOTICE:  Index onek_unique2: Pages 19; Tuples 1000: Deleted 3000. Elap-
sed 0/0 sec.
NOTICE:  Index onek_unique1: Pages 17; Tuples 1000: Deleted 3000. Elap-
sed 0/0 sec.
NOTICE:  Rel onek: Pages: 98 -> 25; Tuple(s) moved: 1000. Elapsed 0/1 sec.
NOTICE:  Index onek_stringul: Pages 28; Tuples 1000: Deleted 1000. Elap-
sed 0/0 sec.
NOTICE:  Index onek_hundred: Pages 12; Tuples 1000: Deleted 1000. Elap-
sed 0/0 sec.
NOTICE:  Index onek_unique2: Pages 19; Tuples 1000: Deleted 1000. Elap-
sed 0/0 sec.
NOTICE:  Index onek_unique1: Pages 17; Tuples 1000: Deleted 1000. Elap-
sed 0/0 sec.
VACUUM
```

Compatibilidad

SQL92

No existe el comando **VACUUM** en SQL92.

Capítulo 15. Aplicaciones

Esta es la información de referencia para las aplicaciones y utilidades de soporte de Postgres.

createdb

Nombre

`createdb` — Crea una nueva base de datos PostgreSQL

Synopsis

`createdb` [*options*] *dbname* [*descripcion*]

Inputs

`-h, -host` *host*

Especifica el nombre de host (hostname) de la maquina sobre la que esta ejecutandose la `postmaster`.

`-p, -port` *port*

Especifica el puerto TCP/IP Internet o la extension del fichero de socket del dominio local Unix en el cual la `postmaster` esta escuchando para recibir conexiones.

`-U, -username` *username*

Usuario como el que se conecta.

`-W, -password`

Fuerza a que se teclee password.

`-e, -echo`

Muestra la consulta que `createdb` genera y envia al motor de la base de datos (backend)

`-q, -quiet`

No muestra ninguna respuesta.

`-D, -location` *datadir*

especifica localizacion alternativa de la base de datos para esta instalacion de la base de datos. Esta es la localizacion de las tablas del sistema, no la localizacion de esta base de datos especifica, que puede ser diferente.

`-E, -encoding encoding`

Especifica el esquema de codificación de caracteres que se usará con esta base de datos.

`dbname`

Especifica el nombre de la base de datos que será creada. El nombre debe ser único entre todas las bases de datos PostgreSQL en esta instalación. El valor por omisión es crear una base de datos con el mismo nombre que el usuario en curso del sistema.

`description`

Opcionalmente esto especifica un comentario que será asociado con la base de datos nuevamente creada.

Las opciones `-h`, `-p`, `-U`, `-W`, y `-e` son pasadas literalmente a *psql*.

Outputs

```
CREATE DATABASE
```

La base de datos fue creada exitosamente.

```
createdb: Creación de la base de datos fallida.
```

(Lo dice todo.)

```
createdb: Comentario a la creación fallida. (La base de datos fue cre-
da.)
```

El comentario/descripción para la base de datos que no ha podido ser creada. La base de datos misma podría haber sido creada ya. Puedes utilizar el comando SQL **COMMENT ON DATABASE** para crearle el comentario después.

Si hay un error en la condición, el error del motor de base de datos (backend) será mostrado. Véase *CREATE DATABASE* y *psql* para más posibilidades.

Descripción

`createdb` crea una nueva base de datos PostgreSQL. El usuario que ejecuta este comando se convierte en el propietario de la base de datos.

`createdb` es una script shell que envuelve un comando SQL *CREATE DATABASE* a través del terminal interactivo de PostgreSQL *psql*. Así pues, no hay nada especial sobre la creación de bases de datos por este u otros métodos. Esto significa que el *psql* debe ser encontrado por el script y que un servidor de base de datos está ejecutándose en el hosts destino. También, cualquier configuración por defecto y variable de entorno disponible para *psql* y la librería front-end *libpq* se aplicarán.

Uso

Para crear la base de datos `demo` utilizando el servidor por defecto de base de datos:

```
$ createdb demo
CREATE DATABASE
```

La respuesta es la misma que hubieses tenido de ejecutar el comando de SQL **CREATE DATABASE**.

Para crear una base de datos `demo` utilizando la `postmaster` en la maquina (host) `eden`, puerto 5000, utilizando el esquema de codificación `LATIN1` con una mirada en la consulta subrayada:

```
$ createdb -p 5000 -h eden -E LATIN1 -e demo
CREATE DATABASE "demo" WITH ENCODING = 'LATIN1'
CREATE DATABASE
```

createlang

Nombre

`createlang` — Añade una nuevo lenguaje de programación a una base de datos PostgreSQL

Synopsis

```
createlang [ opciones_conexion ] [ nom_leng [ nombre_bd ] ]
createlang [ opciones_conexion ] -lista|-l
```

Inputs

`createlang` acepta los siguientes argumentos:

langname

Especifica el nombre del lenguaje de programación del backend que va a ser definido. `createlang` preguntará por *langname* si no está definido en la línea de comandos.

`[-d, -dbname] nombre_bd`

Especifica a qué base de datos se va a añadir el lenguaje.

`-l, -list`

Muestra una lista de los lenguajes ya instalados en la base de datos destino (que debe ser especificada).

`createlang` también acepta los siguientes argumentos en la línea de comandos como parámetros de conexión:

`-h, -host host`

Especifica el nombre de host de la máquina sobre la que `postmaster` está corriendo.

`-p, -puerto port`

Especifica el puerto TCP/IP o el socket del dominio Unix en el que el `postmaster` está atendiendo a las conexiones.

`-U, -nombre usuario username`

Usuario con el que se va a conectar.

`-W, -password`

Fuerza a que se pregunte el password.

Outputs

La mayoría de los mensaje de error son lo suficientemente explicativos. Si no es así, ejecute `createlang` con la opción `-echo` y vea el comando SQL correspondiente para más detalles. Pruebe también bajo `psql` para ver más posibilidades.

Descripción

`createlang` es una utilidad para añadir un nuevo lenguaje de programación a una base de datos PostgreSQL. Actualmente `createlang` currently acepta dos lenguajes: `plsql` y `pltcl`.

Aunque los lenguajes de programación del backend pueden ser añadidos directamente usando varios comandos SQL, se recomienda usar `createlang` porque hace una serie de chequeos y es más fácil de usar. Vea `CREATE LANGUAGE` para más información.

Notas

Utilice `droplang` para borrar un lenguaje.

Uso

Para instalar `pltcl`:

```
$ createlang pltcl
```

createuser**Nombre**

`createuser` — Crea un nuevo usuario PostgreSQL

Synopsis

`createuser` [*opciones*] [*nombre_usuario*]

Inputs

`-h, -host` *host*

Especifica el nombre del host de la máquina sobre la que el `postmaster` corre.

`-p, -puerto` *puerto*

Especifica el puerto TCP/IP o el socket local Unix sobre el que el `postmaster` atiende a las conexiones.

`-e, -echo`

Muestra las consultas que `createdb` genera y envía al backend.

`-q, -quiet`

No muestra respuesta alguna.

`-d, -createdb`

Permite al nuevo usuario crear bases de datos.

`-D, -no-createdb`

Impide al nuevo usuario crear bases de datos.

`-a, -adduser`

Permite al nuevo usuario crear otros usuarios.

`-A, -no-adduser`

Impide al nuevo usuario crear otros usuarios.

`-P, -pwprompt`

Si se especifica este parámetro, `createuser` mostrará un mensaje preguntando por el password del nuevo usuario. Esto no es necesario si no planea usar autenticación por password.

`-i, -sysid` *id_usuario*

Le permite elegir otro id de usuario que no sea el que se da por defecto. Esto no es necesario, pero a algunos les gusta.

nombre_usuario

Especifica el nombre del usuario PostgreSQL que se va a crear. Este nombre debe ser único dentro de todos los existentes en PostgreSQL .

Se le preguntará por un nombre y cualquier otra información que no se haya especificado en la línea de comandos.

Las opciones `-h`, `-p`, y `-e`, son pasadas literalmente a *psql*. Las opciones *psql* `-U` y `-w` también se pueden usar, pero su uso puede ser confuso en este contexto.

Outputs

```
CREATE USER
```

Todo ha ido bien.

```
createuser: creation of user "username" failed
```

Algo no salió bien. El usuario no fue creado.

Si se da un error, el mensaje de error del backend se mostrará. Vea *CREAR USUARIO* y *psql* para más posibilidades.

Descripción

createuser crea un nuevo usuario PostgreSQL . Solamente los usuarios con *usesuper* activado en la clase *pg_shadow* pueden crear nuevos usuarios Postgres .

createuser es un envoltorio del shell script entorno al comando SQL *CREAR USUARIO* a través del terminal interactivo *psql* de PostgreSQL . Así, no hay nada especial en el momento de crear usuarios por medio de estos otros métodos. Esto significa que *psql* debe ser encontrado por el script y que un servidor de bases de datos está corriendo en la máquina al que se accede. Asimismo, cualquier valor por defecto y cualquier variable de entorno disponible para *psql* y *libpq* se aplican.

Uso

Para crear un usuario *joe* en la base de datos por defecto:

```
$ createuser joe
Is the new user allowed to create databases? (y/n) n
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER
```

Para crear al mismo usuario *joe* usando el *postmaster* en la máquina *eden*, puerto 5000, evitando las preguntas en el prompt y teniendo en cuenta la consulta en curso:

```
$ createuser -p 5000 -h eden -D -A -e joe
CREATE USER "joe" NOCREATEDB NOCREATEUSER
CREATE USER
```


dropdb**Nombre**

dropdb — Borra una base de datos PostgreSQL existente

Synopsis

dropdb [*opciones*] *nombre_bd*

Inputs

-h, -host *host*

Especifica el nombre de host de la máquina sobre la que el postmaster esté corriendo.

-p, -port *puerto*

Especifica el puerto TCP/IP o el socket Unix local en el que postmaster atiende conexiones.

-U, -username *nombre_usuario*

Nombre de usuario con el que se va a conectar.

-W, -password

Fuerza la introducción de un password.

-e, -echo

Muestra en pantalla las consultas que dropdb genera y envía al backend.

-q, -quiet

No muestra respuesta alguna.

-i, -interactive

Antes de hacer algo destructivo, pide confirmación a través del prompt.

nombre_bd

Especifica el nombre de la bases de datos que va a ser borrada. Debe ser una de las existentes en esta instalación de PostgreSQL .

Las opciones -h, -p, -U, -W, y -e se pasan literalmente a *psql*.

Outputs

```
DROP DATABASE
```

La base de datos ha sido borrada con éxito.

```
dropdb: Database removal failed.
```

Algo no ha ido bien.

Si se produce un error, se mostrará el mensaje de error del backend. Vea `drop_database` y *psql* para más información.

Descripción

`dropdb` destruye una base de datos PostgreSQL existente. El usuario que ejecute este comando debe ser un superusuario de la base de datos o su propietario.

`dropdb` es un envoltorio del shell script alrededor del comando SQL `drop_database` por medio del terminal interactivo *psql* de PostgreSQL. De este modo, no hay nada especial en borrar bases de datos por medio de este u otros métodos. Esto significa que *psql* debe ser encontrado por el script y que un servidor de bases de datos está en marcha en el host de destino. También cualquier valor por defecto o cualquier variable de entorno disponible para *psql* y *libpq* se aplican.

Uso

Para destruir la base de datos `demo` en el servidor de bases de datos por defecto:

```
$ dropdb demo
DROP DATABASE
```

Para destruir la base de datos `demo` usando el postmaster del host `eden`, puerto 5000, con verificación y echando un vistazo a la consulta en marcha:

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE "demo"
DROP DATABASE
```

droplang

Nombre

`droplang` — Borra un lenguaje de programación de una base de datos PostgreSQL

Synopsis

```
droplang [ opciones de conexión ] [ nombre_lenguaje [ nombre_bd ] ]
droplang [ opciones de conexión ] -list|-l
```

Inputs

`droplang` acepta los siguientes argumento en la línea de comandos:

nombre_lenguaje

Especifica el nombre del lenguaje de programación del backend que se va a borrar. `droplang` preguntará por *nombre_lenguaje* si no se ha especificado en la línea de comandos.

`[-d, -dbname] nombre_bd`

Especifica desde qué base de datos se debe borrar el lenguaje.

`-l, -list`

Muestra la lista de los lenguajes ya instalados en la base de datos destino (que debe ser especificada).

`droplang` también acepta los siguientes argumentos en la línea de comandos como parámetros de conexión:

`-h, -host host`

Especifica el nombre de host de la máquina sobre la que corre `postmaster`.

`-p, -port puerto`

Especifica el puerto TCP/IP o el socket local sobre el que `postmaster` atiende a las conexiones.

`-U, -username nombre_usuario`

Nombre de usuario con el que se vaya a conectar.

`-W, -password`

Fuerza la utilización de un password.

Outputs

La mayoría de los mensajes de error se explican por sí solos. Si no es así, ejecútelo `droplang` con la opción `-echo` y mire bajo el comando SQL correspondiente para más detalles. Mire también bajo *psql*.

Descripción

`droplang` es una utilidad para borrar un lenguaje de programación existente en la base de datos PostgreSQL. `droplang` actualmente acepta dos lenguajes: `plsql` y `pltcl`.

Aunque los lenguajes de programación del backend pueden ser borrados directamente utilizando varios comandos SQL, es recomendable usar `droplang` porque realiza comprobaciones y es más fácil de usar. Vea *DROP LANGUAGE* para más detalles.

Notas

Utilice *createlang* para agregar un lenguaje.

Uso

Para borrar `pltcl`:

```
$ droplang pltcl
```

dropuser

Nombre

`dropuser` — Borra un usuario Postgres

Synopsis

```
dropuser [ opciones ] [ nombre_usuario ]
```

Inputs

`-h, --host host`

Especifica el nombre de host de la máquina en la que el postmaster se está ejecutando.

`-p, --port puerto`

Especifica el puerto TCP/IP el socket local sobre el que postmaster escucha conexiones.

`-e, --echo`

Muestra en pantalla las consultas que `createdb` genera y envía al backend.

`-q, --quiet`

No muestra respuesta alguna.

`-i, -interactive`

Antes de borrar al usuario, pregunta.

`nombre_usuario`

Especifica el nombre de usuario PostgreSQL que va a ser borrado. Este nombre debe existir en la instalación Postgres. Se le preguntará un nombre si no se ha especificado ninguno en la línea de comandos.

Las opciones `-h`, `-p`, y `-e`, son pasadas literalmente a *psql*. Las opciones `psql -U` y `-w` también están disponibles, pero pueden ser confusas en este contexto.

Outputs

```
DROP USER
```

Todo ha ido bien.

```
dropuser: deletion of user "username" failed
```

Algo salió mal. No se ha borrado al usuario.

Cuando se da un error, el mensaje de error del backend será mostrado. Vea *DROP USER* y *psql* para más posibilidades.

Descripción

`dropuser` borrar un usuario PostgreSQL existente y las bases de datos que ese usuario posee. Solamente los usuarios con `usesuper` activado en la clase `pg_shadow` pueden destruir usuarios de PostgreSQL.

`dropuser` es un envoltorio del shell script alrededor del comando SQL *DROP USER* por medio del terminal interactivo *psql* de PostgreSQL. De este modo, no hay nada especial en borrar bases de datos por medio de este u otros métodos. Esto significa que `psql` debe ser encontrado por el script y que un servidor de bases de datos está en marcha en el host de destino. También cualquier valor por defecto o cualquier variable de entorno disponible para `psql` y `libpq` se aplican.

Uso

Para borrar al usuario `joe` del servidor de bases de datos por defecto:

```
$ dropuser joe
DROP USER
```

Para borrar al usuario `joe` usando el postmaster en el host `eden`, puerto 5000, con verificación y echando un vistazo a la consulta en curso:

```
$ dropuser -p 5000 -h eden -i -e joe
User "joe" and any owned databases will be permanently deleted.
Are you sure? (y/n) y
```

```
DROP USER "joe"  
DROP USER
```

ecpg

Nombre

`ecpg` — Embedded SQL C preprocessor (preprocesador C incorporado en SQL)

Synopsis

```
ecpg [ -v ] [ -t ] [ -I include-path ] [ -o outfile ] file1 [ file2 ] [ ... ]
```

Inputs

`ecpg` acepta los siguiente argumentos en línea de comandos:

file

Outputs

`ecpg` creará un fichero o escribirá en `stdout` (salida estándar).

Descripción

pgaccess

Nombre

`pgaccess` — Cliente gráfico interactivo dePostgres

Synopsis

`pgaccess [dbname]`

Entradas

dbname

El nombre de una base de datos existente.

Salidas

Descripción

`pgaccess` proporciona una interfaz gráfica para Postgres donde se pueden gestionar las tablas, editarlas, definir consultas, secuencias y funciones.

Otra forma de acceder a Postgres a través de tcl es con el uso de *pgtclsh* o *pgtksh*.

`pgaccess` permite:

- Abrir cualquier bases de datos en un determinado host, especificando dicho host, el puerto, el puerto especificado, el nombre de usuario y password.
- Ejecutar *VACUUM*.
- Guardar preferencias en el archivo `~/.pgaccessrc`.

Con tablas, `pgaccess` permite:

- Abrir múltiples tablas para visualización, con un máximo de *n* registros (configurable).
- Cambiar el tamaño de las columnas desplazando las líneas verticales que la forman.
- Introducir texto en celdas.
- Ajustar dinámicamente la altura de la celda durante la edición.
- Guardar un formato de tabla para cada tabla.
- Importar / exportar a / de archivos externos (SDF, CSV).
- Usar filtros; introducir filtros como `precio>3.14`.
- Especificar el orden; introducir manualmente los campos por los que realizar la ordenación.
- Edición; doble click sobre el texto que queremos cambiar.
- Borrar registros; situándose en el registro, se pulsa la tecla Del.
- Añadir nuevos registros; guardar nuevo registro con el botón derecho del ratón.
- Crear tablas con un asistente.
- Renombrar y borrar (drop) tablas.
- Recuperar información sobre las tablas, incluyendo propietario, información de los campos, índices.

Con consultas, `pgaccess` permite:

- Definir, editar y almacenar *user defined queries*.
- Guardar formatos de vistas.
- Almacenar consultas como vistas.
- Ejecutar con parámetros opcionales de entrada introducidos por el usuario; p.ej.

```
select * from invoices where year=[parameter "Year of selection"]
```
- Visualizar cualquier resultado de una consulta de selección (select).
- Ejecutar consultas de acción (insert, update, delete).
- Definir consultas usando un constructor visual de consultas con soporte drag & drop, y aliasing de tablas.

Con secuencias, `pgaccess` permite:

- Definir nuevas instancias.
- Inspeccionar instancias existentes.
- Borrar.

Con vistas, `pgaccess` permite:

- Definirlas salvando consultas como vistas.
- Visualizarlas, con posibilidades de ordenación y filtrado.
- Diseñar nuevas vistas.
- Borrar (drop) vistas existentes.

Con funciones, `pgaccess` permite:

- Definirlas.
- Inspeccionarlas.
- Borrarlas.

Con informes, `pgaccess` permite:

- Generar informes simples desde una tabla (beta stage).
- Cambiar fuente, tamaño y estilo de campos y etiquetas.
- Cargar y guardar informes de la base de datos.
- Previsualizar tablas, muestras de impresiones postscript.

Con formularios, `pgaccess` permite:

- Abrir formularios definidos por el usuario.
- Usar un módulo de diseño de formularios.
- Acceder a conjuntos de registros usando widget de consultas.

Con scripts, `pgaccess` permite:

- Definirlos.
- Modificarlos.
- Llamar scripts definidos por el usuario.

pgadmin**Nombre**

`pgadmin` — Postgres es una herramienta de diseño y mantenimiento de bases de datos para Windows 95/98/NT

Synopsis

`pgadmin [nombre de datosfuente [nombre de usuario [palabraclave]]]`

Entradas

nombre de datosfuente

El nombre de un sistem ODBC PostgreSQL existente o Datos fuente del Usuario.

nombre de usuario

Un nombre de usuario válido para el especificado *nombre de datosfuente*.

palabra clave

Una palabra clave válida para el especificado *nombre de datosfuente* y *nombre de usuario*.

Resultados**Descripción**

`pgadmin` es una herramienta de propósito general para diseñár, mantener, y administrar las bases de datos de Postgres. Funciona bajo Windows 95/98 y NT.

Características incluidas:

- Entradas SQL aleatorias.
- Pantallas de información y 'Ayudas' para bases de datos, tablas, índices, secuencias, vistas, programas de arranque, funciones y lenguajes.
- Preguntas y respuestas para configurar Usuarios, Grupos y Privilegios.
- Control de revisión con mejora de la generación de script.
- Configuración de las tablas de Microsoft MSysConf.

- ‘Ayudas’ para importar y exportar datos.
- ‘Ayuda’ para migrar Bases de datos.
- Informes predefinidos en bases de datos, tablas, índices, secuencias, lenguajes y vistas.

pgadmin se distribuye separadamente de Postgres y puede ser descargado desde la dirección <http://www.pgadmin.freemove.co.uk>¹

Notas

1. <http://www.pgadmin.freemove.co.uk>

pg_dump

Nombre

`pg_dump` — Extrae una base de datos Postgres a un fichero de script

Synopsis

```
pg_dump [ base_de_datos ]
pg_dump [ -h huésped ] [ -p puerto ]
      [ -t tabla ]
      [ -a ] [ -c ] [ -d ] [ -D ] [ -n ] [ -N ]
      [ -o ] [ -s ] [ -u ] [ -v ] [ -x ]
      [ base_de_datos ]
```

Entrada

`pg_dump` acepta los siguientes argumentos de la línea de comando:

base_de_datos

Especifica el nombre de la base de datos que se va a extraer. *base_de_datos* tiene como estándar el valor de la variable de entorno USER

-a

Vuelca sólo los datos, no el esquema (las definiciones).

-c

Limpia el esquema antes de crearlo.

-d

Vuelca la data como propios insertos de cadenas.

-D

Vuelca la data como insertos con nombres de atributos

-n

Suprime las dobles comillas de los identificadores, a menos que sean absolutamente necesarias. Esto puede causar problemas al cargar la misma si esta data volcada contiene palabras reservadas usadas por los identificadores. Esta era la conducta estándar en `pg_dump` pre-v6.4.

-N

Incluye comillas dobles en los identificadores. Este es el estándar.

-o

Vuelca los identificadores de objetos (OIDs) para cada tabla.

-s

Vuelca solo el esquema (las definiciones), no la data.

-t *tabla*

Vuelca la data para la *tabla* únicamente.

-u

Usa autenticación por medio de clave de acceso. Pide un nombre de usuario y clave de acceso.

-v

Especifica el modo verbose(parlanchín)

-x

Evita el volcado de ACLs (comandos grant/revoke) y la información de propiedad de la tabla.

`pg_dump` también acepta los siguientes argumentos de línea de comando para parámetros de conexión:

-h *huésped*

Especifica el nombre del huésped de la máquina en la cual se está ejecutando el `postmaster`. El estándar es usar un socket de dominio local Unix en vez de una conexión IP..

-p *puerto*

Especifica el puerto de Internet TCP/IP o extensión de archivo socket de dominio local Unix en el cual `postmaster` está esperando que se efectúen conexiones. En número estándar de puerto es 5432, o el valor de la variable de ambiente `PGPORT` (si está establecida).

-u

Usa autenticación con clave de acceso. Pide *nombre_de_usuario* y *clave_de_acceso*.

Salida

`pg_dump` creará un fichero o escribirá a `stdout`.

La conexión con la base de datos `'templatel'` falló. `connectDB()` falló: ¿Está el `postmaster` ejecutándose y aceptando conexiones en el `'Socket de UNIX'` en el puerto `'puerto'`?

`pg_dump` no pudo unirse al proceso `postmaster` en el huésped y puerto especificados. Si ve usted este mensaje, verifique que `postmaster` se este ejecutando en el huésped indicado, y que usted especificó el puerto correcto. Si su site usa algún sistema de autenticación, verifique que usted tiene las credenciales de autenticación requeridas.

La conexión con la base de datos `'base_de_datos'` falló. FATAL 1: `SetUserId: el usuario 'nombre_de_usuario' no está en 'pg_shadow'`

Usted no posee una entrada válida en la relación `pg_shadow` y no le será permitido tener acceso a Postgres. Contacte a su administrador de Postgres.

`dumpSequence(tabla): SELECT` falló

Usted carece del permiso para leer la base de datos. Contacte a su administrador de site Postgres.

Nota: `pg_dump` ejecuta internamente las directivas **SELECT**. Si tiene problemas ejecutando `pg_dump`, verifique que puede seleccionar la información de la base de datos mediante el uso de, por ejemplo, `psql`.

Descripción

`pg_dump` es un utilitario para volcar una base de datos Postgres en un fichero de script conteniendo comandos de consulta. Los ficheros de script son en formato de texto y pueden ser usados para reconstruir la base de datos, incluso en otras máquinas y con otras arquitecturas. `pg_dump` producirá las consultas necesarias para regenerar todos los tipos definidos por el usuario, funciones, tablas, índices, agregados, y operadores. Adicionalmente, toda la data es copiada en formato de texto el cual puede ser nuevamente copiado, también puede ser importado a herramientas para su edición.

`pg_dump` es útil para verter el contenido de una base de datos que se vaya a mudar de una instalación de Postgres a otra. Después de ejecutar `pg_dump`, se debe examinar el script de salida a ver si contiene alguna advertencia, especialmente a la luz de las limitaciones citadas en la parte inferior.

Notas

`pg_dump` tiene pocas limitaciones. Las limitaciones surgen principalmente de la dificultad para extraer ciertas meta-informaciones de los catálogos del sistema.

- `pg_dump` no entiende los índices parciales. La razón es la misma citada anteriormente; los predicados de los índices parciales se almacenan como planos.

- `pg_dump` no maneja objetos grandes. Los objetos grandes son ignorados y se debe lidiar con ellos de forma manual.

Uso

Para volcar una base de datos del mismo nombre que el usuario:

```
% pg_dump > db.out
```

Para volver a cargar esta base de datos:

```
% psql -e base_de_datos < db.out
```

`pg_dumpall`

Nombre

`pg_dumpall` — Extrae todas las bases de datos Postgres en un archivo de script

Synopsis

```
pg_dumpall
pg_dumpall [ -h máquina ] [ -p puerto ] [ -a ] [ -d ] [ -D ] [ -O ] [ -s ] [ -u ] [ -v ] [ -x ]
```

Entradas

`pg_dumpall` acepta los siguientes argumentos de la línea de órdenes:

`-a`

Vuelca sólo los datos, no el esquema (las definiciones).

`-d`

Vuelca los datos como inserciones de cadenas adecuadas.

`-D`

Vuelca los datos como inserciones con nombres de atributos

-n

Suprime las dobles comillas de los identificadores, a menos que sean absolutamente necesarias. Esto puede causar problemas al cargar estos datos volcados si hay palabras reservadas usadas como identificadores.

-o

Vuelca los identificadores de objetos (OIDs) de cada tabla.

-s

Vuelca sólo el esquema (las definiciones), no los datos.

-u

Usa autenticación con clave de acceso. Pide un nombre de usuario y una clave de acceso.

-v

Especifica el modo verbose (detallado)

-x

Evita el volcado de ACLs (órdenes grant/revoke) e información del propietario de la tabla.

`pg_dumpall` también acepta los siguientes argumentos en la línea de órdenes como parámetros de conexión:

-h *huésped*

especifica el nombre de la máquina en la cual se está ejecutando `postmaster`. El estándar es usar un socket de dominio local Unix en vez de una conexión IP.

-p *puerto*

Especifica el puerto Internet TCP/IP o el fichero de dominio local Unix en el cual esté `postmaster` aguardando conexiones. El número estándar de puerto es 5432, o el valor de la variable de entorno `PGPORT` (si se ha indicado).

-u

Usa autenticación con clave de acceso. Pide *nombre_de_usuario* y *clave_de_acceso*.

Salida

`pg_dumpall` creará un fichero o escribirá a `stdout`.

La conexión a la base de datos 'template1' falló. connectDB() falló: ¿Está `postmaster` ejecutándose y aceptando conexiones en el 'Socket UNIX' en el puerto '*puerto*'?

`pg_dumpall` no pudo unirse al proceso `postmaster` en la máquina y puerto especificados. Si ve usted este mensaje, verifique que `postmaster` esté ejecutándose correctamente en el huésped y puerto que usted especificó. Si su lugar de

trabajo usa algún sistema de autenticación verifique que usted ha obtenido las credenciales de autenticación.

La conexión a la base de datos '*base_de_datos*' falló. FATAL 1: SetUserId: el usuario '*nombre_de_usuario*' no está en '*pg_shadow*'

Usted no tiene una entrada válida en la relación *pg_shadow* y no le será permitido el acceso a Postgres. Contacte con su administrador Postgres.

`dumpSequence(tabla): SELECT` falló

No tiene permiso para leer la base de datos. Contacte a su administrador Postgres.

Nota: *pg_dumpall* ejecuta internamente directivas **SELECT**. Si tiene problemas ejecutando *pg_dumpall*, asegúrese de que puede consultar información de la base de datos usando, por ejemplo, *psql*.

Descripción

pg_dumpall se diseñó para volcar todas las bases de datos Postgres en un fichero. También vuelca la tabla *pg_shadow*, la cual es global para todas las bases de datos. *pg_dumpall* incluye en este archivo las órdenes correctas para crear automáticamente cada una de las bases de datos volcadas antes de cargar los datos.

pg_dumpall toma todas las opciones de *pg_dump* pero *-f*, *-t* y *base_de_datos* deberían ser omitidos.

Refiérase a *pg_dump* para más información con respecto a esta otra utilidad.

Uso

Para volcar todas las bases de datos:

```
% pg_dumpall > db.out
```

Sugerencia: Puede usar la mayoría de las opciones de *pg_dump* con *pg_dumpall*.

Para volver a cargar esta base de datos:

```
% psql -e template1 < db.out
```

Sugerencia: Puede usar la mayoría de las opciones de *psql* cuando vuelva a cargarlas.

psql**Nombre**

`psql` — PostgreSQL interactive terminal

Synopsis

```
psql [ options ] [ dbname [ user ] ]
```

Summary

`psql` is a terminal-based front-end to PostgreSQL. It enables you to type in queries interactively, issue them to PostgreSQL, and see the query results. Alternatively, input can be from a file. In addition, it provides a number of meta-commands and various shell-like features to facilitate writing scripts and automating a wide variety of tasks.

Description**Connecting To A Database**

`psql` is a regular PostgreSQL client application. In order to connect to a database you need to know the name of your target database, the hostname and port number of the server and what user name you want to connect as. `psql` can be told about those parameters via command line options, namely `-d`, `-h`, `-p`, and `-U` respectively. If an argument is found that does not belong to any option it will be interpreted as database name as well. Not all these options are required, defaults do apply. If you omit the host name `psql` will connect via domain sockets to a server on the local host. The default port number is compile-time determined. Since the database server uses the same default, chances are you don't have to specify the port in most settings. The default user name is your Unix username, the same with the database. Note that you can't just connect to any database under any username. Your database administrator should have informed you about your access rights. To save you some typing you can also set the environment variables `PGDATABASE`, `PGHOST`, `PGPORT`, `PGUSER`, respectively to appropriate values.

If the connection could not be made for any reason (e.g., insufficient privileges, postmaster is not running on the server, etc.), `psql` will return an error and terminate.

Entering Queries

In normal operation, `psql` provides a prompt with the name of the database that `psql` is currently connected to followed by the string `"=>"`. For example,

```
$ psql testdb
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
```



```

\? for help on internal slash commands
\g or terminate with semicolon to execute query
\q to quit

testdb=>

```

At the prompt, the user may type in SQL queries. Ordinarily, input lines are sent to the backend when a query-terminating semicolon is reached. An end of line does not terminate a query! Thus queries can be spread over several lines for clarity. If the query was sent and without error, the query results are displayed on the screen.

Whenever a query is executed, `psql` also polls for asynchronous notification events generated by *LISTEN* and *NOTIFY*.

psql Meta-Commands

Anything you enter in `psql` that begins with an unquoted backslash is a `psql` meta-command that is processed by `psql` itself. These commands are what makes `psql` interesting for administration or scripting. Meta-commands are more commonly called slash or backslash commands.

The format of a `psql` command is the backslash, followed immediately by a command verb, then any arguments. The arguments are separated from the command verb and each other by any number of white space characters.

To include whitespace into an argument you must quote it with a single quote. To include a single quote into such an argument, precede it by a backslash. Anything contained in single quotes is furthermore subject to C-like substitutions for `\n` (new line), `\t` (tab), `\digits`, `\odigits`, and `\oxdigits` (the character with the given decimal, octal, or hexadecimal code).

If an unquoted argument begins with a colon (:), it is taken as a variable and the value of the variable is taken as the argument instead.

Arguments that are quoted in “backticks” (```) are taken as a command line that is passed to the shell. The output of the command (with a trailing newline removed) is taken as the argument value. The above escape sequences also apply in backticks.

Some commands take the name of an SQL identifier (such as a table name) as argument. These arguments follow the syntax rules of SQL regarding double quotes: an identifier without double quotes is coerced to lower-case. For all other commands double quotes are not special and will become part of the argument.

Parsing for arguments stops when another unquoted backslash occurs. This is taken as the beginning of a new meta-command. The special sequence `\\` (two backslashes) marks the end of arguments and continues parsing SQL queries, if any. That way SQL and `psql` commands can be freely mixed on a line. But in any case, the arguments of a meta-command cannot continue beyond the end of the line.

The following meta-commands are defined:

`\a`

If the current table output format is unaligned, switch to aligned. If it is not unaligned, set it to unaligned. This command is kept for backwards compatibility. See `\pset` for a general solution.

`\c [title]`

Set the title of any tables being printed as the result of a query or unset any such title. This command is equivalent to `\pset title title`. (The name of this command derives from “caption”, as it was previously only used to set the caption in an HTML table.)

`\connect (or \c) [dbname [username]]`

Establishes a connection to a new database and/or under a user name. The previous connection is closed. If *dbname* is – the current database name is assumed.

If *username* is omitted the current user name is assumed.

As a special rule, `\connect` without any arguments will connect to the default database as the default user (as you would have gotten by starting `psql` without any arguments).

If the connection attempt failed (wrong username, access denied, etc.) the previous connection will be kept if and only if `psql` is in interactive mode. When executing a non-interactive script, processing will immediately stop with an error. This distinction was chosen as a user convenience against typos on the one hand, and a safety mechanism that scripts are not accidentally acting on the wrong database on the other hand.

`\copy table [with oids] { from | to } filename | stdin | stdout [with delimiters 'characters'] [with null as 'string']`

Performs a frontend (client) copy. This is an operation that runs an SQL `COPY` command, but instead of the backend reading or writing the specified file, and consequently requiring backend access and special user privilege, as well as being bound to the file system accessible by the backend, `psql` reads or writes the file and routes the data to or from the backend onto the local file system.

The syntax of the command is in analogy to the SQL `COPY` command, see its description for the details. Note that because of this, special parsing rules apply to the `\copy` command. In particular, the variable substitution rules and backslash escapes do not apply.

Sugerencia: This operation is not as efficient as the SQL `COPY` command because all data must pass through the client/server IP or socket connection. For large amounts of data the other technique may be preferable.

Nota: Note the difference in interpretation of `stdin` and `stdout` between frontend and backend copies: In a frontend copy these always refer to `psql`'s input and output stream. On a backend copy `stdin` comes from wherever the `COPY` itself came from (for example, a script ran with the `-f` option, and `stdout` refers to the query output stream (see `\o` meta-command below).

`\copyright`

Shows the copyright and distribution terms of PostgreSQL.

`\d relation`

Shows all columns of *relation* (which could be a table, view, index, or sequence), their types, and any special attributes such as `NOT NULL` or defaults, if any. If the relation is, in fact, a table, any defined indices are also listed. If the relation is a view, the view definition is also shown.

The command form `\d+` is identical, but any comments associated with the table columns are shown as well.

Nota: If `\d` is called without any arguments, it is equivalent to `\dtvs` which will show a list of all tables, views, and sequences. This is purely a convenience measure.

`\da [pattern]`

Lists all available aggregate functions, together with the data type they operate on. If *pattern* (a regular expression) is specified, only matching aggregates are shown.

`\dd [object]`

Shows the descriptions of *object* (which can be a regular expression), or of all objects if no argument is given. (“Object” covers aggregates, functions, operators, types, relations (tables, views, indices, sequences, large objects), rules, and triggers.) For example:

```
=> \dd version
      Object descriptions
  Name | What | Description
-----+-----+-----
version | function | PostgreSQL version string
(1 row)
```

Descriptions for objects can be generated with the **COMMENT ON SQL** command.

Nota: PostgreSQL stores the object descriptions in the `pg_description` system table.

`\df [pattern]`

Lists available functions, together with their argument and return types. If *pattern* (a regular expression) is specified, only matching functions are shown. If the form `\df+` is used, additional information about each function, including language and description is shown.

`\distvS [pattern]`

This is not the actual command name: The letters i, s, t, v, S stand for index, sequence, table, view, and system table, respectively. You can specify any or all of them in any order to obtain a listing of them, together with who the owner is.

If *pattern* is specified, it is a regular expression restricts the listing to those objects whose name matches. If one appends a “+” to the command name, each object is listed with its associated description, if any.

`\dl`

This is an alias for `\lo_list`, which shows a list of large objects.

`\do [pattern]`

Lists available operators with their operand and return types. If *pattern* is specified, only operators with that name will be shown. (Since this is a regular expression, be sure to quote all special characters in your operator name with backslashes. To prevent interpretation of the backslash as a new command, you might also wish to quote the argument.)

`\dp [pattern]`

This is an alias for `\z` which was included for its greater mnemonic value (“display permissions”).

`\dT [pattern]`

Lists all data types or only those that match *pattern*. The command form `\dT+` shows extra information.

`\edit (or \e) [filename]`

If *filename* is specified, the file is edited and after the editor exits its content is copied back to the query buffer. If no argument is given, the current query buffer is copied to a temporary file which is then edited in the same fashion.

The new query buffer is then re-parsed according to the normal rules of `psql`, where the whole buffer is treated as a single line. (Thus you cannot make “scripts” this way, use `\i` for that.) This means also that if the query ends with (or rather contains) a semicolon, it is immediately executed. In other cases it will merely wait in the query buffer.

Sugerencia: `psql` searches the environment variables `PSQL_EDITOR`, `EDITOR`, and `VISUAL` (in that order) for an editor to use. If all of them are unset, `/bin/vi` is run.

`\echo text [...]`

Prints the arguments to the standard output, separated by one space and followed by a newline. This can be useful to intersperse information in the output of scripts. For example:

```
=> \echo `date`
Tue Oct 26 21:40:57 CEST 1999
```

If the first argument is an unquoted `-n` the trailing newline is not written.

Sugerencia: If you use the `\o` command to redirect your query output you may wish to use `\qecho` instead of this command.

`\encoding [encoding]`

Sets the client encoding, if you are using multibyte encodings. Without an argument, this command shows the current encoding.

`\f [string]`

Sets the field separator for unaligned query output. The default is “|” (a “pipe” symbol). See also `\pset` for a generic way of setting output options.

`\g [{ filename | command }]`

Sends the current query input buffer to the backend and optionally saves the output in *filename* or pipes the output into a separate Unix shell to execute *command*. A bare `\g` is virtually equivalent to a semicolon. A `\g` with argument is a “one-shot” alternative to the `\o` command.

`\help (or \h) [command]`

Give syntax help on the specified SQL command. If *command* is not specified, then `psql` will list all the commands for which syntax help is available. If *command* is an asterisk (“*”), then syntax help on all SQL commands is shown.

Nota: To simplify typing, commands that consists of several words do not have to be quoted. Thus it is fine to type `\help alter table`.

`\H`

Turns on HTML query output format. If the HTML format is already on, it is switched back to the default aligned text format. This command is for compatibility and convenience, but see `\pset` about setting other output options.

`\i filename`

Reads input from the file *filename* and executes it as though it had been typed on the keyboard.

Nota: If you want to see the lines on the screen as they are read you must set the variable `ECHO` to `all`.

`\l (or \list)`

List all the databases in the server as well as their owners. Append a “+” to the command name to see any descriptions for the databases as well. If your PostgreSQL installation was compiled with multibyte encoding support, the encoding scheme of each database is shown as well.

`\lo_export loid filename`

Reads the large object with OID *loid* from the database and writes it to *filename*. Note that this is subtly different from the server function `lo_export`, which acts with the permissions of the user that the database server runs as and on the server’s file system.

Sugerencia: Use `\lo_list` to find out the large object's OID.

Nota: See the description of the `LO_TRANSACTION` variable for important information concerning all large object operations.

`\lo_import filename [comment]`

Stores the file into a PostgreSQL “large object”. Optionally, it associates the given comment with the object. Example:

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

The response indicates that the large object received object id 152801 which one ought to remember if one wants to access the object ever again. For that reason it is recommended to always associate a human-readable comment with every object. Those can then be seen with the `\lo_list` command.

Note that this command is subtly different from the server-side `lo_import` because it acts as the local user on the local file system, rather than the server's user and file system.

Nota: See the description of the `LO_TRANSACTION` variable for important information concerning all large object operations.

`\lo_list`

Shows a list of all PostgreSQL “large objects” currently stored in the database along with their owners.

`\lo_unlink loid`

Deletes the large object with OID *loid* from the database.

Sugerencia: Use `\lo_list` to find out the large object's OID.

Nota: See the description of the `LO_TRANSACTION` variable for important information concerning all large object operations.

`\o [{filename | |command}]`

Saves future query results to the file *filename* or pipe future results into a separate Unix shell to execute *command*. If no arguments are specified, the query output will be reset to `stdout`.

“Query results” includes all tables, command responses, and notices obtained from the database server, as well as output of various backslash commands that query the database (such as `\d`), but not error messages.

Sugerencia: To intersperse text output in between query results, use `\qecho`.

`\p`

Print the current query buffer to the standard output.

`\pset parameter [value]`

This command sets options affecting the output of query result tables. *parameter* describes which option is to be set. The semantics of *value* depend thereon.

Adjustable printing options are:

`format`

Sets the output format to one of `unaligned`, `aligned`, `html`, or `latex`. Unique abbreviations are allowed. (That would mean one letter is enough.)

“Unaligned” writes all fields of a tuple on a line, separated by the currently active field separator. This is intended to create output that might be intended to be read in by other programs (tab-separated, comma-separated). “Aligned” mode is the standard, human-readable, nicely formatted text output that is default. The “HTML” and “LaTeX” modes put out tables that are intended to be included in documents using the respective mark-up language. They are not complete documents! (This might not be so dramatic in HTML, but in LaTeX you must have a complete document wrapper.)

`border`

The second argument must be a number. In general, the higher the number the more borders and lines the tables will have, but this depends on the particular format. In HTML mode, this will translate directly into the `border=...` attribute, in the others only values 0 (no border), 1 (internal dividing lines), and 2 (table frame) make sense.

`expanded (or x)`

Toggles between regular and expanded format. When expanded format is enabled, all output has two columns with the field name on the left and the data on the right. This mode is useful if the data wouldn’t fit on the screen in the normal “horizontal” mode.

Expanded mode is support by all four output modes.

`null`

The second argument is a string that should be printed whenever a field is null. The default is not to print anything, which can easily be mistaken for, say, an empty string. Thus, one might choose to write `\pset null "(null)"`.

fieldsep

Specifies the field separator to be used in unaligned output mode. That way one can create, for example, tab- or comma-separated output, which other programs might prefer. To set a tab as field separator, type `\pset fieldsep "\t"`. The default field separator is `"|"` (a “pipe” symbol).

recordsep

Specifies the record (line) separator to use in unaligned output mode. The default is a newline character.

tuples_only (or t)

Toggles between tuples only and full display. Full display may show extra information such as column headers, titles, and various footers. In tuples only mode, only actual table data is shown.

title [text]

Sets the table title for any subsequently printed tables. This can be used to give your output descriptive tags. If no argument is given, the title is unset.

Nota: This formerly only affected HTML mode. You can now set titles in any output format.

tableattr (or T) [text]

Allows you to specify any attributes to be places inside the HTML table tag. This could for example be `cellpadding` or `bgcolor`. Note that you probably don’t want to specify `border` here, as that is already taken care of by `\pset border`.

pager

Toggles the list of a pager to do table output. If the environment variable `PAGER` is set, the output is piped to the specified program. Otherwise `more` is used.

In any case, `psql` only uses the pager if it seems appropriate. That means among other things that the output is to a terminal and that the table would normally not fit on the screen. Because of the modular nature of the printing routines it is not always possible to predict the number of lines that will actually be printed. For that reason `psql` might not appear very discriminating about when to use the pager and when not to.

Illustrations on how these different formats look can be seen in the *Examples* section.

Sugerencia: There are various shortcut commands for `\pset`. See `\a`, `\C`, `\H`, `\t`, `\T`, and `\x`.

Nota: It is an error to call `\pset` without arguments. In the future this call might show the current status of all printing options.

`\q`

Quit the `psql` program.

`\qecho text [...]`

This command is identical to `\echo` except that all output will be written to the query output channel, as set by `\o`.

`\r`

Resets (clears) the query buffer.

`\s [filename]`

Print or save the command line history to *filename*. If *filename* is omitted, the history is written to the standard output. This option is only available if `psql` is configured to use the GNU history library.

Nota: As of `psql` version 7.0 it is no longer necessary, in fact, to save the command history as that will be done automatically on program termination. The history is then also automatically loaded every time `psql` starts up.

`\set [name [value [...]]]`

Sets the internal variable *name* to *value* or, if more than one value is given, to the concatenation of all of them. If no second argument is given, the variable is just set with not value. To unset a variable, use the `\unset` command.

Valid variable names can contain characters, digits, and underscores. See the section about `psql` variables for details.

Although you are welcome to set any variable to anything you want to, `psql` treats several variables special. They are documented in the section about variables.

Nota: This command is totally separate from the SQL command `SET`.

`\t`

Toggles the display of output column name headings and row count footer. This command is equivalent to `\pset tuples_only` and is provided for convenience.

`\T table_options`

Allows you to specify options to be placed within the table tag in HTML tabular output mode. This command is equivalent to `\pset tableattr table_options`.

`\w {filename | /command}`

Outputs the current query buffer to the file *filename* or pipes it to the Unix command *command*.

`\x`

Toggles extended row format mode. As such it is equivalent to `\pset expanded`.

`\z [pattern]`

Produces a list of all tables in the database with their appropriate access permissions listed. If an argument is given it is taken as a regular expression which limits the listing to those tables which match it.

```
test=> \z
Access permissions for database "test"
Relation | Access permissions
-----+-----
my_table | {"=r","joe=arwR","group staff=ar"}
(1 row )
```

Read this as follows:

- `"=r"`: PUBLIC has read (**SELECT**) permission on the table.
- `"joe=arwR"`: User joe has read, write (**UPDATE**, **DELETE**), "append" (**INSERT**) permissions, and permission to create rules on the table.
- `"group staff=ar"`: Group staff has **SELECT** and **INSERT** permission.

The commands *GRANT* and *REVOKE* are used to set access permissions.

`\! [command]`

Escapes to a separate Unix shell or executes the Unix command *command*. The arguments are not further interpreted, the shell will see them as is.

`\?`

Get help information about the slash ("`\`") commands.

Command-line Options

If so configured, `psql` understands both standard Unix short options, and GNU-style long options. The latter are not available on all systems.

`-a, --echo-all`

Print all the lines to the screen as they are read. This is more useful for script processing rather than interactive mode. This is equivalent to setting the variable `ECHO` to `all`.

`-A, --no-align`

Switches to unaligned output mode. (The default output mode is otherwise aligned.)

-c, -command *query*

Specifies that `psql` is to execute one query string, *query*, and then exit. This is useful in shell scripts.

query must be either a query string that is completely parseable by the backend (i.e., it contains no `psql` specific features), or it is a single backslash command. Thus you cannot mix SQL and `psql` meta-commands. To achieve this you could pipe the string into `psql`, like so: `echo "\x \ select * from foo;" | psql`.

-d, -dbname *dbname*

Specifies the name of the database to connect to. This is equivalent to specifying *dbname* as the first non-option argument on the command line.

-e, -echo-queries

Show all queries that are sent to the backend. This is equivalent to setting the variable `ECHO` to `queries`.

-E, -echo-hidden

Echos the actual queries generated by `\d` and other backslash commands. You can use this if you wish to include similar functionality into your own programs. This is equivalent to setting the variable `ECHO_HIDDEN` from within `psql`.

-f, -file *filename*

Use the file *filename* as the source of queries instead of reading queries interactively. After the file is processed, `psql` terminates. This in many ways equivalent to the internal command `\i`.

Using this option is subtly different from writing `psql < filename`. In general, both will do what you expect, but using `-f` enables some nice features such as error messages with line numbers. There is also a slight chance that using this option will reduce the startup overhead. On the other hand, the variant using the shell's input redirection is (in theory) guaranteed to yield exactly the same output that you would have gotten had you entered everything by hand.

-F, -field-separator *separator*

Use *separator* as the field separator. This is equivalent to `\pset fieldsep` or `\f`.

-h, -host *hostname*

Specifies the host name of the machine on which the `postmaster` is running. Without this option, communication is performed using local Unix domain sockets.

-H, -html

Turns on HTML tabular output. This is equivalent to `\pset format html` or the `\H` command.

-l, -list

Lists all available databases, then exits. Other non-connection options are ignored. This is similar to the internal command `\list`.

-o, -output *filename*

Put all query output into file *filename*. This is equivalent to the command `\o`.

-p, -port *port*

Specifies the TCP/IP port or, by omission, the local Unix domain socket file extension on which the `postmaster` is listening for connections. Defaults to the value of the `PGPORT` environment variable or, if not set, to the port specified at compile time, usually 5432.

-P, -pset *assignment*

Allows you to specify printing options in the style of `\pset` on the command line. Note that here you have to separate name and value with an equal sign instead of a space. Thus to set the output format to LaTeX, you could write `-P format=latex`.

-q

Specifies that `psql` should do its work quietly. By default, it prints welcome messages and various informational output. If this option is used, none of this happens. This is useful with the `-c` option. Within `psql` you can also set the `QUIET` variable to achieve the same effect.

-R, -record-separator *separator*

Use *separator* as the record separator. This is equivalent to the `\pset record-sep` command.

-s, -single-step

Run in single-step mode. That means the user is prompted before each query is sent to the backend, with the option to cancel execution as well. Use this to debug scripts.

-S, -single-line

Runs in single-line mode where a newline terminates a query, like a semicolon would do.

Nota: This mode is provided for those who insist on it, but you are not necessarily encouraged to use it. In particular, if you mix SQL and meta-commands on a line the order of execution might not always be clear to the inexperienced user.

-t, -tuples-only

Turn off printing of column names and result row count footers, etc. It is completely equivalent to the `\t`.

-T, -table-attr *table_options*

Allows you to specify options to be placed within the HTML table tag. See `\pset` for details.

-u

Makes `psql` prompt for the user name and password before connecting to the database.

This option is deprecated, as it is conceptually flawed. (Prompting for a non-default user name and prompting for a password because the backend requires it are really two different things.) You are encouraged to look at the `-U` and `-w` options instead.

`-U, -username username`

Connects to the database as the user *username* instead of the default. (You must have permission to do so, of course.)

`-v, -variable, -set assignment`

Performs a variable assignment, like the `\set` internal command. Note that you must separate name and value, if any, by an equal sign on the command line. To unset a variable, leave off the equal sign. These assignments are done during a very early state of startup, so variables reserved for internal purposes might get overwritten again.

`-V, -version`

Shows the `psql` version.

`-W, -password`

Requests that `psql` should prompt for a password before connecting to a database. This will remain set for the entire session, even if you change the database connection with the meta-command `\connect`.

As of version 7.0, `psql` automatically issues a password prompt whenever the backend requests password authentication. Because this is currently based on a “hack”, the automatic recognition might mysteriously fail, hence this option to force a prompt. If no password prompt is issued and the backend requires password authentication the connection attempt will fail.

`-x, -expanded`

Turns on extended row format mode. This is equivalent to the command `\x`.

`-, -help`

Shows help about `psql` command line arguments.

Advanced features

Variables

`psql` provides variable substitution features similar to common Unix command shells. This feature is new and not very sophisticated, yet, but there are plans to expand it in the future. Variables are simply name/value pairs, where the value can be any string of any length. To set variables, use the `psql` meta-command `\set`:

```
testdb=> \set foo bar
```

sets the variable “foo” to the value “bar”. To retrieve the content of the variable, precede the name with a colon and use it as the argument of any slash command:

```
testdb=> \echo :foo
bar
```

Nota: The arguments of `\set` are subject to the same substitution rules as with other commands. Thus you can construct interesting references such as `\set :foo 'something'` and get “soft links” or “variable variables” of Perl or PHP fame, respectively. Unfortunately (or fortunately?), there is not way to do anything useful with these constructs. On the other hand, `\set bar :foo` is a perfectly valid way to copy a variable.

If you call `\set` without a second argument, the variable is simply set, but has no value. To unset (or delete) a variable, use the command `\unset`.

`psql`’s internal variable names can consist of letters, numbers, and underscores in any order and any number of them. A number of regular variables are treated specially by `psql`. They indicate certain option settings that can be changed at runtime by altering the value of the variable or represent some state of the application. Although you can use these variables for any other purpose, this is not recommended, as the program behavior might grow really strange really quickly. By convention, all specially treated variables consist of all upper-case letters (and possibly numbers and underscores). To ensure maximum compatibility in the future, avoid such variables. A list of all specially treated variables follows.

DBNAME

The name of the database you are currently connected to. This is set everytime you connect to a database (including program startup), but can be unset.

ECHO

If set to “all”, all lines entered or from a script are written to the standard output before they are parsed or executed. To specify this on program startup, use the switch `-a`. If set to “queries”, `psql` merely prints all queries as they are sent to the backend. The option for this is `-e`.

ECHO_HIDDEN

When this variable is set and a backslash command queries the database, the query is first shown. This way you can study the PostgreSQL internals and provide similar functionality in your own programs. If you set the variable to the value “noexec”, the queries are just shown but are not actually sent to the backend and executed.

ENCODING

The current client multibyte encoding. If you are not set up to use multibyte characters, this variable will always contain “SQL_ASCII”.

HISTCONTROL

If this variable is set to `ignorespace`, lines which begin with a space are not entered into the history list. If set to a value of `ignoredups`, lines matching the previous history line are not entered. A value of `ignoreboth` combines the two options. If unset, or if set to any other value than those above, all lines read in interactive mode are saved on the history list.

Nota: This feature was shamelessly plagiarized from `bash`.

HISTSIZE

The number of commands to store in the command history. The default value is 500.

Nota: This feature was shamelessly plagiarized from `bash`.

HOST

The database server host you are currently connected to. This is set everytime you connect to a database (including program startup), but can be unset.

IGNOREEOF

If unset, sending an EOF character (usually Control-D) to an interactive session of `psql` will terminate the application. If set to a numeric value, that many EOF characters are ignored before the application terminates. If the variable is set but has no numeric value, the default is 10.

Nota: This feature was shamelessly plagiarized from `bash`.

LASTOID

The value of the last affected oid, as returned from an **INSERT** or **lo_insert** command. This variable is only guaranteed to be valid until after the result of the next SQL command has been displayed.

LO_TRANSACTION

If you use the PostgreSQL large object interface to specially store data that does not fit into one tuple, all the operations must be contained in a transaction block. (See the documentation of the large object interface for more information.) Since `psql` has no way to keep track if you already have a transaction in progress when you call one of its internal commands **\lo_export**, **\lo_import**, **\lo_unlink** it must take some arbitrary action. This action could either be to roll back any transaction that might already be in progress, or to commit any such transaction, or to do nothing at all. In the latter case you must provide you own **BEGIN TRANSACTION/COMMIT** block or the results will be unpredictable (usually resulting in the desired action not being performed in any case).

To choose what you want to do you set this variable to one of “rollback”, “commit”, or “nothing”. The default is to roll back the transaction. If you just want to load one or a few objects this is fine. However, if you intend to transfer many large objects, it might be advisable to provide one explicit transaction block around all commands.

ON_ERROR_STOP

By default, if non-interactive scripts encounter an error, such as a malformed SQL query or internal meta-command, processing continues. This has been the traditional behaviour of `psql` but it is sometimes not desirable. If this variable is set, script processing will immediately terminate. If the script was called from another script it will terminate in the same fashion. If the outermost script was not called from an interactive `psql` session but rather using the `-f` option, `psql` will return error code 3, to distinguish this case from fatal error conditions (error code 1).

PORT

The database server port you are currently connected to. This is set everytime you connect to a database (including program startup), but can be unset.

PROMPT1, PROMPT2, PROMPT3

These specify what the prompt `psql` issues is supposed to look like. See “*Prompting*” below.

QUIET

This variable is equivalent to the command line option `-q`. It is probably not too useful in interactive mode.

SINGLELINE

This variable is set by the command line options `-s`. You can unset or reset it at run time.

SINGLESTEP

This variable is equivalent to the command line option `-s`.

USER

The database user you are currently connected as. This is set everytime you connect to a database (including program startup), but can be unset.

SQL Interpolation

An additional useful feature of `psql` variables is that you can substitute (“interpolate”) them into regular SQL statements. The syntax for this is again to prepend the variable name with a colon (:).

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

would then query the table `my_table`. The value of the variable is copied literally, so it can even contain unbalanced quotes or backslash commands. You must make sure that it makes sense where you put it. Variable interpolation will not be performed into quoted SQL entities.

A popular application of this facility is to refer to the last inserted OID in subsequent statement to build a foreign key scenario. Another possible use of this mechanism is to copy the contents of a file into a field. First load the file into a variable and then proceed as above.


```
testdb=> \set content '\" `cat my_file.txt` '\"
testdb=> INSERT INTO my_table VALUES (:content);
```

One possible problem with this approach is that `my_file.txt` might contain single quotes. These need to be escaped so that they don't cause a syntax error when the third line is processed. This could be done with the program `sed`:

```
testdb=> \set content `sed -e "s/'/\\'/g" < my_file.txt`
```

Observe the correct number of backslashes (6)! You can resolve it this way: After `psql` has parsed this line, it passes `sed -e "s/'/\\'/g" < my_file.txt` to the shell. The shell will do its own thing inside the double quotes and execute `sed` with the arguments `-e` and `s/'/\\'/g`. When `sed` parses this it will replace the two backslashes with a single one and then do the substitution. Perhaps at one point you thought it was great that all Unix commands use the same escape character. And this is ignoring the fact that you might have to escape all backslashes as well because SQL text constants are also subject to certain interpretations. In that case you might be better off preparing the file externally.

Since colons may legally appear in queries, the following rule applies: If the variable is not set, the character sequence "colon+name" is not changed. In any case you can escape a colon with a backslash to protect it from interpretation. (The colon syntax for variables is standard SQL for embedded query languages, such as `ecpg`. The colon syntax for array slices and type casts are PostgreSQL extensions, hence the conflict.)

Prompting

The prompts `psql` issues can be customized to your preference. The three variables `PROMPT1`, `PROMPT2`, and `PROMPT3` contain strings and special escape sequences that describe the appearance of the prompt. Prompt 1 is the normal prompt that is issued when `psql` requests a new query. Prompt 2 is issued when more input is expected during query input because the query was not terminated with a semicolon or a quote was not closed. Prompt 3 is issued when you run an SQL **COPY** command and you are expected to type in the tuples on the terminal.

The value of the respective prompt variable is printed literally, except where a percent sign ("%") is encountered. Depending on the next character, certain other text is substituted instead. Defined substitutions are:

%M

The hostname of the database server (or "." if Unix domain socket).

%m

The hostname of the database server truncated after the first dot.

%>

The port number at which the database server is listening.

%n

The username you are connected as (not your local system user name).

%/

The name of the current database.

`%~`

Like `%/`, but the output is “~” (tilde) if the database is your default database.

`%#`

If the current user is a database superuser, then a “#”, otherwise a “>”.

`%R`

In prompt 1 normally “=”, but “^” if in single-line mode, and “!” if the session is disconnected from the database (which can happen if `\connect` fails). In prompt 2 the sequence is replaced by “-”, “*”, a single quote, or a double quote, depending on whether `psql` expects more input because the query wasn’t terminated yet, because you are inside a `/* ... */` comment, or because you are inside a quote. In prompt 3 the sequence doesn’t resolve to anything.

`%digits`

If *digits* starts with `0x` the rest of the characters are interpreted at a hexadecimal digit and the character with the corresponding code is substituted. If the first digit is `0` the characters are interpreted as on octal number and the corresponding character is substituted. Otherwise a decimal number is assumed.

`%:name:`

The value of the `psql`, variable *name*. See the section “Variables” for details.

`%`command``

The output of *command*, similar to ordinary “back-tick” substitution.

To insert a percent sign into your prompt, write `%%`. The default prompts are equivalent to `'%/%R%# '` for prompts 1 and 2, and `'>> '` for prompt 3.

Nota: This feature was shamelessly plagiarized from `tcsh`.

Miscellaneous

`psql` returns 0 to the shell if it finished normally, 1 if a fatal error of its own (out of memory, file not found) occurs, 2 if the connection to the backend went bad and the session is not interactive, and 3 if an error occurred in a script and the variable `ON_ERROR_STOP` was set.

Before starting up in interactive mode, `psql` attempts to read and execute commands from the file `$HOME/.psqlrc`. It could be used to set up the client or the server to taste (using the `\set` and `SET` commands).

GNU readline

`psql` supports the readline and history libraries for convenient line editing and retrieval. The command history is stored in a file named `.psql_history` in your home directory and is reloaded when `psql` starts up. Tab-completion is also supported, although the completion logic makes no claim to be an SQL parser. When available, `psql` is automatically built to use these features. If for some reason you do not like

the tab completion, you can turn it off by putting this in a file named `.inputrc` in your home directory:

```
$if psql
set disable-completion on
$endif
```

(This is not a `psql` but a `readline` feature. Read its documentation for further details.)

If you have the `readline` library installed but `psql` does not seem to use it, you must make sure that PostgreSQL's top-level `configure` script finds it. `configure` needs to find both the library `libreadline.a` (or `libreadline.so` on systems with shared libraries) *and* the header files `readline.h` and `history.h` (or `readline/readline.h` and `readline/history.h`) in appropriate directories. If you have the library and header files installed in an obscure place you must tell `configure` about them, for example:

```
$ ./configure --with-includes=/opt/gnu/include --with-libs=/opt/gnu/lib ...
```

Then you have to recompile `psql` (not necessarily the entire code tree).

The GNU `readline` library can be obtained from the GNU project's FTP server at <ftp://ftp.gnu.org>¹.

Examples

Nota: This section only shows a few examples specific to `psql`. If you want to learn SQL or get familiar with PostgreSQL, you might wish to read the Tutorial that is included in the distribution.

The first example shows how to spread a query over several lines of input. Notice the changing prompt.

```
testdb=> CREATE TABLE my_table (
testdb->   first integer not null default 0,
testdb->   second text
testdb-> );
CREATE
```

Now look at the table definition again:

```
testdb=> \d my_table
          Table "my_table"
  Attribute |  Type  |      Modifier
-----+-----+-----
  first    | integer | not null default 0
  second   | text    |
```

At this point you decide to change the prompt to something more interesting:

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

Let's assume you have filled the table with data and want to take a look at it:

```

peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-----+-----
      1 | one
      2 | two
      3 | three
      4 | four
(4 rows)

```

Notice how the int4 columns in right aligned while the text column in left aligned. You can make this table look differently by using the `\pset` command.

```

peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM my_table;
+-----+-----+
| first | second |
+-----+-----+
|      1 | one    |
|      2 | two    |
|      3 | three  |
|      4 | four   |
+-----+-----+
(4 rows)

```

```

peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM my_table;
first second
---  ---
      1 one
      2 two
      3 three
      4 four
(4 rows)

```

```

peter@localhost testdb=> \pset border 1
Border style is 1.
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep ","
Field separator is ",".
peter@localhost testdb=> \pset tuples_only
Showing only tuples.
peter@localhost testdb=> SELECT second, first FROM my_table;
one,1
two,2
three,3
four,4

```

Alternatively, use the short commands:

```

peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first | 1
second | one
-[ RECORD 2 ]-
first | 2

```

```

second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four

```

Appendix

Bugs and Issues

- In some earlier life `psql` allowed the first argument to start directly after the (single-letter) command. For compatibility this is still supported to some extent but I am not going to explain the details here as this use is discouraged. But if you get strange messages, keep this in mind. For example

```

testdb=> \foo
Field separator is "oo".

```

is perhaps not what one would expect.

- `psql` only works smoothly with servers of the same version. That does not mean other combinations will fail outright, but subtle and not-so-subtle problems might come up.

Notas

1. <ftp://ftp.gnu.org>

pgtclsh

Nombre

`pgtclsh` — Postgres Cliente para shell TCL

Synopsis

```
pgtclsh [ base_de_datos ]
```

Entrada

base_de_datos

El nombre de una base de datos existente a la que se quiera tener acceso.

Salida**Descripción**

`pgtclsh` proporciona un interfaz al shell TCL para Postgres.

Otra manera de tener acceso a Postgres por medio de tcl es usando *pgtksh* o *pgaccess*.

pgtksh**Nombre**

`pgtksh` — Postgres Shell gráfico para TCL/TK

Synopsis

```
pgtksh [ base_de_datos ]
```

Entrada

base_de_datos

El nombre de una base datos existente a la que se quiera tener acceso.

Salida**Descripción**

`pgtksh` proporciona un interfaz gráfico al shell TCL/TK para Postgres.

Otra manera de tener acceso a Postgres por medio de TCL es usando *pgtclsh* o *pgaccess*.

vacuumdb**Nombre**

`vacuumdb` — Limpia y analiza una base de datos PostgreSQL

Synopsis

```
vacuumdb [ opciones de conexión ] [ -analyze | -z ] [ -alldb | -a ] [ -
verbose | -v ]
        [ -table 'tabla [ ( columna [,...] ) ]' ] [ [-d] nombre_bd ]
```

Entradas

`vacuumdb` acepta los siguientes argumentos en la línea de comandos:

`[-d, -dbname] nombre_bd`

Especifica el nombre de la base de datos que de be ser limpiada o analizada.

`-z, -analyze`

Calcula estadísticas sobre la base de datos para ser usadas por el optimizador.

`-a, -alldb`

Limpia todas las bases de datos.

`-v, -verbose`

Imprime información detallada durante el proceso.

`-t, -table tabla [(columna [,...])]`

Limpia o analiza únicamente la *tabla* indicada. Se pueden especificar nombres de columnas únicamente cuando se usa la opción `-analyze`.

Sugerencia: Si usted da el nombre de columnas que deben ser analizadas, probablemente tendrá que usar caracteres de escape de la shell para los paréntesis.

`vacuumdb` también acepta los siguientes argumentos de línea de comandos, para parámetros de conexión:

`-h, -host anfitrión`

Especifica el nombre de la máquina anfitriona en la cual se está ejecutando el `postmaster`.

`-p, -port puerta`

Especifica la puerta de Internet TCP/IP o el fichero Unix de extensión de dominio local de conexión en que el `postmaster` recibe conexiones.

`-U, -username nombre`

Nombre de usuario que se debe usar para conectar.

`-W, -password`

Obliga el pedido de contraseña antes de ejecutar.

`-e, -echo`

Escribe una copia de los comandos que `vacuumdb` genera y envía al servidor.

`-q, -quiet`

No muestre la respuesta.

Mensajes de Resultados

`VACUUM`

Todo corrió bien.

`vacuumdb: La limpieza falló.`

Algo ha fallado. `vacuumdb` es apenas un guión de interfaz. Consulte *VACUUM* y *psql* para un discusión detallada de los mensajes de error y posibles problemas.

Descripción

`vacuumdb` es un utilitario para limpiar una base de datos PostgreSQL. `vacuumdb` también produce estadísticas internas usadas por el optimizador de búsquedas de Postgres.

`vacuumdb` es un guión que envuelve al comando *VACUUM* de PostgreSQL, por medio del terminal interactivo *psql*. No existe diferencia efectiva entre la limpieza de bases de datos usando este u otros métodos. El guión deberá lograr encontrar a *psql* y deberá existir un servidor de bases de datos en ejecución en el anfitrión usado. Serán usadas cualquier configuración y variables de estado de *psql* y de la librería de interfaz *libpq*.

Uso

Para limpiar la base de datos prueba:

```
$ vacuumdb prueba
```

Para analizar para el optimizador una base de datos llamada *bdgrande*:

```
$ vacuumdb -analyze bdgrande
```

Para analizar para el optimizador una única columna *cual* en la tabla *tal* de una base de datos llamada *xyzzy*:

```
$ vacuumdb -analyze -verbose -table 'tal(cual)' xyzzy
```


Notas

1. <http://www.pgadmin.freemove.co.uk>
1. <ftp://ftp.gnu.org>

Capítulo 16. Aplicaciones del sistema

Esta es la información de referencia para los servidores y utilidades de soporte de Postgres.

initdb

Nombre

initdb — Crea una nueva instalación de la base de datos de PostgreSQL

Synopsis

```
initdb [ -pgdata|-D dbdir ]  
      [ -sysid|-i sysid ]  
      [ -pwprompt|-W ]  
      [ -encoding|-E encoding ]  
      [ -pglib|-L libdir ]  
      [ -noclean | -n ] [ -debug | -d ] [ -template | -t ]
```

Inputs

-pgdata=dbdir
-D dbdir
PGDATA

Esta opción especifica en que parte del sistema de archivos será almacenada la base de datos. Ésta es la única información requerida por el `initdb`, pero podemos omitirla estableciendo la variable de entorno `PGDATA` lo que puede ser conveniente ya que el servidor de la base de datos (`postmaster`) puede encontrar el directorio de la base de datos más adelante a través de la misma variable.

-sysid=sysid
-i sysid

Selecciona el id del sistema para el super usuario (`root`) de la base de datos. Por omisión apunta al id de aquel usuario que este ejecutando `initdb`. Realmente no es importante cuál sea el id del sistema para el super usuario, ya que uno podría elegir comenzar la numeración con cualquier número como 0 o 1.

-pwprompt
-W

Ocasiona que el `initdb` pregunte por el password del super usuario (`root`) de la base de datos. Si uno no planea usar la autenticación a través de passwords, entonces realmente no es importante. De cualquier manera uno no podrá utilizar la autenticación a través de passwords hasta que haya establecido un password.

`-encoding=encoding`
`-E encoding`

Selecciona la codificación multibyte para la base de datos modelo (o plantilla). De hecho esta también será la codificación por defecto para cualquier base de datos que uno cree más adelante, a menos que usted la cambie. Para utilizar la característica de codificación multibyte, se debe especificar durante el tiempo de construcción (creación de la BD), en cuyo caso uno también selecciona el valor por defecto para esta opción.

Otros parámetros utilizados menos comúnmente están también disponibles:

`-pglib=libdir`
`-l libdir`

`initdb` necesita algunos archivos de entrada para poder inicializar la base de datos. Esta opción indica dónde encontrarlos. Normalmente, uno no tiene que preocuparse por esto puesto que el `initdb` conoce los esquemas de instalación más comunes y encontrará, normalmente, los archivos por sí mismo. Se le dirá si usted necesita especificar su ubicación explícitamente. Si sucede esto, uno de los ficheros se llama `global.bki.source` y está instalado tradicionalmente junto con los otros archivos en el directorio de bibliotecas (por ejemplo, `/usr/local/pgsql/lib`) `/usr/local/pgsql/lib`).

`-template`
`-t`

Replace the `template1` Substituye la base de datos `template1` en un sistema de base de datos existente, y no toca otra cosa. Esto es útil cuando se necesita actualizar el `template1` de la base de datos usando el `initdb` de una versión más nueva de PostgreSQL, o cuando el `template1` de la base de datos se ha corrompido por algún problema del sistema. Normalmente el contenido del `template1` se mantendrá constante a través de la vida del sistema de base de datos. No se puede destruir cualquier otra cosa ejecutando el `initdb` con la opción `-template`.

`-noclean`
`-n`

Por defecto, cuando `initdb` determina que un error evita que se cree totalmente el sistema de base de datos, remueve cualquier archivo que pudo haber creado, antes de determinar que no puede acabar el trabajo. Esta opción inhibe "tidying-up" y es por lo tanto, útil para depurar.

`-debug`
`-d`

Imprime la salida de depuración de la "carga inicial backend" y algunos otros mensajes de poco interés para el público en general. La "carga inicial backend" es la aplicación que el `initdb` usa para crear las tablas del catálogo. Esta opción genera una enorme cantidad de salida.

Salidas

`initdb` creará los ficheros en el área de datos especificada que son las tablas del sistema y el marco de trabajo para una instalación completa.

Descripción

`initdb` crea un nuevo sistema de base de datos de PostgreSQL database system. Un sistema de base de datos es una colección de bases de datos que son todas administradas por el mismo usuario de UNIX y manejadas por un solo postmaster

Crear un sistema de base de datos consiste en crear los directorios en los cuales los datos de la base de datos serán almacenados. generar las tablas de catálogo compartidas (son tablas que no pertenecen a ninguna base de datos determinada). crear el `template1` de la base de datos. Cuando usted crea una nueva base de datos, todo el `template1` de la base de datos se copia. Contiene las tablas de catálogo llenas para cosas como los tipos interconstruidos

No se debe ejecutar el `initdb` como root. Esto se debe ya que uno no puede ejecutar el servidor de la base de datos ni siquiera como root, pero el servidor necesita tener acceso a los archivos que `initdb` crea. Además, durante la fase de la inicialización, cuando no hay usuarios y ningún control de acceso instalado, postgres solamente se conectará con el nombre de usuario actual de UNIX, así que uno debe iniciar una sesión bajo la cuenta que poseerá el proceso del servidor.

Aunque `initdb` procurará crear el directorio de datos respectivo, lo cierto es que no tendrá el permiso para hacerlo. Por lo tanto, es una buena idea crear el directorio de datos antes de ejecutar `initdb` y entregar la propiedad de él al super usuario de la base de datos.

`initlocation`

Nombre

`initlocation` — Crea un área de almacenamiento secundario para la base de datos de PostgreSQL

Synopsis

`initlocation directory`

Entradas

directory

¿Dónde deseas almacenar las bases de datos alternativas dentro del sistema de archivos UNIX?

Salidas

`initlocation` creará directorios en el lugar especificado .

Descripción

`initlocation` crea una nueva área de almacenamiento secundario para la base de datos PostgreSQL. Vea la discusión en *CREATE DATABASE* sobre cómo manejar y utilizar áreas de almacenamiento secundario. Si el argumento no contiene un slash (/) y es inválido como vía (path), entonces se asume que es una variable de entorno (ambiente), la cual es referenciada. Vea los ejemplos al final.

Para poder utilizar este comando usted debe haber entrado (con “su”, por ejemplo) a la base de datos como super usuario (root).

Uso

Para crear una base de datos en una posición alterna, usando una variable de entorno:

```
$ export PGDATA2=/opt/postgres/data
$ initlocation PGDATA2
$ createdb 'testdb' -D 'PGDATA2/testdb'
```

Alternativamente, si usted permite vías (paths) absolutas entonces podría escribir:

```
$ initlocation /opt/postgres/data
$ createdb 'testdb' -D '/opt/postgres/data/testdb'
```

ipcclean

Nombre

`ipcclean` — Limpia la memoria compartida y los semáforos de "backends" abortados.

Synopsis

```
ipcclean
```

Entradas

Ninguna.

Salidas

Ninguna.

Descripción

`ipcclean` limpia la memoria compartida y el espacio de semáforos de "backends" abortados, borrando todas las instancias que son propiedad del usuario `postgres`. Solamente el DBA (DataBase Administrator - Administrador de la Base de Datos) debe ejecutar este programa ya que puede causar algún tipo de comportamiento extraño (es decir, caídas) si se ejecuta durante una ejecución multiusuario. Este programa se debe ejecutar si aparecen mensajes como por ejemplo `semget: No queda espacio libre en el dispositivo al ejecutar el proceso postmaster o el servidor "backend"`.

Si se ejecuta esta orden mientras el proceso `postmaster` está corriendo, se eliminará la memoria compartida y los semáforos almacenados por el `postmaster`. Esto puede provocar el fallo general de los servidores "backend" iniciados por ese `postmaster`.

Este script es un "hack", pero en los muchos años desde que fué escrito, nadie ha venido con una solución igualmente eficaz y portable. Cualquier sugerencia será bienvenida.

El script hace una suposición sobre el formato de salida de la utilidad `ipcs`, suposición que puede no ser cierta en todos los sistemas operativos, por lo que puede fallar en su SO particular.

`pg_passwd`

Nombre

`pg_passwd` — manipula el fichero plano de passwords.

Synopsis

`pg_passwd filename`

Descripción

`pg_passwd` es una herramienta para manipular la funcionalidad del fichero plano de passwords de Postgres. Este estilo de autenticación de passwords no se *requiere* en una instalación, pero es uno de los diversos mecanismos utilizados en la seguridad.

Especifique el archivo de passwords en el mismo estilo que autenticación `ident` en: `$PGDATA/pg_hba.conf`:

```
host    unv          133.65.96.250    255.255.255.255 password passwd
```

Donde la línea anterior permite el acceso desde 133,65,96,250 usando los passwords listados en `$PGDATA/passwd`. El formato del archivo de passwords sigue el formato de `/etc/passwd` y `/etc/shadow`. El primer campo es el nombre de usuario, y el segundo campo es el password cifrado. El resto es totalmente ignorado. Así las tres líneas siguientes de ejemplo especifican el mismo par de nombre de usuario y password:

```
pg_guest:/nB7.w5Auq.BY:10031:::
pg_guest:/nB7.w5Auq.BY:93001:930::/home/guest:/bin/tcsh
pg_guest:/nB7.w5Auq.BY:93001
```

Provea del fichero de passwords al comando `pg_passwd`. En el caso descrito anteriormente, después de cambiar el directorio de trabajo a PGDATA, la ejecución siguiente del comando especifica el nuevo password para `pg_guest`:

```
% pg_passwd passwd
Username: pg_guest
Password:
Re-enter password:
```

Donde la petición `Password:` y `Re-enter password:` requieren el mismo password de entrada pero no se visualizarán en la terminal. El archivo original de passwords se renombra como `passwd.bk`.

`psql` utiliza la opción `-u` para invocar este estilo de autenticación.

Las líneas siguientes muestran ejemplos de uso de la opción:

```
% psql -h hyalos -u unv
Username: pg_guest
Password:
Bienvenido al monitor interactivo de PostgreSQL:
  Lea por favor el archivo COPYRIGHT para los términos de derechos de au-
  tor del tipo de PostgreSQL.
  Escriba \? para la ayuda en comandos slash (/)
  Escriba \q para salir
  Escriba \g o terminar con punto y coma para ejecutar la consulta
  Usted está conectado actualmente con la base de datos: unv
unv =>
```

La autenticación de Perl5 utiliza el nuevo estilo de `Pg.pm` como esto:

```
$conn = Pg::connectdb("host=hyalos dbname=unv
                      user=pg_guest password=xxxxxxx");
```

Para más detalles, refiérase a `src/interfaces/perl5/Pg.pm`.

La autenticación `Pg{tcl,tk}sh` utiliza el comando `pg_connect` con la opción `-conninfo` por lo tanto:

```
% set conn [pg_connect -conninfo \\\
                      "host=hyalos dbname=unv \\\
                      user=pg_guest password=xxxxxxx "]
```

Se pueden enumerar todas las claves para la opción ejecutando el comando siguiente:

```
% puts [ pg_conndefaults]
```


pg_upgrade

Nombre

`pg_upgrade` — permite la actualización de una versión anterior sin tener que volver a recargar los datos.

Synopsis

```
pg_upgrade [ -f filename ] old_data_dir
```

Descripción

`pg_upgrade` es una utilidad para actualizar una versión anterior de PostgreSQL sin la necesidad de recargar todos los datos. No todas las transiciones de versiones de Postgres se pueden manejar de esta manera. Verifique las notas de la versión para saber si hay detalles en su instalación.

Actualización de Postgres with `pg_upgrade`

1. Respalde su directorio de datos existente, preferiblemente haciendo un vaciado completo con el `pg_dumpall`.

2. Luego realice:

```
% pg_dumpall -s >db.out
```

para vaciar la antigua tabla de definiciones de la base de datos sin ningún dato.

3. Detenga el antiguo postmaster y todos los "backends".

4. Renombre (usando `mv`) su antiguo directorio `pgsql data/` a `data.old/`.

5. Ejecute

```
% make install
```

para instalar los nuevos binarios.

6. Ejecute `initdb` para crear una nueva base de datos `template1` que contenga las tablas del sistema para la nueva versión.

7. Inicie el nuevo postmaster. (Nota: es de suma importancia que ningún usuario se conecte a la base de datos hasta que la actualización esté completada. Quizás desee iniciar el postmaster sin la opción `-i` y/o alterar `pg_hba.conf` temporalmente.)

8. Cambie su directorio de trabajo hacia el directorio principal del `pgsql`, y ejecute:

```
% pg_upgrade -f db.out data.old
```

El programa hará algunas verificaciones para cerciorarse de que todo esta configurado correctamente, y ejecutará el script `db.out` para volver a reconstruir todas las

bases de datos y tablas que uno tenía, pero sin datos. Entonces moverá físicamente los archivos de datos que no contienen tablas del sistema y los índices desde `data.old/` hacia los subdirectorios indicados debajo de `data.old/` sustituyendo los archivos de datos vacíos creados durante la ejecución del script `db.out`.

9. Restablezca si es necesario su antiguo archivo `pg_hba.conf` para permitir conexiones a los usuarios.
10. Detenga y vuelva a iniciar el postmaster.
11. Examine *cuidadosamente* el contenido de la base de datos actualizada. Si encuentra algún problema, entonces necesitará recuperar sus datos restableciendo su respaldo completo `pg_dump`. Puede eliminar el directorio `data.old/` cuando se encuentre satisfecho con los resultados obtenidos.
12. La base de datos actualizada se encontrará en un estado no limpio. Probablemente deseará ejecutar un **VACUUM ANALYZE** antes de que comience el trabajo de producción.

postgres

Nombre

`postgres` — Ejecuta un proceso Postgres de usuario único

Synopsis

```
postgres [ dbname ]
postgres [ -B nBuffers ] [ -C ] [ -D DataDir ] [ -E ] [ -F ]
      [ -O ] [ -Q ] [ -S SortSize ] [ -d [ DebugLevel ] ] [ -e ]
      [ -o ] [ OutputFile ] [ -s ] [ -v protocol ] [ dbname ]
```

Entradas

`postgres` acepta los siguientes argumentos en la línea de comandos:

dbname

El argumento opcional *dbname* especifica el nombre de la base de datos a acceder. *dbname* toma por defecto el valor de la variable de entorno `USER`.

`-B nBuffers`

Si el motor de datos se está ejecutando bajo el `postmaster`, *nBuffers* es el número de búfers de memoria compartida que el `postmaster` tiene reservados para los procesos servidores que arranca. Si el motor de datos se ejecuta como un proceso independiente, especifica el número de búfers a reservar. Este valor es por defecto de 64 búfers, de 8k cada uno (o el valor que `BLCKSZ` tenga asignado en `config.h`)

`-C`

No mostrar el número de versión del servidor.

-D *DataDir*

Especifica el directorio a usar como raíz para el árbol de directorios de las bases de datos. Si **-D** no se especifica, el nombre del directorio de datos por defecto es el valor de la variable de entorno `PGDATA`. Si `PGDATA` no tiene un valor asignado, entonces el directorio usado es `$POSTGRESHOME/data`. Si ni la variable de entorno ni esta opción de línea de comandos están asignadas, se usa el directorio por defecto indicado durante la compilación.

-E

Muestra todas las consultas.

-F

Desactiva la ejecución automática de `fsync()` después de cada transacción. Esta opción mejora el rendimiento, pero una caída del sistema durante una transacción en curso puede provocar la pérdida de los últimos datos introducidos. Sin la llamada a `fsync()` los datos son almacenados temporalmente por el sistema operativo, y escritos en disco más tarde.

-O

Ignorar las restricciones que impiden la modificación de la estructura de las tablas de sistema. Estas tablas son típicamente las que incluyen «pg_» al principio del nombre.

-Q

Especifica el modo «silencioso».

-S *SortSize*

Especifica la cantidad de memoria a usar por ordenaciones internas y hashes antes de reordenar en ficheros temporales en disco. El valor se indica en kilobytes, y su valor por defecto es de 512 kilobytes. Nótese que para una consulta compleja, varias ordenaciones y/o hashes deben ejecutarse en paralelo, y cada una puede utilizar hasta *SortSize* kilobytes antes de empezar a poner datos en ficheros temporales.

-d [*DebugLevel*]

El argumento opcional *DebugLevel* determina el volumen de información de depuración que el servidor producirá. Si *DebugLevel* es uno, el postmaster registrará todo el tráfico de conexión y nada más. Para valores 2 y mayores, la depuración es activada en el proceso del motor de datos y el postmaster muestra más información, incluyendo su entorno y tráfico de proceso. Nótese que si no se especifica un archivo para almacenar la información de depuración la misma aparecerá en la consola del proceso padre postmaster.

-e

Esta opción controla cómo son interpretadas las fechas en la entrada y salida de la base de datos. Si la opción **-e** es incluida, las fechas pasadas a y desde los procesos de aplicación asumirán el formato «Europeo» (`DD-MM-YYYY`). En caso contrario se asume que las fechas están en formato «Americano» (`MM-DD-YYYY`). Las fechas se aceptan por el motor de datos en una amplia variedad de formatos, y para la entrada de fechas, este parámetro afecta a la interpretación de casos ambiguos. Véase *Data Types* para más información.

-o *OutputFile*

Envía los mensajes de error y la información de depuración a *OutputFile*. Si el motor de datos se ejecuta bajo el *postmaster*, los mensajes de error se envían también a la aplicación además de a *OutputFile*, pero la información de depuración se envía a la consola del *postmaster* (puesto que sólo un descriptor de archivo puede enviarse a un fichero real).

-s

Muestra información de tiempo y otras estadísticas al final de cada consulta. Esto es útil para medidas de rendimiento o para su uso en el ajuste del número de búfers.

-v *protocol*

Especifica el número del protocolo a emplear entre la aplicación y el motor de datos en esta sesión en particular.

Hay otras varias opciones que pueden especificarse, usadas principalmente con propósitos de depuración. Se listan aquí únicamente para su uso por desarrolladores de Postgres. *Se desaconseja claramente el uso de cualquiera de estas opciones.* Además, cualquiera de estas opciones puede desaparecer o cambiar en cualquier momento.

Estas opciones para casos especiales son:

-A *n | r | b | Q | fn | fP | X | fn | fP*

Esta opción genera un tremendo volumen de información.

-L

Desactiva el sistema de bloqueos.

-N

Desactiva el uso del carácter de nueva línea como un delimitador de consultas.

-f [*s | i | m | n | h*]

Prohíbe el uso de métodos particulares de escaneo y reuniones: *s* y *i* desactivan scaneos secuenciales y de índices, respectivamente, mientras *n*, *m*, y *h* lo hacen con reuniones de bucles enlazados, merge y hash.

Nota: Ni los scaneos secuenciales ni las uniones de bucles enlazados pueden ser desactivados completamente; Neither sequential scans nor nested-loop joins can be disabled completely; las opciones *-fs* y *-fn* simplemente impiden al optimizador usar estos tipos de planes si hay cualquier otra alternativa.

-i

Previene la ejecución de la consulta. En su lugar muestra el árbol del plan de ejecución.

-p *dbname*

Indica al motor de datos que ha sido iniciado por un *postmaster* y hace diferentes suposiciones sobre la gestión de los búfers, descriptores de ficheros, etc. Los parámetros posteriores a *-p* están restringidos a los considerados «seguros».

-t pa[rser] | pl[anner] | e[xecutor]

Muestra estadísticas de tiempo para cada consulta relacionadas con cada uno de los módulos principales del sistema. No puede ser usada con `-s`.

Salidas

Del infinito número de mensajes de error que pueden verse cuando se ejecuta directamente el motor de datos, probablemente los más comunes son:

`semget: No space left on device`

Si se muestra este mensaje, debería ejecutarse `ipcclean`. Una vez hecho esto pruébese a iniciar `postmaster` de nuevo. Si todavía no funciona, probablemente es necesario configurar el núcleo para emplear memoria compartida y semáforos, como se describe en las notas de instalación. Si se cuenta con un núcleo con memoria compartida particularmente pequeña y/o límites al uso de semáforos, será necesario reconfigurarlo para incrementar uno o los otros.

Sugerencia: Es posible posponer la reconfiguración del núcleo reduciendo `-B` para minimizar el uso de memoria compartida por Postgres.

Descripción

El motor de datos de Postgres puede ser ejecutado directamente desde un shell de usuario. Esto debería hacerse únicamente para tareas de depuración por el DBA, y no mientras otros motores de datos están siendo gestionados por un `postmaster` en este conjunto de bases de datos.

Algunos de los parámetros descritos aquí pueden pasarse al motor de datos a través del campo «opciones de base de datos» de una petición de conexión, y por lo tanto pueden referirse a un motor particular sin necesidad de reiniciar el `postmaster`. Esto es particularmente práctico para parámetros relacionados con la depuración.

El argumento opcional `dbname` especifica el nombre de la base de datos a acceder. `dbname` toma por defecto el valor de la variable de entorno `USER`.

Notes

Existen utilidades para resolver problemas de memoria compartida como `ipcs(1)`, `ipcrm(1)`, e `ipcclean(1)`. Ver también *postmaster*.

postmaster**Nombre**

`postmaster` — Ejecuta el servidor (backend) multiusuario de Postgres

Synopsis

```
postmaster [ -B nBuffers ] [ -D
DataDir ] [ -i ] [ -l ]
postmaster [ -B nBuffers ] [ -D
DataDir ] [ -N nBackends ] [ -S ]
        [ -d [ DebugLevel ] [ -i ] [
-l ] [ -o BackendOptions ] [ -p
port ]
postmaster [ -n | -s ] ...
```

Inputs

`postmaster` acepta los siguientes parámetros en su línea de comandos:

-B *nBuffers*

Indica el número de buffers de memoria compartida que `postmaster` asignar y administrará para los procesos del servidor que inicie. El valor predeterminado para esta opción es 64 buffers, siendo cada buffer de 8 kilobytes (o lo que sea que esté indicado en `BLCKSZ` en `config.h`).

-D *DataDir*

Especifica el directorio a usar como raíz del árbol de directorios de bases de datos. Si no se especifica `-D`, el nombre de directorio predeterminado es el valor de la variable de entorno `PGDATA`. Si `PGDATA` no está especificada, entonces se utiliza el directorio `$POSTGRESHOME/data`. Si no se especifica ni la variable de entorno ni esta opción de línea de comando, el directorio predeterminado es el utilizado al momento de la compilación.

-N *nBackends*

El máximo número de procesos en el servidor (backend) que `postmaster` tiene permitido iniciar. En la configuración predeterminada este valor está usualmente definido en 32, y puede ser fijado hasta un valor máximo de 1024 si su sistema puede soportar esa cantidad de procesos. Tanto el valor predeterminado como el máximo puede modificarse cuando se compila Postgres (vea el archivo `src/include/config.h`).

-S

Indica que el proceso de `postmaster` debe iniciarse en modo silencioso. Esto es, anulará la vinculación con la terminal del usuario (que tiene el control) e iniciará su propio grupo de proceso. Esta opción no debería utilizarse en conjunto con las opciones de depuración ya que cualquier mensaje enviado a la salida estándar y a la salida de error estándar serán descartados.

`-d [DebugLevel]`

Este argumento *DebugLevel* determina la cantidad de información de depuración que producirá el servidor. Si *DebugLevel* es uno, *postmaster* rastreará todo el tráfico de conexión y nada más. Para niveles iguales o mayores a 2 se activa la depuración y el proceso del servidor y *postmaster* muestran más información, incluyendo el entorno del servidor y tráfico de proceso. Note que si no se especifica ningún archivo para que los servidores del backend envíen su información, esta información será exhibida en la terminal de su proceso *postmaster* padre.

`-i`

Esta opción habilita las comunicaciones mediante TCP/IP o mediante el socket de dominio Internet. Sin esta opción solamente es posible la comunicación a través del socket de dominio Unix local.

`-l`

Este parámetro habilita la comunicación mediante el socket SSL. También es necesario especificar la opción `-i`. Además, debió habilitarse SSL en el momento de la compilación.

`-o BackendOptions`

Las opciones de *postgres* que se especifican en *BackendOptions* son pasadas a todos los procesos iniciados en el servidor por este *postmaster*. are passed to all backend server processes started by this *postmaster*. Si la cadena de opciones contiene espacios, entonces debe encerrársela entre comillas.

`-p port`

Especifica el puerto TCP/IP o la extensión de archivo del socket del dominio Unix local en el cual *postmaster* deberá esperar por conexiones solicitadas desde las aplicaciones del lado del cliente. El valor predeterminado es el especificado en la variable de entorno PGPORT o, si PGPORT no fue especificada, se toma como predeterminado el valor establecido cuando Postgres fue compilado (normalmente 5342). Si se especifica un puerto distinto del predeterminado, a todas las aplicaciones cliente (incluyendo *psql*) deberá especificárseles el mismo puerto ya sea mediante las opciones de línea de comando o utilizando la variable de entorno PGPORT.

Existen algunas opciones de línea de comandos disponibles para realizar depuraciones en caso de que un proceso en el servidor termine de forma anormal. Estas opciones controlan el comportamiento de *postmaster* en estas situaciones, y *ninguna de ellas está pensada para ser utilizada en situaciones normales*.

La estrategia usual para esta situación es notificar a todos los demás procesos en el servidor que deben terminar y reinicializar la memoria y semáforos compartidos. Esto es así debido a que un proceso de servidor que funcione de manera errática podría corromper alguno de estos recursos compartidos antes de terminar.

Estas opciones especiales son:

`-n`

postmaster no reinicializará las estructuras compartidas. Un programador podría luego analizarlas con el programa *shmemdoc* y examinar la memoria compartida y los estados de los semáforos.

-S

`postmaster` detendrá todos los demás procesos del servidor enviándoles la señal `SIGSTOP`, pero no hará que terminen. Esto permite a los programadores del sistema realizar vuelcos de núcleo a mano para todos los procesos del servidor.

Salidas

`semget: No space left on device`

Si aparece este mensaje, debería ejecutar el comando `ipcclean`. Una vez hecho esto, pruebe iniciar `postmaster` nuevamente. Si aun no funciona, probablemente necesite configurar el núcleo (kernel) de su sistema para que pueda utilizar memoria compartida y semáforos, tal como se describe en las notas de instalación. Si ejecuta múltiples instancias de `postmaster` en un sólo host, o tiene un kernel con muy poca memoria compartida o un límite de semáforos muy pequeño, tal vez deba reconfigurarlo su kernel para incrementar sus parámetros de memoria compartida y semáforos.

Sugerencia: Tal vez pueda posponer la reconfiguración del kernel disminuyendo lo especificado con `-B` para reducir la utilización de memoria compartida por parte de Postgres, o disminuyendo lo especificado con `-N` para reducir la cantidad de semáforos que utiliza Postgres.

`StreamServerPort: cannot bind to port`

Si se encuentra con este mensaje, debe asegurarse de que no existen otros procesos de `postmaster` ejecutándose en el momento. La manera más fácil de determinar esto es mediante el comando

```
% ps -ax | grep postmaster
```

en sistemas basados en BSD, o

```
% ps -e | grep postmast
```

en sistemas tipo System V o compatibles con POSIX como ser HP-UX.

Si está seguro de que no existen otros procesos de `postmaster` en ejecución, y aun así sigue recibiendo este error, intente especificar un puerto diferente utilizando la opción `-p`. También puede obtener este mensaje de error si finaliza `postmaster` y lo vuelve a iniciar inmediatamente utilizando el mismo número de puerto; simplemente espere unos segundos hasta que el sistema operativo cierre el puerto antes de intentar nuevamente. Finalmente, puede que obtenga este mensaje de error si especifica un número de puerto que su sistema operativo considere reservado. Por ejemplo, muchas versiones de Unix consideran que los puertos con número menor a 1024 deben ser confiables y solo permite al superusuario tener acceso a ellos.

`IpcMemoryAttach: shmat() failed: Permission denied`

Una explicación plausible es que otro usuario intentó iniciar un proceso `postmaster` en el mismo puerto el cual ha adquirido recursos compartidos y luego ha finalizado. Dado que las claves de memoria compartidas de Postgres se basan en el número de puerto asignado al proceso `postmaster`, estos conflictos

tiene más probabilidad de ocurrir si existe más de una instalación en un mismo servidor. Si no hay otros procesos `postmaster` en ejecución (vea más arriba), ejecute `ipcclean` e intente nuevamente. Si existen otros `postmaster` ejecutándose, deberá contactar a los propietarios de estos procesos para coordinar la asignación de puertos y/o la remoción de los segmentos de memoria compartida no utilizados.

Description

`postmaster` administra la comunicación entre los procesos del cliente y del servidor, así como la asignación de buffers compartidos y semáforos SysV (en máquinas que no tengan instrucciones del tipo test-and-set). `postmaster` no interactúa directamente con el usuario y debe ser iniciado como un proceso en segundo plano.

Sólo un `postmaster` debe estar ejecutándose a la vez para una instalación Postgres dada. Aquí una instalación significa un directorio de base de datos y un número de puerto de `postmaster`. Se puede ejecutar más de un `postmaster` en una misma máquina si cada uno de ellos tiene un directorio y un número de puerto diferente.

Notes

Siempre que se posible *evite* utilizar `SIGKILL` para forzar la finalización de `postmaster`. En su lugar debería utilizarse `SIGHUP`, `SIGINT`, o `SIGTERM` (la señal predeterminada para `kill(1)`). La utilización

```
% kill -KILL
```

o su forma alternativa

```
% kill -9
```

impedirá que `postmaster` pueda liberar los recursos del sistema (memoria compartida y semáforos) que poseía antes de finalizar. En cambio, si `postmaster` logra liberar los recursos en su poder, le evitará a usted tener que lidiar con los problemas de memoria compartida que se describieron anteriormente.

Existen varias utilidades para resolver problemas de memoria compartida, entre las cuales se encuentran `ipcs(1)`, `ipcrm(1)`, y `ipcclean(1)`.

Utilización

Para iniciar `postmaster` utilizando los valores predeterminados, escriba:

```
% nohup postmaster >logfile 2>&1 &
```

Este comando iniciará `postmaster` en el puerto predeterminado (5432). Esta es la manera más simple, y la más común, de iniciar `postmaster`.

Para iniciar `postmaster` con un número de puerto específico y un nombre de ejecutable:

```
% nohup postmaster -p 1234 &
```

Este comando ejecutará `postmaster` comunicándose a través del puerto 1234. Para poder conectarse a este `postmaster` utilizando `psql`, necesitará ejecutarlo del siguiente modo

```
% psql -p 1234
```

o fijar la variable de entorno `PGPORT`:

```
% setenv PGPORT 1234  
% psql
```

Capítulo 17. Portes

Este manual describe la versión 6.5 Postgres. La comunidad de desarrollo de Postgres ha compilado y probado Postgres en varias plataformas. Visita esta página web¹ para tener la última información.

Plataformas actualmente soportadas

En el momento de esta publicación, las siguientes plataformas han sido probadas:

Tabla 17-1. plataformas soportadas

OS	Procesador	Versión	Enviado	Apuntes
AIX 4.3.2	RS6000	v6.5	1999-05-26	(Andreas Zeugswetter)
BSDI	x86	v6.5	1999-05-25	(Bruce Momjian)
FreeBSD 2.2.x-4.0	x86	v6.5	1999-05-25	(Tatsuo Ishii, Marc Fournier)
DGUX 5.4R4.11	m88k	v6.3	1998-03-01	v6.4 probablemente con problemas
Digital Unix 4.0	Alpha	v6.5.3	1999-11-04	(Pedro J. Lobo)
HPUX	PA-RISC	v6.4	1998-10-25	Both 9.0x and 10.20 (Tom I Helbekkmo)
IRIX 6.5.6f	MIPS	v6.5.3	2000-02-18	MIPSPPro 7.3.1.1m; full
linux 2.0.x	Alpha	v6.5.3	1999-11-05	(Ryan Kirkpatrick)
linux 2.2.x	arm41	v6.5.3	1999-11-05	(Mark Knox)
linux 2.2.x/glibc2	x86	v6.5.3	1999-11-05	(Lamar Owens)
linux 2.0.x	MIPS	v6.4	1998-12-16	Cobalt Qube (Tatsuo Ishii)
linux 2.0.x	Sparc	v6.4	1998-10-25	(Tom Szybist)
linuxPPC 2.1.24	PPC603e	v6.4	1998-10-26	Powerbook 2400c (Tatsuo Ishii)
mklinux DR3	PPC750	v6.4	1998-09-16	PowerMac 7600 (Tatsuo Ishii)
NetBSD	arm32	v6.5	1999-04-14	(Andrew McMurtry)
NetBSD 1.3.2	x86	v6.4	1998-10-25	(Brook Milligan)
NetBSD	m68k	v6.4.2	1998-12-28	Mac SE/30 (Mr. Mutsaers)
NetBSD-current	NS32532	v6.4	1998-10-27	pequeños problemas en
NetBSD/sparc 1.3H	Sparc	v6.4	1998-10-27	(Tom I Helbekkmo)
NetBSD 1.3	VAX	v6.3	1998-03-01	(Tom I Helbekkmo)
QNX-4.25	x86	v6.5.2	1999-11-08	requiere pequeños parches
SCO OpenServer 5	x86	v6.5	1999-05-25	(Andrew Merrill)
SCO UnixWare 7	x86	v6.5	1999-05-25	(Andrew Merrill)
Solaris	x86	v6.4	1998-10-28	(Marc Fournier)
Solaris 2.6-2.7	Sparc	v6.4	1998-10-28	(Tom Szybist, Frank Fournier)
SunOS 4.1.4	Sparc	v6.3	1998-03-01	Patches submitted (Tatsuo Ishii)
SVR4	MIPS	v6.4	1998-10-28	Sin soporte para comp
Windows	x86	v6.4	1999-01-06	Librerías del lado del
Windows NT	x86	v6.5	1999-05-26	Trabajando con la librería

Plataformas listadas para v6.3.x y v6.4.x también trabajan con la v6.5, pero no hemos recibido confirmación explícita de la misma en el momento de la creación de la lista.

Nota: Para Windows NT, el porte de la parte del servidor Postgres se ha conseguido recientemente Postgres Se requiere de la librería Cygnus para compilarlo.

Plataformas no soportadas

Hay pocas plataformas con las cuales se haya intentado y se haya informado que no trabaja con la distribución estándar. Otras listadas aquí no proveen de suficientes librerías para intentarlo.

Tabla 17-2. Posiblemente Plataformas incompatibles

OS	Procesador	Versión	Enviado	Apuntes
MacOS	all	v6.x	1998-03-01	Sin librerías compatib
NextStep	x86	v6.x	1998-03-01	Sólo soporte cliente; v
SVR4 4.4	m88k	v6.2.1	1998-03-01	Confirmado con parch

Notas

1. <http://www.postgresql.org/docs/admin/ports.htm>

Capítulo 18. Opciones de Configuración

Parámetros de configuración (configure)

El conjunto de parámetros disponibles en configure se puede obtener escribiendo

```
$ ./configure -help
```

Los siguientes parámetros pueden ser de interés para los instaladores:

Nombre de directorios y ficheros:

-prefix=PREFIX	ficheros de instalación independiente de la arquitectura en PREFIX
-bindir=DIR	[/usr/local/pgsql] ejecutables de usuario en el DIR [EPREFIX/bin]
-libdir=DIR	librerías de código objeto en el DIR [EPREFIX/lib]
-includedir=DIR	ficheros de cabeceras C en el in DIR [PREFIX/include]
-mandir=DIR	documentación man en el DIR [PREFIX/man]

Características y paquetes:

-disable-FEATURE	no incluir la FEATURE (lo mismo que -enable-FEATURE=no)
-enable-FEATURE[=ARG]	incluir FEATURE [ARG=yes]
-with-PACKAGE[=ARG]	usar PACKAGE [ARG=yes]
-without-PACKAGE	no usar PACKAGE (lo mismo que -with-PACKAGE=no)

-enable y -with opciones reconocidas:

-with-template=template	usar el fichero plantilla del sistema operativo
-with-includes=incdir	ver directorio plantilla sitio donde están los ficheros cabecera para tk/tcl, etc. en el DIR
-with-libs=incdir	buscar librerías también en DIR
-with-libraries=libdir	buscar librerías también en DIR
-enable-locale	activa el soporte local
-enable-recode	activa el soporte de codificación cirílica
-with-mb=encoding	activa el soporte para multi-byte
-with-pgport=portnum	cambia el puerto de inicio por defecto
-with-maxbackends=n	define el número máximo por defecto de procesos servidores
-with-tcl	construye interfaces Tcl y pgtclsh
-with-tclconfig=tcldir	tclConfig.sh y tkConfig.sh están en DIR
-with-perl	construye interfaces con Perl
-with-odbc	construye el paquete del driver ODBC
-with-odbcinst=odbcdir	cambia el directorio por defecto de odbcinst.ini
-enable-cassert	activa los chequeos de afirmación (depurando)
-with-CC=compiler	usa el compilador de C especificado
-with-CXX=compiler	usa el compilador de C++ especificado
-without-CXX	previene la construcción de código C++

Algunos sistemas pueden tener problemas de construcción con algunas características específicas de Postgres. Por ejemplo, sistemas con el compilador de C++ dañado

pueden necesitar especificar `-without-cxx` para el proceso de construcción para saltarse la construcción de `libpq++`.

Parámetros de construcción (make)

Muchos parámetros relacionados con la instalación pueden activar en la etapa de construcción de la instalación de Postgres.

En muchos casos, estos parámetros deben colocarse en un fichero, `Makefile.custom`, utilizado para este propósito. La distribución por defecto no contiene este fichero opcional, pero puedes crearlo con el editor de texto que tu elijas. Cuando actualizas una instalación, tu puedes simplemente copiar tu viejo `Makefile.custom` a la nueva instalación antes que hagas la construcción.

```
make [ variable=value [,...] ]
```

Unas pocas de las muchas variables que puedes especificar son:

POSTGRES DIR

Lo más alto en el árbol de la instalación.

BINDIR

Localización de las aplicaciones y utilidades.

LIBDIR

localización de las librerías, incluyendo las librerías compartidas.

HEADERDIR

Localización de los ficheros include.

ODBCINST

localización de las librerías, incluyendo las librerías compartidas `psqlODBC` (ODBC)

.

Hay otros parámetros opcionales que no se utilizan comúnmente. Muchos de las que listan debajo son apropiadas cuando se estaba desarrollando el código del servidor Postgres.

CFLAGS

Establece los flags para el compilador de C. Debe ser especificado con `"+="` para conservar los parámetros por defecto.

YFLAGS

Establece los flags para el parser yacc/bison. Puede usarse `-v` para ayudar a diagnosticar problemas de construcción de un nuevo parser. Debe ser especificado con `"+="` para conservar los parámetros por defecto.

USE_TCL

Activa el constructor del interfaces Tcl.

HSTYLE

Páginas HTML estilo DocBook para construir la documentación de partida. No usar a menos que tu estés desarrollando nueva documentación de documentos fuente SGML compatibles con DocBook en `doc/src/sgml/`.

PSTYLE

Páginas estilo DocBook para construir la documentación impresa de partida. No usar a menos que tu estés desarrollando nueva documentación de documentos fuente SGML compatibles con DocBook en `doc/src/sgml/`.

Aquí hay un ejemplo de `Makefile.custom` para un sistema Linux PentiumPro:

```
# Makefile.custom
# Thomas Lockhart 1999-06-01

POSTGRES_DIR= /opt/postgres/current
CFLAGS+= -m486 -O2

# documentation

HSTYLE= /home/tgl/SGML/db118.d/docbook/html
PSTYLE= /home/tgl/SGML/db118.d/docbook/print
```

Soporte Local

Nota: Escrito por Oleg Bartunov. Ver Oleg's web page¹ para más información sobre el soporte de lengua local y Rusa.

Mientras que estaba en un proyecto para una compañía en Moscú, Rusia, Me encontré con el problema que postgresql no tenía soporte para alfabetos nacionales. Después de mirar posibles soluciones alternativas decidí desarrollar un soporte local yo mismo. No soy un programador en C pero ya había tenido experiencia con la programación local cuando trabajo en perl (depurando) y glimpse. Después de bastantes días sumergido por el árbol de fuente de Postgres Realice muy pocas correcciones en `src/backend/utils/adt/varlena.c` and `src/backend/main/main.c` para conseguir lo que quería! Di soporte sólo para `LC_CTYPE` and `LC_COLLATE`, pero más tarde otros lo añadieron para `LC_MONETARY`. Tuve muchos mensajes de la gente a cerca de este parche por eso decidí enviárselo a los desarrolladores y (sorprendentemente) lo incorporaron dentro de la distribución Postgres.

La gente a veces se queja que el soporte local no funciona para ellos. Hay algunos errores comunes:

- No configurar debidamente postgresql antes de compilarlo. Tu debes ejecutar la configuración con la opción `-enable-locale` para activar el soporte local. No iniciar el entorno correctamente cuando se inicia postmaster. Tu debes definir las variables de entorno `LC_CTYPE` and `LC_COLLATE` antes de ejecutar postmaster porque por detrás coge información local del entorno. Yo uso el siguiente shell script (`runpostgres`):

```
#!/bin/sh

export LC_CTYPE=koi8-r
export LC_COLLATE=koi8-r
postmaster -B 1024 -S -D/usr/local/pgsql/data/ -o '-Fe'
```

y lo ejecuto en rc.local así

```
/bin/su - postgres -c "/home/postgres/runpostgres"
```

- Un soporte local estropeado en un OS (por ejemplo, el soporte local en libc bajo Linux algunas veces ha sido cambiado y esto ha causado muchos problemas) El más reciente perl tiene también soporte local y si el soporte local es defectuoso **perl -v** da un aviso parecido a esto:

```
8:17[mira]:~/WWW/postgres>setenv LC_CTYPE not_exist
8:18[mira]:~/WWW/postgres>perl -v
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
LC_ALL = (unset),
  LC_CTYPE = "not_exist",
  LANG = (unset)
are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").
```

- Localización errónea de los ficheros locales! Las posibles localizaciones son: /usr/lib/locale (Linux, Solaris), /usr/share/locale (Linux), /usr/lib/nls/loc (DUX 4.0). Chequea **man locale** para encontrar la localización correcta. Bajo Linux yo hice un enlace simbólico entre /usr/lib/locale y /usr/share/locale para estar seguro que la próxima libc no estropea mi soporte local.

Cuales son los Beneficios?

Tu puedes usar ~* y el operador order by para cadenas que contienen caracteres de alfabetos nacionales. Los usuarios no Ingleses definitivamente lo necesitan. Si tu no quieres usar el soporte local libera la variable USE_LOCALE.

Cuales son las Desventajas?

Hay una evidente desventaja si utilizamos el soporte local - que el la velocidad! Por eso, utilízalo sólo si verdaderamente lo necesitas.

Autenticación Kerberos

Kerberos es un sistema de autenticación de seguridad estándar industrial indicado para ser distribuido bajo redes publicas.

Disponibilidad

El sistema de autenticación Kerberos no se distribuye con Postgres. Las versiones de Kerberos están típicamente disponibles como software opcional para los vendedores de sistemas operativos. Además, la distribución del código fuente se puede obtener a través de MIT Project Athena².

Nota: Puedes desear obtener la versión del MIT si tu proveedor te proporciona una versión, ya que en algunos puntos de venta está deliberadamente capado o proporcionado sin interoperatividad con la versión del MIT.

Los usuarios que se encuentran fuera de los Estados Unidos de América o Canadá están avisados de que la distribución del código actual de encriptación de Kerberos está restringida por las leyes de exportación de Gobierno U. S.

Las preguntas acerca de tu Kerberos deben estar dirigidas a tu proveedor o a MIT Project Athena³. Notar que las FAQLs (Lista de preguntas frecuentes) se envían periódicamente a la Kerberos mailing list⁴ (envía to subscribe⁵), y USENET news group⁶.

Instalación

Instalación de Kerberos se trata en detalle en las *Notas de instalación de Kerberos*. Tener cuidado que el fichero llave servidor (the `srvtab` o `keytab`) es de alguna manera accesible para la lectura por la cuenta Postgres.

Postgres y sus clientes pueden ser compilables para usar la versión 4 o la versión 5 de los protocolos Kerberos del MIT configurando la variable `KRBVERS` en el fichero `src/Makefile.global` al valor apropiado. Puedes también cambiar la localización donde espera encontrar Postgres las librerías asociadas, los ficheros de cabecera y el fichero llave del servidor.

Cuando la compilación se haya completado, Postgres debe registrarse como un servicio Kerberos. Mirar las *Notas de Operaciones de Kerberos* y las páginas del manual relacionadas para más detalle del registro de servicios.

Operaciones

Después de la instalación inicial, Postgres debe operar en todos los sentidos como un servicio normal Kerberos. Para más detalles en el uso de la autenticación, ver la *Guía del Usuario PostgreSQL* en las secciones referentes a `postmaster` y `psql`.

En los comentarios de la versión 5 de Kerberos, las siguientes suposiciones están hechas para el usuario y el nombrado de servicio:

- Los nombres de los usuarios principales (anames) se asemen para contener el nombre del usuario actual Unix/Postgres en el primer componente.
- El servicio Postgres se asume para ser tenido como dos componentes, el nombre de servicio y el nombre de host, canonizada en la versión 4 (p.e., con todos los sufijos de dominio borrados).

Tabla 18-1. Ejemplos de Parámetros de Kerberos

Parámetro	Ejemplo
user	frew@S2K.ORG
user	ao- ki/HOST=miyu.S2K.Berkeley.EDU@S2K.ORG
host	postgres_dbms/ucbvax@S2K.ORG

El soporte de la versión 4 desaparecerá después de algún tiempo tras la puesta en producción de la revisión de la versión 5 del MIT.

Notas

1. <http://www.sai.msu.su/~megera/postgres/>
2. <ftp://athena-dist.mit.edu>
3. info-kerberos@athena.mit.edu
4. <mailto:kerberos@ATHENA.MIT.EDU>
5. <mailto:kerberos-request@ATHENA.MIT.EDU>
6. <news:comp.protocols.kerberos>

Capítulo 19. Distribución del Sistema

Figura 19-1. Distribución de los archivos de Postgres

Distribución de los archivos de Postgres muestra cómo se ordena la distribución de Postgres en su disco, cuando es instalada de la forma usual. Para simplificar, asumiremos que Postgres se ha instalado en el directorio `/usr/local/pgsql`. Por lo tanto, siempre que se mencione el directorio `/usr/local/pgsql`, usted deberá sustituirlo por aquél en el que haya instalado Postgres en su sistema. Todas los órdenes de Postgres se instalan en en el directorio `/usr/local/pgsql/bin`. Por consiguiente, debería incluir este directorio en su variable de entorno de path. Si utiliza un intérprete de órdenes derivado del Berkeley C, como `csh` o `tcsh`, incluya la línea

```
set path = ( /usr/local/pgsql/bin path )
```

en el archivo `.login` de su directorio particular (`home`). Si, en cambio, utiliza un intérprete de órdenes derivado del Bourne, como `sh`, `ksh` o `bash`, deberá agregar las líneas:

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

al archivo `.profile` ubicado en su directorio de inicio. De aquí en adelante se asumirá que usted ha agregado el directorio `/usr/local/pgsql/bin` a su path. Además, en este documento se hará referencia frecuentemente a «fijar una variable del intérprete de órdenes» o «fijar una variable de entorno». Si no comprendió totalmente el último párrafo sobre la modificación de la variable `PATH`, debería consultar la documentación sobre el funcionamiento de su intérprete de órdenes antes de continuar.

Si no ha realizado la instalación con las opciones por defecto, tal vez tenga algo de trabajo extra. Por ejemplo, si el servidor de base de datos está ubicado en una máquina remota, necesitará colocar el nombre del servidor en la variable de entorno `PGHOST`. Tal vez también necesite fijar la variable de entorno `PGPORT`. Básicamente, si intenta ejecutar una aplicación y ésta se queja por no poder conectarse con el proceso principal (`postmaster`), deberá volver un poco sobre sus pasos y asegurarse de que ha establecido todas las variables de entorno necesarias con los valores correctos.

Capítulo 20. Instalación

Instrucciones para la instalación de PostgreSQL 7.0.

Los comandos que se mencionan a continuación fueron probados utilizando el shell `bash` sobre la distribución RedHat 5.2 de Linux. A menos que se indique lo contrario, estos comandos serán igualmente aplicables para la mayoría de los sistemas. Comandos como `ps` y `tar` pueden variar entre las distintas plataformas en cuanto a las opciones que deben usarse. *Utilice su sentido común* antes de teclear cualquiera de estos comandos.

Si aún no tiene la distribución de PostgreSQL, puede obtenerla en ftp.postgresql.org¹. Una vez obtenida, desempaquetela utilizando los siguientes comandos:

```
$ gunzip postgresql-7.0.tar.gz
$ tar -xf postgresql-7.0.tar
$ mv postgresql-7.0 /usr/src
```

Nuevamente, estos comandos pueden ser distintos en su sistema.

Antes de comenzar

Para compilar PostgreSQL se requiere la utilidad GNU `make`. Otras utilidades similares *no funcionarán* en este caso. En los sistemas GNU/Linux, GNU `make` es a herramienta por defecto. En otros sistemas puede que encuentre que la herramienta GNU `make` se encuentre instalada con el nombre “`gmake`”. De aquí en adelante, utilizaremos este nombre para referirnos a GNU `make`, sin importar cuál sea el nombre que tiene en su sistema. Para probar GNU `make` teclee:

```
$ gmake -version
```

Si necesita obtener GNU `make`, lo puede encontrar en [ftp://ftp.gnu.org](http://ftp.gnu.org)².

En <http://www.postgresql.org/docs/admin/ports.htm>³ puede encontrar información actualizada sobre las plataformas soportadas. En general, la mayoría de la plataformas compatibles con Unix que utilicen bibliotecas actualizadas debería ser capaz de ejecutar PostgreSQL. En el subdirectorio `doc` de la distribución existen varios documentos del tipo LEAME y otros con Preguntas de Uso Frecuente (FAQ en inglés) específicos para esa distribución, que pueden resultarle útiles si está teniendo problemas.

La cantidad mínima de memoria que se requiere para ejecutar PostgreSQL es de sólo 8 MB. Sin embargo, se verifica una notable mejora en la velocidad cuando ésta se expande a 96 MB o más. La regla es que, por más memoria que usted instale en su sistema, nunca será demasiada.

Verifique que exista suficiente espacio libre en el disco. Necesitará alrededor de 30 MB para los archivos con el código fuente durante la compilación, y unos 5 MB para el directorio de instalación. Una base de datos vacía ocupa aproximadamente 1 MB. De no estar vacía, la base ocupará unas cinco veces el espacio que ocuparía un archivo de texto que contuviera los mismos datos. Si ejecuta las pruebas de regresión, necesitará alrededor de 20 MB extra como espacio temporal.

Para revisar el espacio libre en el disco, utilice:

```
$ df -k
```

Dados los precios actuales de los discos duros, debería considerar conseguir uno grande y rápido antes de poner a trabajar una base de datos.

Procedimiento de Instalación

Instalación de PostgreSQL

Para una instalación de nuevas, o una actualización desde una versión previa de PostgreSQL:

1. Cree la cuenta de superusuario PostgreSQL. Éste es el usuario bajo el que corre el servidor. Para el uso en producción, deberá usted crear una cuenta de usuario diferente, sin privilegios (habitualmente se utiliza `postgres`). Si no tiene usted acceso como `root`, o quiere evitarse este paso, su propia cuenta de usuario es suficiente.

Ejecutar PostgreSQL como `root`, `bin`, o cualquier otra cuenta con permisos de acceso especiales es un riesgo de seguridad, y por ello está permitido.

No necesitará usted compilar e instalar bajo esta cuenta (aunque puede hacerlo). Se le dirá cuando necesite conectarse como el superusuario de la base de datos.

2. Si no está usted actualizando un sistema existente, salte a paso 4.

Ahora necesitará usted realizar una copia de seguridad (backup) de su base de datos existente. Si tiene una instalación razonablemente reciente de su base de datos (posterior a la 6.0), conseguirá un vaciado de la misma tecleando:

```
$ pg_dumpall > db.out
```

Si quiere usted conservar las identificación de los objetos (oids), utilice la opción `-o` al ejecutar `pg_dumpall`. Sin embargo, a no ser que tenga usted una razón especial para hacer esto, como podría ser utilizar estos identificadores como claves en las tablas, no lo haga.

Asegurese de utilizar el comando `pg_dumpall` de la versión que está usted ejecutando actualmente. Pero no utilice el script de 6.0 o el superusuario PostgreSQL tomará la propiedad de *todo*. Si es esta la versión que tiene usted, debería usted utilizar el comando `pg_dumpall` de una versión 6.x.x posterior. El correspondiente a la versión 7.0 no trabajará en bases de datos anteriores. Si está usted actualizando desde una versión previa a Postgres95 v1.09, deberá usted realizar un backup de su base de datos, instalar Postgres95 v1.09, restaurar su base de datos, y realizar el backup de nuevo.

Atención

Debe usted asegurarse de que su base de datos no se actualiza durante su backup. Si es necesario, pare el postmaster, edite los permisos del fichero `/usr/local/pgsql/data/pg_hba.conf` para permitirle a usted sólo su uso, y relance de nuevo postmaster.

3. Si está usted actualizando un sistema existente, mate ahora el servidor de la base de datos. Teclee:

```
$ ps ax | grep postmaster
```

Esto debería listar los números de proceso para una serie de procesos, de un modo similar a:

```
263 ? SW 0:00 (postmaster)
777 p1 S 0:00 grep postmaster
```

Teclee la siguiente línea, reemplazando *pid* con el identificador (id) del proceso *postmaster* (263 en el caso anterior). (No utilice el id del proceso "grep postmaster".) (N. del T. también puede hacerlo con la línea

```
$ ps ax | grep postmaster | grep -v grep
```

que le dará la misma salida, pero sin incluir la línea correspondiente al mismo proceso "grep". Fin de la N. del T.)

```
$ kill pid
```

Sugerencia: En sistemas que arrancan PostgreSQL en el durante la secuencia de arranque de la máquina, probablemente se encontrara un fichero startup que cumplirá el mismo cometido. Por ejemplo, en un sistema Linux RedHat, se debería encontrar que

```
$ /etc/rc.d/init.d/postgres.init stop
```

funcione correctamente para parar la base.

También deberá trasladar los directorios anteriores a otro sitio. Teclee lo siguiente:

```
$ mv /usr/local/pgsql /usr/local/pgsql.old
```

con sus propias rutas particulares.

4. Configure el código fuente para su sistema. Este es el paso en el que puede usted especificar su ruta de instalación actual para el proceso de construcción, y hacer elecciones sobre lo que tenga usted instalado. Cambiese al subdirectorio *src* y teclee:

```
$ ./configure
```

seguido de todas las opciones que desee usted dar. Para una primera instalación, debería ir todo bien sin dar ninguna. Para obtener una lista completa de las opciones, teclee:

```
./configure -help
```

Algunas de las opciones que se utilizan más a menudo son:

-prefix=BASEDIR

Selecciona un directorio base diferente para la instalación de PostgreSQL. La opción de defecto es */usr/local/pgsql*.

-enable-locale

Si quiere usted utilizar locales.

-enable-multibyte

Le permitirá utilizar páginas de caracteres multibyte. Se emplea principalmente para lenguajes como japonés, coreano o chino.

`-with-perl`

Construye la interface Perl. Note por favor que la interface Perl se instalará en el lugar habitual de los módulos Perl (habitualmente bajo `/usr/lib/perl`), de modo que deberá usted tener acceso root para realizar esta opción correctamente.

`-with-odbc`

Construye el paquete del driver ODBC.

`-with-tcl`

Construye las librerías de interface y los programas que requieren Tcl/Tk, incluyendo `libpgtcl`, `pgtclsh` y `pgtksh`.

5. Compile el programa. Teclee:

```
$ make
```

El proceso de compilación ocupará entre 10 minutos y una hora, variando en función de la máquina y de las opciones elegidas.

La última línea que se muestre por el proceso debería ser:

```
All of PostgreSQL is successfully made. Ready to install.
```

Recuerde que “make” se puede llamar “make” en su sistema.

6. Instale el programa. Teclee:

```
$ make install
```

7. Dígle a su sistema como encontrar las nuevas librerías compartidas. Cómo hacer esto varía de unas plataformas a otras. Lo que tiende a trabajar en todas partes es fijar la variable de entorno `LD_LIBRARY_PATH`:

```
$ LD_LIBRARY_PATH=/usr/local/pgsql/lib
$ export LD_LIBRARY_PATH
```

Quizá quiera usted poner estas dos líneas en un script de arranque de su shell, como `~/.bash_profile`.

En algunos sistemas se prefiere el siguiente método, pero debe usted tener acceso root. Edite el fichero `/etc/ld.so.conf` y añada una línea

```
/usr/local/pgsql/lib
```

Y ahora corra el comando `/sbin/ldconfig`.

En la duda, diríjase a las páginas de manual de su sistema. Si recibe usted más tarde un mensaje como

```
./psql: error in loading shared libraries
libpq.so.2.1: cannot open shared object file: No such file or directory
```

entonces es que todo lo anterior era necesario. Simplemente realice este paso de nuevo.

8. Cree la instalación de la base de datos. Para hacer esto, debe usted conectarse como su cuenta de superusuario de PostgreSQL. No trabajará como root.

```
$ mkdir /usr/local/pgsql/data
$ chown postgres /usr/local/pgsql/data
$ su - postgres
$ /usr/local/pgsql/initdb -D /usr/local/pgsql/data
```


La opción `-D` especifica la situación donde se almacenarán los datos. Puede usted utilizar cualquier otro path, porque no tiene porqué estar bajo el directorio de la instalación. Sólo asegúrese de que la cuenta del superusuario puede escribir en el directorio (o crearlo) antes de arrancar **initdb**. (Si estaba usted siguiendo los pasos de la instalación hasta ahora como el superusuario de PostgreSQL, puede que tenga usted que conectarse como root temporalmente para crear el directorio de datos.)

9. Los pasos previos deberían haberle indicado como arrancar el servidor de la base de datos. Ahagamos ahora:

```
$ /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data
```

Esto arrancará el servidor en primer término. Para mandarlo a segundo plano, utilice la opción `-s`.

10. Si está usted actualizando desde una instalación anterior, extraiga sus datos con:

```
$ /usr/local/pgsql/bin/psql < db.out
```

Y haga también una copia de seguridad de su anterior fichero `pg_hba.conf`, así como de todos los demás ficheros que pueda usted haber creado para la autenticación, tales como ficheros de claves de acceso.

Con todo esto concluimos la instalación propiamente dicha. Para hacer su vida más productiva y agradable, debería mirar los siguientes pasos y sugerencias opcionales.

- La vida será más conveniente si fija usted algunas variables de entorno. Primero, probablemente quiera usted incluir `/usr/local/pgsql/bin` (o su equivalente) en su PATH. Para hacer esto, añada lo siguiente en su fichero de arranque de la shell, tal como `~/.bash_profile` (o `/etc/profile`, si quiere usted que afecte a todos los usuarios):

```
PATH=$PATH:/usr/local/pgsql/bin
```

Aún más, si usted fija la variable PGDATA en el entorno del superusuario de PostgreSQL, podrá usted omitir la opción `-D` para `postmaster` y `initdb`.

- Probablemente quiera usted instalar la documentación man y HTML. Teclee

```
$ cd /usr/src/pgsql/postgresql-7.0/doc
$ gmake install
```

Esto instalará ficheros bajo `/usr/local/pgsql/doc` y `/usr/local/pgsql/man`. Para permitir a su sistema encontrar la documentación man, necesitará añadir una línea como la siguiente en el fichero de arranque de la shell:

```
MANPATH=$MANPATH:/usr/local/pgsql/man
```

La documentación está también disponible en formato Postscript. Si tiene usted una impresora Postscript, o tiene su impresora ya preparada para aceptar ficheros Postscript utilizando un filtro de impresión, podrá imprimir la Guía de Usuario simplemente tecleando

```
$ cd /usr/local/pgsql/doc
$ gunzip -c user.ps.tz | lpr
```

Aquí tiene lo que debería hacer usted si tiene Ghostscript en su sistema y está escribiendo en una impresora laserjet:

```
$ alias gshp='gs -sDEVICE=laserjet -r300 -dNOPAUSE'
$ export GS_LIB=/usr/share/ghostscript:/usr/share/ghostscript/fonts
```

```
$ gunzip user.ps.gz
$ gshp -sOUTPUTFILE=user.hp user.ps
$ gzip user.ps
$ lpr -l -s -r manpage.hp
```

En caso de dudas, refierase a sus manuales o a su experto local.

Probablemente debería empezar por leer la Guía del Administrador si es usted completamente nuevo en PostgreSQL, porque contiene información sobre como declarar usuarios y la autenticación a la base de datos.

- Habitualmente, querrá usted modificar su computadora de modo que arranque el servidor de base de datos siempre que se ponga en marcha. Esto no es necesario; el servidor PostgreSQL se puede ejecutar normalmente desde cuentas no privilegiadas sin intervención de root.

Diferentes sistemas tienen diferentes convenciones para arrancar demonios en el momento de la puesta en marcha, de modo que deberá usted familiarizarse primer con ellos. La mayoría de los sistemas tienen un fichero `/etc/rc.local` o `/etc/rc.d/rc.local` que en la mayoría de los casos no es un mal lugar para situar este comando. Siempre que lo haga, el postmaster deberá ser ejecutado por el superusuario de PostgreSQL (`postgres`) y *no por root* o cualquier otro usuario. Por ello, probablemente quiera usted formar las líneas de comando iniciandolas con su `-c '...' postgres`.

Podría ser interesante mantener un registro de las salidas del servidor. Para arrancar de esta forma el servidor, intente:

```
nohup su -c 'postmaster -D /usr/local/pgsql/data > server.log 2>&1' postgres &
```

Aquí tenemos algunas otras sugerencias específicas del sistema operativo:

- Edite el fichero `rc.local` en NetBSD o el fichero `rc2.d` en SPARC Solaris 2.5.1 para que contenga la siguiente línea:

```
su postgres -c "/usr/local/pgsql/bin/postmaster -S -D /usr/local/pgsql/data"
```

- En FreeBSD RELEASE-2.2 editE `/usr/local/etc/rc.d/pgsql.sh` para que contenga las siguientes líneas y hégale `chmod 755` y `chown root:bin`.

```
#!/bin/sh
[ -x /usr/local/pgsql/bin/postmaster ] && {
    su -l postgres -c 'exec /usr/local/pgsql/bin/postmaster
        -D/usr/local/pgsql/data
        -S -o -F > /usr/local/pgsql/errlog' &
    echo -n ' postgres'
}
```

Usted puede colocar las rupturas de líneas como se muestra antes. La shell es capaz de seguir traduciendo más allá del final de la línea si no se ha terminado una expresión. El `exec` salva un nivel de shell bajo el proceso postmaster, de modo que el padre es `init`.

- En Linux RedHat, añada un fichero `/etc/rc.d/init.d/postgres.init` que se basará en el ejemplo que se encuentra en `contrib/linux/`. Y a continuación haga un link simbólico a este fichero desde `/etc/rc.d/rc5.d/S98postgres.init`.

- Ejecute los test de regresión. Los test de regresión son un conjunto de pruebas que verifican que PostgreSQL corre en su máquina en la forma en que los desarrolladores esperan que lo haga. Debería hacer esto definitivamente antes de poner un servidor en uso en producción. El fichero `/usr/src/pgsql/postgresql-7.0/src/test/regress/README` contiene instrucciones detalladas para correr e interpretar los tests de regresión.

Notas

1. <ftp://ftp.postgresql.org>
2. <ftp://ftp.gnu.org>
3. <http://www.postgresql.org/docs/admin/ports.htm>

Capítulo 21. Instalacion en Win32

Instrucciones para la instalacion de Postgres version 6.4(libreria de clientes) en Win32.

Construccion de librerias

Los "makefiles" incluidos en Postgres estan escritos para Microsoft Visual C++ y probablemente no trabajen en otros sistemas. Es posible compilar las librerias manualmente en otros casos.

Para contruir las librerias, cambie al directorio `src` y escriba los comandos

```
copy include\config.h.win32 include\config.h
nmake /f win32.mak
```

Esto asume que usted tiene Visual C++ en su camino.

Los siguientes archivos seran creados:

- `interfaces\libpq\Release\libpq.dll` - La libreria dinamica enlazable
- `interfaces\libpq\Release\libpqdll.lib` - Libreria Importada para conectar el programa a `libpq.dll`
- `interfaces\libpq\Release\libpq.lib` - Version Estatica de la libreria
- `bin\psql\Release\psql.exe` - El monitor interactivo de SQL

Instalacion de las librerias

La unica parte de la libreria que sera realmente instalada es la libreria de `libpq.dll`. El archivo en la mayoria de los casos debe ser puesto en el directorio `WINNT\SYSTEM32` (o en `WINDOWS\SYSTEM` en un sistema Windows 95/98). Si este archivo es instalado usando un programa de instalacion, debe ser instalado con un examinador de versiones usando `VERSIONINFO` que esta incluido en el archivo, para asegurar que una version mas reciente de la libreria no sea sobre escrita.

Si planea desarrollar usando `libpq` en esta maquina, tendra que anadir los directorios `src\include` y `src\interfaces\libpq` al camino en sus configuraciones.

Usando las librerias

Para usar las librerias, debe anadir los archivos `libpqdll.lib` a su proyecto (en Visual C++, solo haga un click con el boton derecho en el proyecto y escoja anadirlo).

Una vez que esto esta hecho, sera posible usar la libreria como lo haria en cualquiera plataforma basada en UNIX.

Capítulo 22. Entorno de tiempo de ejecución

En este capítulo se detalla la interacción entre Postgres y el sistema operativo.

Utilizando Postgres desde Unix

Todas las órdenes de Postgres que se ejecutan directamente desde un shell de Unix se encuentran en el directorio ".../bin". Incluir este directorio en su variable de entorno path facilitará mucho la ejecución de los mismos.

Existe una colección de catálogos del sistema en cada servidor. La misma incluye una clase (pg_user) que contiene una instancia para cada usuario válido en Postgres. La instancia especifica un conjunto de privilegios sobre Postgres, como la posibilidad de actuar como superusuario en Postgres, la posibilidad de crear/destruir bases de datos y la posibilidad de actualizar los catálogos del sistema. Un usuario de Unix no puede hacer nada con Postgres hasta que se instale una instancia apropiada en dicha clase. Puede encontrarse más información sobre los catálogos del sistema ejecutando consultas sobre las clases apropiadas.

Iniciando postmaster

No le puede suceder nada a una base de datos a menos que esté corriendo el proceso postmaster. Como administrador, hay una serie de cosas que debe recordar antes de iniciar postmaster. Vea las secciones de instalación y configuración en este mismo manual. De todos modos, si ha unсталado Postgres siguiendo las instrucciones de instalación al pie de la letra, lo único que tendrá que hacer para iniciar el proceso postmaster es introducir esta simple orden:

```
% postmaster
```

Ocasionalmente, postmaster escribe mensajes que le serán de ayuda para resolver problemas. Si desea ver los mensajes de diagnóstico de postmaster, puede iniciarlo con la opción -d y redirigir la salida al archivo de registro:

```
% postmaster -d > pm.log 2>&1 &
```

Si no desea ver los mensajes, inícielo de la forma

```
% postmaster -S
```

y postmaster será "S"ilencioso. Observe que al no haber el signo ampersand ("&") al final del último ejemplo, no se ejecuta como proceso de fondo.

Usando pg_options

Nota: Contribución de Massimo Dal Zotto¹

El archivo opcional `data/pg_options` contiene opciones usadas por el backend para controlar los mensajes de trazado y otros parámetros ajustables del mismo. El archivo se vuelve a leer cuando el backend recibe la señal `SIGHUP`, permitiendo cambiar las opciones de tiempo de ejecución al vuelo, sin que sea preciso reiniciar Postgres. En este archivo se pueden incluir opciones de depuración usadas por el paquete de trazado (`backend/utils/misc/trace.c`) o parámetros numéricos usados por el backend para controlar su comportamiento.

Todas las `pg_options` se inicializan a cero al iniciar el backend. Si se añaden o se modifican opciones, serán leídas por todos los backend que se inicien a continuación. Para que cualquier cambio tome efecto en los backend que están activos, es preciso enviar una señal `SIGHUP` al postmaster, quien reenviará la señal a todos los backend activos. Se pueden activar los cambios para un backend específico enviándole directamente una señal `SIGHUP`.

Las `pg_options` pueden especificarse también con la opción `-T` de Postgres:

```
postgres opciones -T "verbose=2,query,hostlookup-"
```

Las funciones usadas para indicar errores y mensajes de depuración pueden usar la facilidad `syslog(2)`. Los mensajes que se escriben en `stdout` o `stderr` incluyen un prefijo con la fecha, la hora y el número de proceso del backend que las genera.

```
#timestamp      #pid      #message
980127.17:52:14.173 [29271] StartTransactionCommand
980127.17:52:14.174 [29271] ProcessUtility: drop table t;
980127.17:52:14.186 [29271] SIIncNumEntries: table is 70% full
980127.17:52:14.186 [29286] Async_NotifyHandler
980127.17:52:14.186 [29286] Waking up sleeping backend process
980127.19:52:14.292 [29286] Async_NotifyFrontEnd
980127.19:52:14.413 [29286] Async_NotifyFrontEnd done
980127.19:52:14.466 [29286] Async_NotifyHandler done
```

Este formato facilita la lectura de los mensajes y permite saber exactamente lo que hace cada backend y en qué momento. También facilita la escritura de scripts `awk` o `perl` sencillos que analicen los historiales para detectar errores o problemas en la base de datos o para calcular estadísticas temporales de transacciones.

Los mensajes escritos a través de `syslog` utilizan la facilidad `LOG_LOCAL0`. El uso de `syslog` se controla mediante la opción `syslog` en `pg_options`. Por desgracia, muchas funciones llaman directamente a `printf()` para escribir sus mensajes a `stdout` o `stderr`, cuya salida no se puede controlar mediante la opción `syslog` ni puede incluir fecha y hora. Sería recomendable sustituir todas las llamadas a `printf` con la macro `PRINTF`, y todas las escrituras a `stderr` por la macro `EPRINTF`, para poder controlar toda la salida de un modo uniforme.

El formato del archivo `pg_options` es como sigue:

```
# comentario
option=valor_entero # Establece el valor de option
option              # establece option = 1
option+             # establece option = 1
option-             # establece option = 0
```


Observe que *keyword* puede ser una abreviatura de un nombre de opción de los definidos en `backend/utils/misc/trace.c`.

Ejemplo 22-1. Archivo `pg_options`

Por ejemplo, mi archivo `pg_options` contiene los siguientes valores:

```
verbose=2
query
hostlookup
showportnumber
```

Opciones reconocidas

Actualmente están definidas las opciones:

`all`

Marca de traza global. Los valores permitidos son:

0

Mensajes de trazado activados individualmente

1

Activar todos los mensajes de trazado

-1

Inhibir todos los mensajes de trazado

`verbose`

Marca de verbosidad. Valores permitidos:

0

Sin mensajes, éste es el valor por omisión.

1

Escribir mensajes de información.

2

Escribir más mensajes de información.

`query`

Trazar peticiones. Valores permitidos:

0

No escribir la petición.

1

Escribir una versión condensada de la petición en una línea.

4

Escribir la consulta completa.

plan

Escribir el plan de consulta.

parse

Escribir la salida del traductor de consultas.

rewritten

Escribir la consulta reescrita.

parserstats

Escribir las estadísticas del traductor de consultas.

plannerstats

Escribir las estadísticas del planificador.

executorstats

Escribir las estadísticas de ejecución.

shortlocks

De momento no se usa, pero se precisa para habilitar nuevas características en el futuro.

locks

Trazar bloqueos.

userlocks

Trazar bloqueos de usuario.

spinlocks

Trazar 'spin locks'.

notify

Trazar funciones de notificación.

malloc

Sin uso por el momento.

palloc

Sin uso por el momento.

lock_debug_oidmin

Minimum relation oid traced by locks.

lock_debug_relid

oid, if not zero, of relation traced by locks.

lock_read_priority

Sin uso por el momento.

deadlock_timeout

Temporizador de comprobación de bloqueos circulares..

syslog

Marca de syslog. Valores permitidos:

0

Mensajes a stdout/stderr.

1

Mensajes a stdout/stderr y syslog.

2

Mensajes solamente a syslog.

hostlookup

Habilitar la consulta de nombre de host en ps_status.

showportnumber

Mostrar el número de puerto en ps_status.

notifyunlock

Desbloqueo de pg_listener después de notify.

notifyhack

Borrar tuplas duplicadas de pg_listener.

Notas

1. <mailto:dz@cs.unitn.it>

Capítulo 23. Seguridad

La seguridad de la base de datos esta implementada en varios niveles:

- Protección de los ficheros de la base de datos. Todos los ficheros almacenados en la base de datos estan protegidos contra escritura por cualquier cuenta que no sea la del superusuario de Postgres.
- Las conexiones de los clientes al servidor de la base de datos estan permitidas, por defecto, únicamente mediante sockets Unix locales y no mediante sockets TCP/IP. Ha de arrancarse el demonio con la opcion `-i` para permitir la conexion de clientes no locales.
- Las conexiones de los clientes se pueden restringir por dirección IP y/o por nombre de usuario mediante el fichero `pg_hba.conf` situado en `PG_DATA`.
- Las conexiones de los clientes pueden ser autenticadas mediante otros paquetes externos.
- A cada usuario de Postgres se le asigna un nombre de usuario y (opcionalmente) una contraseña. Por defecto, los usuarios no tienen permiso de escritura a bases de datos que no hayan creado.
- Los usuarios pueden ser incluidos en *grupos*, y el acceso a las tablas puede restringirse en base a esos grupos.

Autenticacion de Usuarios

Autenticacion es el proceso mediante el cual el servidor de la base de datos y el `postmaster` se aseguran de que el usuario que está solicitando acceso a la base de datos es en realidad quien dice ser. Todos los usuarios que quieren utilizar Postgres se comprueban en la tabla `pg_user` para asegurarse que están autorizados a hacerlo. Actualmente, la verificación de la identidad del usuario se realiza de distintas formas:

Desde la shell del usuario

Un demonio que se lanza desde la shell del usuario anota el id original del usuario antes de realizar un `setuid` al id del usuario `postgres`. El id original del usuario se emplea como base para todo tipo de comprobaciones.

Desde la red

Si Postgres se instala como distribuido, el acceso al puerto TCP del `postmaster` está disponible para todo el mundo. El ABD configura el fichero `pg_hba.conf` situado en el directorio `PG_DATA` especificando el sistema de autenticacion a utilizar en base al equipo que realiza la conexión y la base de datos a la que se conecta. Ver `pg_hba.conf(5)` para obtener una descripción de los sistemas de autenticación disponibles. Por supuesto la autenticación basada en equipos no es perfecta incluso en los sistemas Unix. Es posible, para determinados intrusos, enmascarar el equipo de origen. Estos temas de seguridad están fuera del alcance de Postgres.

Nombres de usuario y grupos

Para crear un nuevo usuario, ejecute el programa `createuser`.

Para añadir un usuario o un grupo de usuarios a un nuevo grupo uno de los usuarios debe crear el grupo y añadir al resto a ese grupo. En `Postgres` estos pasos no pueden realizarse actualmente mediante el comando **`create group`**. Los grupos se definen añadiendo los valores a la tabla `pg_group`, y usando el comando **`grant`** para asignar privilegios al grupo.

Crear Usuarios

Crear Grupos

Actualmente no hay una forma fácil de crear grupos de usuarios. Hay que añadirlos/actualizarlos uno a uno en la tabla `pg_group` table. Por ejemplo: `jolly=> insert into pg_group (groname, grosysid, grolist) jolly=> values ('posthackers', '1234', '{5443, 8261}')`; `INSERT 548224 jolly=> grant insert on foo to group posthackers; CHANGE jolly=>` Los campos de `pg_group` son: * `groname`: El nombre del grupo. Este campo debe de ser unicamente alfanumérico. No añadas subrayados u otros signos de puntuación. * `grosysid`: El id del grupo. El tipo del campo es `int4` y debe de ser único para cada grupo. * `grolist`: La lista de id de usuarios que pertenecen al grupo. Este campo es de tipo `int4`.

Asignar usuarios a los Grupos

Control de Acceso

`Postgres` proporciona mecanismos para permitir a los usuarios limitar el acceso que otros usuarios tendrán a sus datos.

SuperUsuarios de la Base de Datos

Los SuperUsuarios de la base de datos (aquellos que tienen el campo `pg_user . usesuper` activado) ignoran todos los controles de acceso descritos anteriormente con dos excepciones: las actualizaciones del catálogo del sistema no están permitidas si el usuario no tiene el campo `pg_user . usecatupd` activado, y nunca se permite la destrucción del catálogo del sistema (o la modificación de sus estructuras).

Privilegios de acceso

El uso de los privilegios de acceso para limitar la lectura, escritura y la puesta de reglas a las clases se trata en *`grant/revoke(1)`*.

Borrado de clases y modificación de estructuras.

Los comandos que borran o modifican la estructura de una clase, como **`alter`**, **`drop table`**, y **`drop index`**, solo funcionan con el propietario de la clase. Como hemos dicho antes, estas operaciones no están permitidas *nunca* en los catálogos del systema.

Funciones y Reglas

Las funciones y las reglas permiten a los usuarios insertar código en el servidor de la base de datos que otros usuarios pueden ejecutar sin saberlo. Ambos mecanismos permiten a los usuarios alojar *caballos de troya* con relativa impunidad. La única protección efectiva es un estrecho control sobre quien puede definir funciones (por ejemplo, escribir en tablas con campos SQL) y reglas. También se recomienda auditar seguimientos y alertas en `pg_class`, `pg_user` y `pg_group`.

Funciones

Las funciones escritas en cualquier lenguaje excepto SQL se ejecutan por el servidor de la base de datos con el mismo permiso que el usuario *postgres* (el servidor de la base de datos funciona con el user-id de *postgres*). Es posible cambiar las estructuras de datos internas del servidor por los usuarios, desde dentro de funciones de confianza. Es por ello que este tipo de funciones pueden, entre otras cosas, evitar cualquier sistema de control de acceso. Este es un problema inherente a las funciones definidas por los usuarios en C.

Reglas

Como en las funciones SQL, las reglas también se ejecutan con la identidad y los permisos del usuario que llamó al servidor de la base de datos.

Caveats

There are no plans to explicitly support encrypted data inside of Postgres (though there is nothing to prevent users from encrypting data within user-defined functions). There are no plans to explicitly support encrypted network connections, either, pending a total rewrite of the frontend/backend protocol.

User names, group names and associated system identifiers (e.g., the contents of `pg_user.usesysid`) are assumed to be unique throughout a database. Unpredictable results may occur if they are not.

Capítulo 24. Agregar y Eliminar Usuarios

`createuser` permite que usuarios específicos accedan a Postgres. `dropuser` elimina usuarios y previene que éstos accedan a Postgres.

Estas órdenes sólo afectan a los usuarios con respecto a Postgres; no tienen efecto en otros privilegios del usuario o en su estado con respecto al sistema operativo subyacente.

Capítulo 25. Gestión de Disco

Localizaciones Alternativas

Se puede crear una base de datos en una localización diferente a la establecida por defecto durante la instalación. Recuerde que todos los accesos a base de datos ocurren realmente a través del proceso en segundo plano, así que éste debe poder acceder a cualquier especificación.

Se crean localizaciones alternativas y referencias mediante una variable de entorno que da el path absoluto hasta la situación de almacenamiento deseada. Esta variable de entorno debe estar definida antes de que el proceso en segundo plano sea arrancado y debe ser modificable mediante la cuenta del administrador de postgres. Cualquier variable de entorno puede ser utilizada para referirse a una localización alternativa, si bien se recomienda la utilización de un nombre de variable con prefijo PGDATA para evitar confusión y conflicto con otras variables.

Nota: En versiones previas de Postgres, también estaba permitido utilizar un nombre de path absoluto para especificar una localización de almacenamiento alternativa. Se prefiere el método de especificación de variables de entorno, puesto que concede al administrador del sistema más flexibilidad en la gestión del almacenamiento en disco. Si prefiere utilizar paths absolutos, puede hacerlo definiendo "ALLOW_ABSOLUTE_DBPATHS" y recompilando Postgres. Para hacer esto, añada cualquiera de estas líneas

```
#define ALLOW_ABSOLUTE_DBPATHS 1
```

al archivo `src/include/config.h`, o especifique

```
CFLAGS+= -DALLOW_ABSOLUTE_DBPATHS
```

en su `Makefile.custom`.

Recuerde que la creación de una base de datos la ejecuta realmente un proceso de la base de datos en segundo plano. Por lo tanto, cualquier variable de entorno que especifique una localización alternativa debe ser definida antes de que el proceso en segundo plano sea arrancado. Para definir una localización alternativa apuntando a PGDATA2 `/home/postgres/data`, primero escriba

```
% setenv PGDATA2 /home/postgres/data
```

para definir la variable de entorno que será utilizada con las órdenes siguientes. Normalmente, querrá definir esta variable en el fichero de inicialización del super usuario de Postgres, `.profile` o `.cshrc` para asegurar que está definido al arrancar el sistema. Se puede utilizar cualquier variable de entorno para referirse a una localización alternativa, aunque se prefiere que las variables estén prefijadas con "PGDATA" para eliminar confusiones y la posibilidad de conflictos con otras variables, o su reescritura.

Para crear un área de almacenamiento de datos en PGDATA2, asegúrese de que `/home/postgres` ya existe y puede ser escrito por el administrador de postgres. Después desde la línea de órdenes, escriba

```
% setenv PGDATA2 /home/postgres/data
% initlocation $PGDATA2
```

```
Creating Postgres database system directory /home/postgres/data  
Creating Postgres database system directory /home/postgres/data/base
```

Para comprobar la nueva localización, cree una base de datos test escribiendo

```
% createdb -D PGDATA2 test  
% dropdb test
```

Capítulo 26. Gestión de una base de datos

Si el programa `postmaster` de Postgres está cargado y en ejecución, podemos crear algunas bases de datos con las que experimentar. En este documento describiremos las órdenes básicas para gestionar una base de datos.

Creación de una base de datos

Supongamos que quiere crear una base de datos llamada `mibase`. Puede hacerlo con el siguiente orden:

```
% createdb nombredb
```

Postgres le permite crear cualquier número de bases de datos en una máquina dada, y usted se convierte automáticamente en el administrador de la base de datos que acaba de crear. Los nombres de las bases de datos han de comenzar por un carácter alfabético, y su longitud está limitada a 31 caracteres. No todos los usuarios están autorizados para convertirse en administradores de bases de datos. Si Postgres rechaza la orden de crear bases de datos, es que necesita que el administrador del sistema le garantice derechos para crear las bases de datos. Consulte a su administrador de sistemas si le ocurre eso.

Acceso a la base de datos

Una vez que ha construido la base de datos, puede acceder a ella por los siguientes medios:

- Ejecutando el programa monitor de terminal de Postgres (`psql`) que le permite introducir, editar y ejecutar órdenes SQL de un modo interactivo.
- Escribiendo un programa en C que use la biblioteca de rutinas `libpq`. Esto le permite enviar órdenes SQL desde C y obtener las respuestas y mensajes de estado en su programa. Esta interfaz se discute en el documento *Guía de programación en PostgreSQL*.

Puede que quiera ejecutar el programa `psql`, para probar los ejemplos de este manual. Puede activarlo para la base de datos `nombredb` escribiendo la orden:

```
% psql nombredb
```

Recibirá como respuesta el siguiente mensaje:

```
Welcome to the Postgres interactive sql monitor:
```

```
type \? for help on slash commands
type \q to quit
type \g or terminate with semicolon to execute query
You are currently connected to the database: nombredb
```

```
nombredb=>
```

Este indicativo le informa de que el monitor de terminal se encuentra dispuesto y que puede escribir consultas SQL en el espacio de trabajo creado por el citado monitor de terminal. El programa `psql` responde a códigos de escape que comienzan con la barra invertida, `"\"`. Por ejemplo, puede obtener ayuda sobre la sintaxis de varias órdenes SQL de Postgres escribiendo:

```
nombredb=> \h
```

Una vez que ha terminado de introducir sus consultas en el espacio de trabajo, puede pasar el contenido de éste al servidor de Postgres escribiendo:

```
nombredb=> \g
```

Esto le dice al servidor que procese la consulta. Si termina su consulta con punto y coma, no es necesario que introduzca la secuencia `\g`. `psql` procesará automáticamente consultas terminadas en punto y coma. Para leer las consultas de un fichero, en lugar de introducirlas interactivamente, escriba:

```
nombredb=> \i fichero
```

Para salir de `psql` y volver a Unix, escriba:

```
nombredb=> \q
```

y `psql` terminará y le devolverá el intérprete de órdenes. (Para conocer más órdenes de escape, escriba `\h` en la entrada del monitor.) Los espacios en blanco (espacios, tabuladores y saltos de línea) pueden usarse libremente en las consultas SQL. Los comentarios en una sola línea se indican mediante dos guiones (`"--"`). Todo lo que vaya desde los guiones hasta el fin de la línea será ignorado. Los comentarios que abarcan múltiples líneas, o que están dentro de una línea se encierran entre `"/* ... */"`, una convención que se tomó prestada de Ingres.

Destrucción de una base de datos

Si usted es el administrador de la base de datos `mibase`, puede destruirla usando el siguiente orden de Unix:

```
% dropdb nombredb
```

Esta acción elimina físicamente todos los ficheros asociados con la base de datos y no puede ser invertida, por lo que sólo debe ser usada con gran precaución.

Es también posible destruir una base de datos desde una sesión SQL usando:

```
> drop database nombredb
```

Copia de seguridad y restauración

Atención

Deben realizarse copias de seguridad de las bases de datos regularmente. Dado que Postgres gestiona sus propios ficheros en el sistema, *no se recomienda* confiar en los sistemas de copia de seguridad del sistema para las copias de respaldo de las bases de datos; no hay garantía de que los ficheros estén en un estado consistente que permita su uso después de la restauración.

Postgres proporciona dos utilidades para realizar las copias de seguridad de su sistema: `pg_dump` para copias de seguridad de bases de datos individuales y `pg_dumpall` para realizar copias de seguridad de toda la instalación de una sola vez.

La copia de seguridad de una sola base de datos puede realizarse usando la siguiente orden:

```
% pg_dump nombredb > nombredb.pgdump
```

y puede ser restaurada usando

```
cat nombredb.pgdump | psql nombredb
```

Esta técnica puede usarse para mover bases de datos a una nueva localización y para renombrar bases de datos existentes..

Bases de datos grandes

Autor: Escrito por Hannu Krosing¹ on 1999-06-19.

Dado que Postgres permite tablas de mayor tamaño que el permitido por el sistema de ficheros, puede resultar problemático el volcado de una tabla a un fichero, ya que el fichero resultante seguramente superará el tamaño máximo permitido.

Como `pg_dump` escribe en stdout, puede usar las herramientas *nix para sortear estos posibles problemas:

- Uso de volcados comprimidos:

```
% pg_dump nombredb | gzip > nombrefichero.dump.gz
```

la recuperamos con:

```
% createdb nombredb
% gunzip -c nombrefichero.dump.gz | psql nombredb
```

```
o
```

```
% cat nombrefichero.dump.gz | gunzip | psql nombredb
```

- Use split:

```
% pg_dump nombredb | split -b 1m - nombrefichero.dump.
```

y lo recuperamos con:

```
% createdb nombredb  
% cat nombrefichero.dump.* | psql nombredb
```

Por supuesto, el nombre del fichero (*nombrefichero*) y el contenido de la salida de `pg_dump` no tiene por qué coincidir con el nombre de la base de datos. Además, la base de datos restaurada puede tener un nombre distinto, por lo que este mecanismo también es efectivo para renombrar bases de datos.

Notas

1. hannu@trust.ee

Capítulo 27. Tratamiento de problemas

Fallos de inicio de Postmaster

Hay varias posibles razones para que postmaster no pueda inicializarse. Compruebe el fichero de registro de postmaster, o inícielo manualmente (sin redirigir la salida estándar o la de errores) para ver los mensajes que aparecen. Alguno de los posibles mensajes de error son autoexplicativos, pero los hay que pueden no serlos tanto:

```
FATAL: StreamServerPort: bind() failed: Address already in use
       Is another postmaster already running on that port?
```

Esto normalmente significa lo que sugiere: accidentalmente ha iniciado una segunda instancia de postmaster en el mismo puerto en el que ya se está ejecutando uno. Sin embargo, si el mensaje de error del núcleo no es "Address already in use" o alguna variante, puede estar ocurriendo otro problema. Por ejemplo, el tratar de iniciar una sesión de postmaster en un puerto de error reservado puede producir algo como:

```
$ postmaster -i -p 666
FATAL: StreamServerPort: bind() failed: Permission denied
       Is another postmaster already running on that port?
```

```
IpcMemoryCreate: shmget failed (Invalid argument) key=5440001, size=83918612, per-
mission=600
FATAL 1: ShmemCreate: cannot create region
```

Un mensaje como éste posiblemente indica que el límite impuesto al tamaño de las zonas de memoria compartidas es menor que área de «buffer» que Postgres está intentando crear. (O puede significar que no dispone de soporte para la memoria compartida de tipo SysV configurado en su núcleo.) Como arreglo temporal puede tratar de iniciar postmaster con un número de «buffers» menor de lo normal (parámetro -B). Sin embargo, debería reconfigurar su núcleo para incrementar el tamaño permitido para la memoria compartida. Este mensaje puede aparecer cuando trate de iniciar varias sesiones de postmaster en la misma máquina, si el total de espacio necesario excede el límite impuesto por el núcleo.

```
IpcSemaphoreCreate: semget failed (No space left on device) key=5440026, num=16, per-
mission=600
```

Un mensaje como éste *no* significa que se haya quedado sin espacio en el disco; significa que la cantidad máxima de semáforos permitidos por el núcleo para el SysV es menor que la cantidad que Postgres intenta crear. Como antes, puede evitar este problema iniciando el postmaster con un número de procesos «backend» menor (parámetro -N), pero sería mejor que incrementara el límite impuesto por el núcleo.

Problemas con la conexión del Cliente

Una vez que tiene el postmaster en ejecución, al tratar de conectar con él mediante una aplicación cliente puede producirse un fallo por varias razones. Los ejemplos de

mensajes de error mostrados aquí son para clientes basados en las versiones recientes de libpq; los clientes basados en otras bibliotecas de interfaz pueden producir otros mensajes, con más o menos información.

```
connectDB() - connect() failed: Connection refused
Is the postmaster running (with -i) at 'server.joe.com' and accepting con-
nections on TCP/IP port '5432'?
```

Este es el fallo genérico de 'No puedo encontrar un postmaster con el que comunicarme'. Puede ocurrir algo así cuando se intenta una comunicación TCP/IP o mediante socket Unix con un postmaster local:

```
connectDB() - connect() failed: No such file or directory
Is the postmaster running at 'localhost' and accepting connections on Unix soc-
ket '5432'?
```

La última línea es útil para verificar que el cliente está tratando de conectar donde se supone que debe hacerlo. Si en realidad no hay ningún postmaster ejecutándose allí, el mensaje de error del núcleo será del tipo de 'Conexión rehusada' o de 'No existe el fichero o directorio', como los anteriores. (Es particularmente importante tener en cuenta que 'Conexión rehusada' en este contexto *no* significa que el postmaster haya recibido la petición de conexión y la haya rechazado; en este caso se produce un mensaje diferente, como se verá.) Otros mensajes de error, como el de "Connection timed out" sí indican problemas más importantes, como la falta de conectividad en la red.

```
No pg_hba.conf entry for host 123.123.123.123, user joeblow, database testdb
```

Esto es lo más probable que obtenga si consigue contactar con un postmaster, pero éste no quiere hablar con usted. Como sugiere el mensaje, el postmaster rehúsa la petición de conexión porque no encuentra un renglón de autorización en su fichero de configuración pg_hba.conf

```
Password authentication failed for user 'joeblow'
```

Los mensajes como éste indican que ha contactado con el postmaster, y éste está dispuesto a hablar con usted, pero no lo hará hasta que supere el método de autorización especificado en el fichero pg_hba.conf. Compruebe la clave que está enviando, o su programa IDENT o Kerberos, si el mensaje de error menciona alguno de esos tipos de autenticación.

```
FATAL 1: SetUserId: user 'joeblow' is not in 'pg_shadow'
```

Esta es otra variante de fallo de autenticación: no se ha ejecutado la orden de Postgres 'create_user' para el nombre de usuario indicado.

```
FATAL 1: Database testdb does not exist in pg_database
```

No hay base de datos con ese nombre bajo el control de ese postmaster. Nótese que si no especifica el nombre de la base de datos, se aplica por defecto su nombre de usuario en Postgres, lo que puede no ser lo correcto.

Depuración de mensajes

El `postmaster` presenta ocasionalmente mensajes que pueden ser de ayuda en la solución de problemas. Si desea ver mensajes de depuración de `postmaster`, puede iniciarlo con la opción `-d` y redirigir la salida a un fichero de registro:

```
% postmaster -d > pm.log 2>&1 &
```

Si no desea ver estos mensajes, puede escribir

```
% postmaster -S
```

y el `postmaster` entrará en modo 'S'ilencioso. Nótese que no se incluye el símbolo `'&'` en el último ejemplo, ya que el `postmaster` se ejecutará en segundo plano.

pg_options

Nota: Contribución de Massimo Dal Zotto¹

El fichero opcional `data/pg_options` contiene opciones de ejecución usadas por el backend para controlar mensajes de ejecución y otros parámetros ajustables. Lo que hace interesante a este fichero es el hecho de que es releído por el backend cuando recibe una señal `SIGHUP`, haciendo así posible cambiar opciones de ejecución sin tener que reiniciar Postgres. Las opciones especificadas en este fichero pueden incluir puntos de depuración usados por el paquete `trace` (`backend/utls/misc/trace.c`) o parámetros numéricos que puede usar el backend para controlar su comportamiento. Se pueden definir nuevas opciones y parámetros en `backend/utls/misc/trace.c` y en `backend/include/utls/trace.h`.

Las opciones de `pg_option` pueden especificarse con el parámetro `-T` de Postgres:

```
postgres opciones -T "verbose=2,query,hostlookup-"
```

Las funciones usadas para imprimir errores y mensajes de depuración pueden ahora usar la utilidad `syslog(2)`. Los mensajes impresos en `stdout` o `stderr` son precedidos por una etiqueta informativa que incluye la fecha y hora y el pid del backend:

```
#timestamp          #pid    #message
980127.17:52:14.173 [29271] StartTransactionCommand
980127.17:52:14.174 [29271] ProcessUtility: drop table t;
980127.17:52:14.186 [29271] SIIncNumEntries: table is 70% full
980127.17:52:14.186 [29286] Async_NotifyHandler
980127.17:52:14.186 [29286] Waking up sleeping backend process
980127.19:52:14.292 [29286] Async_NotifyFrontEnd
980127.19:52:14.413 [29286] Async_NotifyFrontEnd done
980127.19:52:14.466 [29286] Async_NotifyHandler done
```

Este formato mejora la legibilidad de los registros, y permite comprender qué «backend» concreto está haciendo qué y en qué momento. También hace más fácil escribir guiones (scripts) en awk o perl que monitoricen el fichero de registro para detectar errores o problemas en la base de datos, o para contabilizar estadísticas temporales de las transacciones.

Los mensajes impresos por syslog usan la utilidad de registro LOG_LOCAL0. El uso de syslog puede ser controlado por las opciones referentes a él en syslog. Desgraciadamente, muchas funciones llaman directamente a `printf()` para mostrar sus mensajes en stdout o stderr y esta salida no puede ser redirigida a syslog o incluir información sobre fecha y hora. Sería aconsejable que todas las llamadas a `printf` pudieran ser reemplazadas por la macro `PRINTF` y la salida a stderr se cambiaran para que usaran `EPRINTF` en su lugar, de modo que se pudieran controlar todas las salidas de un modo uniforme.

El formato del fichero `pg_options` es como sigue:

```
# comentario
opción=valor_entero # set value for opción
opción              # set opción = 1
opción+             # set opción = 1
opción-             # set opción = 0
```

Nótese que *palabra_clave* puede ser también una abreviación del nombre de opción definido en `backend/utls/misc/trace.c`.

Véase *Usando pg_options* para una lista completa de las opciones y sus posible valores.

Notas

1. <mailto:dz@cs.unitn.it>

Capítulo 28. Recuperación de bases de datos

Esta sección necesita ser escrita. ¿Algún voluntario?

Capítulo 29. Pruebas de regresión

Instrucciones y análisis del test de regresión

Los tests de regresión de PostgreSQL son un conjunto completo de pruebas para la implementación de SQL embebidos en PostgreSQL. Realizan pruebas tanto sobre operaciones SQL estándar, como también sobre las capacidades añadidas por PostgreSQL.

Los tests de regresión fueron desarrollados originalmente por Jolly Chen y Andrew Yu, y fueron extensamente repasados/reempaquetados por Fournier y Thomas Lockhart. Para PostgreSQL v6.1 en adelante los tests de regresión forman parte de cada release oficial.

Algunas bases de datos PostgreSQL correctamente instaladas y totalmente funcionales pueden fallar en alguno de estos test de regresión debido a problemas con la representación del punto flotante y el soporte de zona horaria. Los tests actuales son evaluados usando un sencillo algoritmo "diff", y son muy sensibles a pequeñas diferencias en el sistema. Para tests aparentemente fallidos, si se examinan estas diferencias, pueden revelar no ser significativas.

Las notas sobre tests de regresión de abajo asumen lo siguiente (excepto en casos indicados):

- Las instrucciones son compatibles con Unix. Vea la nota abajo.
- Se usan las opciones por defecto excepto donde se indica.
- El usuario postgres es el superusuario Postgres.
- La ruta de las fuentes es /usr/src/pgsql (son posibles otras rutas).
- La ruta de los ejecutables es /usr/local/pgsql (son posibles otras rutas).

Entorno de regresión

Para preparar los tests de regresión, haga **make all** en el directorio de los tests de regresión. Esto compila un programa C con funciones extendidas PostgreSQL en un librería compartida. Se generan algunos guiones (scripts) SQL localizados y archivos de salida comparativos para los tests que los necesiten. La localización reemplaza macros en los archivos de fuentes con rutas absolutas y nombres de usuario.

Normalmente, los tests de regresión deben ser ejecutados por el usuario postgres ya que el directorio 'src/test/regress' y subdirectorios son de su propiedad. Si ejecuta los test de regresión con otro usuario el directorio 'src/test/regress' debe tener permisos de escritura para ese usuario.

Antes era estrictamente necesario ejecutar el postmaster con la zona horaria del sistema establecida en PST, pero ya no es necesario. Puede ejecutar los tests de regresión sobre su configuración habitual del postmaster. El guión (script) del test establecerá la variable de entorno PGTZ para asegurar que los tests dependientes de la zona horaria produzcan los resultados esperados. De todas formas, su sistema debe proporcionar librerías de soporte para la zona horaria PST8PDT, o los tests dependientes de la zona horaria fallarán. Para comprobar que su equipo soporta esto, escriba lo siguiente:

```
setenv TZ PST8PDT
date
```

La orden "date" de arriba tiene que devolver la hora actual del sistema en la zona horaria PST8PDT. Si la base de datos PST8PDT no está disponible, entonces el sistema tiene que devolver la hora en GMT. Si la zona horaria PST8PDT no está disponible, puede establecer las reglas para esa zona horaria explícitamente.

```
setenv PGTZ PST8PDT7,M04.01.0,M10.05.03
```

Estructura de directorios

Nota: Esto debería ser una tabla en la sección anterior.

```
input/ .... .Archivos fuente que son convertidos, usando 'make all', en
             alguno de los archivos .sql en el subdirectorío 'sql'

output/ ... .Archivos fuente que son convertidos, usando 'make all', en
           archivos .out en el subdirectorío 'expected'

sql/ ..... .Archivos sql usados para ejecutar los tests de regresión

expected/ . .Archivos .out que representan lo que *esperamos* que parezcan
            los resultados

results/ .. .Archivos .out que contienen lo que los resultados *realmente*
            parecen. Además es usado como almacén temporal para el test de
            copia de tablas.
```

Procedimiento para el test de regresión

Las instrucciones están probadas en un RedHat Linux versión 4.2 usando el intérprete de órdenes bash. Excepto donde se indique, seguramente funcione en la mayoría de sistemas. Instrucciones como `ps` y `tar` cambian mucho las opciones que debe usar en cada plataforma. *Use el sentido común* antes de escribir estas instrucciones.

Para una instalación nueva o si está actualizando una versión anterior de Postgres:

Configuración de la Regresión de Postgres

1. El archivo `/usr/src/pgsql/src/test/regress/README` tiene instrucciones detalladas para la ejecución e interpretación de los tests de regresión. Lo que sigue es una versión más corta:

Si el postmaster no se está ejecutando ya, inicie el postmaster en una ventana que esté disponible escribiendo

```
postmaster
```


o inicie el demonio postmaster en segundo plano escribiendo

```
cd
nohup postmaster > regress.log 2>&1 &
```

Ejecute postmaster desde la cuenta de superusuario de Postgres (normalmente la cuenta postgres).

Nota: No ejecute `postmaster` desde la cuenta de root.

2. Si ha ejecutado anteriormente los tests de regresión, borre el directorio de trabajo con:

```
cd /usr/src/pgsql/src/test/regress
gmake clean
```

No necesita escribir "gmake clean" si es la primera vez que está ejecutando los tests.

3. Ejecute los tests de regresión. Escriba

```
cd /usr/src/pgsql/src/test/regress
gmake all
```

4. Ejecute los tests de regresión. Escriba

```
cd /usr/src/pgsql/src/test/regress
gmake runtest
```

5. Debería obtener en la pantalla (y además escrito en el archivo `./regress.out`) una serie de líneas indicando qué tests han pasado y qué tests han fallado. Tenga en cuenta que puede ser normal que alguno de los tests falle. Para los tests fallidos, use `diff` para comparar los archivos de los directorios `./results` y `./expected`. Si falla `float8`, escriba algo como esto:

```
cd /usr/src/pgsql/src/test/regress
diff -w expected/float8.out results
```

6. Después de ejecutar los tests y examinar los resultados, escriba

```
dropdb regression
cd /usr/src/pgsql/src/test/regress
gmake clean
```

para recuperar el espacio en disco temporal usado por los tests.

Análisis de Regresión

Los resultados se encuentran en los archivos del directorio `./results`. Estos resultados pueden ser comparados con los resultados del directorio `./expected` usando `'diff'`. (El guión (script) del test hace esto por usted, y deja las diferencias en `./regression.diffs`.)

Los archivos pueden no corresponderse de forma exacta. El guión del test informará de una diferencia como "failure" (fallo), pero la diferencia puede deberse a pequeñas variaciones entre plataformas en los mensajes de error, comportamiento de la librería matemática, etc. "Fallos" de este estilo no indican necesariamente un problema con Postgres.

Por tanto, es necesario examinar las diferencias de cada test "fallido" con el fin de determinar si existe un problema realmente. Los siguientes puntos intentan proporcionar una guía para determinar si una diferencia es significativa o no.

Diferencias en los mensajes de error

Alguno de los tests de regresión incluyen intencionadamente valores de entrada no válidos. Los mensajes de error pueden venir tanto del código de Postgres como de las rutinas de sistema de la plataforma en la que nos encontremos. En el último caso, los mensajes pueden variar entre plataformas, pero deben reflejar información similar. Estas diferencias en los mensajes darán como resultado un test "fallido" que puede ser validado mediante una inspección.

Diferencias en fechas y horas

Muchos de los resultados de fecha y hora son dependientes del entorno de la zona horaria. Los archivos de referencia están generados para la zona horaria PST8PDT (Berkeley, California) y aparentemente pueden parecer fallos si los tests no se ejecutan con esta zona horaria establecida. El programa que ejecuta los tests de regresión establece la variable de entorno PGTZ a PST8PDT para asegurar resultados parecidos.

Parece que algunos sistemas no aceptan la sintaxis recomendada para establecer explícitamente las reglas de la zona horaria local; puede ser que necesite usar una forma distinta para establecer PGTZ en estas máquinas.

Algunos sistemas que usan librerías antiguas de zonas horarias fallan al aplicar las correcciones de ahorro de luz solar en las fechas anteriores a 1970, causando que las horas de esas fechas aparezcan en PST a pesar de todo. Esto dará pie a diferencias localizadas en los resultados de los tests.

Diferencias en punto flotante

Algunos de los tests implican calcular números de 64-bits (float8) a partir de las columnas de una tabla. Se han observado diferencias en los resultados que devuelven funciones matemáticas en columnas de tipo float8. Los tests con float8 y de geometría son particularmente propensos a pequeñas diferencias entre plataformas. Se precisa una comparación con lupa por parte humana para determinar diferencias que normalmente se encuentran 10 posiciones a la derecha del punto decimal.

Algunos errores de señales del sistema con `pow()` y `exp()` difieren de los mecanismos que espera el actual código de Postgres.

Diferencias en polígonos

Varios de los tests incluyen operaciones con coordenadas sobre el callejero de Oakland/Berkley CA. Los datos de este mapa vienen expresados como polígonos cuyos vértices están representados en pares de números float8 (latitud y longitud decimal). Inicialmente, se crean y llenan algunas tablas con coordenadas, después se crean algunas vistas (Views) haciendo el Join de dos tablas usando el operador de intersección de polígonos (##), y después se realiza un Select sobre la vista. Cuando comparamos los resultados de diferentes plataformas, las diferencias aparecen en el segundo o tercer lugar a la derecha del punto decimal. Las instrucciones SQL donde se dan estos problemas son las siguientes:

```
QUERY: SELECT * from street;
QUERY: SELECT * from iexit;
```

Diferencias aleatorias

Hay al menos un caso de test en random.out que esta diseñado para producir resultados aleatorios. Esto causa que random falle el test de regresión cada vez. Escribir

```
diff results/random.out expected/random.out
```

debe producir una o unas pocas líneas de diferencias por esta razón, pero otras variaciones en punto flotante o en arquitecturas distintas pueden causar más diferencias.

Los archivos “expected”

Los archivos ./expected/*.out fueron adaptados del monolítico archivo original expected.input proporcionado por Jolly Chen. Versiones más modernas de estos archivos generadas en varias máquinas de desarrollo han sido sustituidas después de una cuidadosa (?) inspección. Muchas de estas máquinas de desarrollo están ejecutando variantes del Unix OS (FreeBSD, Linux, etc) en hardware Ix86. El archivo original expected.input fue creado en un sistema SPARC Solaris 2.4 usando el código de postgres5-1.02a5.tar.gz. Fue comparado con un archivo creado en un sistema I386 Solaris 2.4 y las diferencias fueron solamente en los polígonos de punto flotante en el tercer dígito a la derecha del punto decimal. (vea más arriba) El archivo original sample.regress.out se obtuvo de la entrega 1.01 de postgres construida por Jolly Chen y se incluye aquí para referencia. Tendría que haberse ejecutado con una máquina DEC ALPHA ya que el Makefile.global en la version 1.01 de postgres tiene PORTNAME=alpha.

Archivos de comparación específicos de la plataforma

Como alguno de los tests producen resultados inherentes a la plataforma usada, hemos proporcionamos una forma para suplir los archivos de comparación específicos para cada plataforma. Frecuentemente se da la misma variación en múltiples plataformas; en vez de dar un archivo de comparación separado para cada plataforma, existe un archivo guía que define qué archivo de comparación usar. De forma que, para eliminar fallos tontos de una plataforma en particular, debe elegir o crear un

fichero de resultados variantes, y añadir una línea al archivo guía, que es "mapa de resultados".

Cada línea del archivo guía es de la siguiente forma

```
testname/platformnamepattern=comparisonfilename
```

El nombre del test (testname) es sencillamente el nombre del módulo de regresión de ese test en particular. El patrón del nombre de la plataforma (platformnamepattern) está generado al estilo de `expr(1)` (que es una expresión regular con el símbolo `^` implícito al principio). Esta se comprueba con el nombre de la plataforma tal como viene escrito en `config.guess`. El nombre del fichero de comparación (comparisonfilename) es el nombre del sustituto del fichero de resultados de comparación.

Por ejemplo: el test de regresión `int2` incluye una entrada deliberada de un valor que es demasiado largo para caber en un `int2`. El mensaje de error específico que es producido es dependiente de la plataforma; nuestra plataforma de referencia saca

```
ERROR:  pg_atoi: error reading "100000": Numerical result out of range
```

pero en un buen número de otras plataformas Unix saca

```
ERROR:  pg_atoi: error reading "100000": Result too large
```

En este caso, proporcionamos una variante del archivo de comparación, `int2-too-large.out`, que incluye la sintaxis de este mensaje de error. Para no mostrar estos "fallos" tontos en las plataformas HPPA, el `resultmap` (mapa de resultados) incluye

```
int2/hppa=int2-too-large
```

que se activará en cualquier máquina en el que la salida de `config.guess` comience por `'hppa'`. Otras líneas en el `resultmap` seleccionan la variante del archivo de comparación para otras plataformas donde sea apropiado.

Capítulo 30. Notas de versiones

Version 6.5.3

Esta es basicamente una limpieza de la version 6.5.2. Hemos añadido un nuevo pgacces que se perdio en la 6.5.2, e instalado una correccion especifica para NT.

Migracion a v6.5.3

No se requiere un dump/restores para aquellos que esten ejecutando una 6.5.*.

Lista Detallada de Cambios

Version actualizada de pgacces 0.98
Parche especifico para NT
Correccion para reglas de volcado en tablas heredadas

Version 6.5.2

Esta es basicamente una limpieza de la version 6.5.1. Hemos corregio una variedad de problemas reportados por usuarios de 6.5.1.

Migracion to v6.5.2

No se requiere un dump/restores para aquellos que esten ejecutando una 6.5.*.

Lista Detallada de Cambios

corregidas las subselect+CASE (Tom)
Añadida configuracion de SHLIB_LINK para los portes de solaris_i386 y solaris_sparc(Daren Sefcik)
Correcciones para CASE y WHERE en clausulas "join"(Tom)
Correccion para aborto en BTScan(Tom)
Reparada la comprobacion para UNIQUE redundante e indices PRIMARY KEY(Thomas)
Mejorado para que se compruebe las restricciones en multi-columns(Thomas)
Correccion para Win32 que tenia problemas con MB habilitado(Hiroki Kataoka)
Permite a yacc de BSD y a bison compilar codigo pl(Bruce)
Corregido el trabajo con SET NAMES
corregidos los int8 (Thomas)
Correccion del consumo de memoria de "vacuum"(Hiroshi,Tatsuo)
Reduccion del consumo total de memoria de "vacuum"(Tom)
Correccion para timestamp(datetime)
Correcciones de problemas en las reglas de paso al analizador sintactico(Tom)
Correccion del problema de cuotas en mkMakefile.tcldefs.sh.in y mkMakefile.tkdefs.sh.in
This is to re-use space on index pages freed by vacuum(Vadim)
documentado -x para pg_dump(Bruce)
Correcin para operadores unarios en la regla de paso al analizador sintactico(Tom)
Comentado el FileUnlink de exceso de segmentos durante mdtruncate() (Tom)
Enlazamiento en Irix corregido por Yu Cao >yucao@falcon.kla-tencor.com<

Reparado el error logico en LIKE: no debia devolver un LIKE_ABORT cuando alcanza el final de un patron antes del final del texto(Tom)
 Reparada limpieza incorrecta de montones de memoria reservadas durante aborto de transaccion(Tom)
 Version actualizada de pgaccess 0.98

Version 6.5.1

Esta es basicamente una limpieza de la version 6.5. Hemos corregio una variedad de problemas reportados por usuarios de 6.5.

Migracion to v6.5.1

No se requiere un dump/restores para aquellos que esten ejecutando una 6.5.

Lista Detallada de Cambios

Añadido un fichero NT LEAME
 Correcciones de portabilidad para linux_ppc, Irix, linux_alpha, OpenBSD, alpha
 Eliminado QUERY_LIMIT, utilizar SELECT...LIMIT
 Correccion para EXPLAIN en herencia(Tom)
 Parche para permitir "vacuum" en tablas con multi-segmentos(Hirosi)
 Correccion para la selectividad del optimizador R-Tree(Tom)
 Corregida laguna el descriptor de ficheros ACL(Atsushi Ogawa)
 Nueva expresion del codigo de sub arboles(Tom)
 Se evitan escrituras en disco para transacciones de solo-lectura(Vadim)
 Correccion para eliminacion de tablas temporales si la ultima transaccion fue abortada (Bruce)
 Correccion para prevenir que sean creadas tuplas demasiado largas(Bruce)
 correcciones en plpgsql
 Se permiten numeros de puerto de 32k - 64k(Bruce)
 Añadido ^ precedence(Bruce)
 Renombrados ficheros ordendo llamados pg_temp a pg_sorttemp(Bruce)
 Correccion para microsegundos en valores temporales(Tom)
 Limpieza de la fuente del Tutorial
 Nuevo porte a linux_m68k
 Correccion para la ordenacion de NULL's en algunos casos(Tom)
 Corregidas las dependencias para librerias compartidas(Tom)
 Corregido fallos tecnicos que afectaban a GROUP BY en subselects(Tom)
 Correccion para algunas alarmas del compilador (Tomoaki Nishiyama)
 Añadido soporte para Win1250 (Checo) (Pavel Behal)

Version 6.5

Esta version marca un avance grande en el conocimiento que el equipo de desarrollo tiene del codigo fuente que heredamos de Berkeley. Veras que podemos añadir mayores features mas facilmente, gracias al incremento en tamaño y experiencia de nuestro mundialmente extenso equipo de desarrollo.

He aquí un conciso resumen de los más notables cambios:

Control de concurrencia multi-version (MVCC en inglés Multi-version concurrency control)

Esto elimina nuestro viejo bloqueo a nivel de tabla, y lo reemplaza con un bloqueo del sistema que es superior a la mayoría de los sistemas de bases de datos comerciales. En un sistema tradicional, cada registro que es modificado se bloquea hasta que se confirma la transacción, previniendo lecturas por otros usuarios. MVCC utiliza de modo natural el carácter multi-version de PostgreSQL para permitir que las lecturas continúen leyendo datos consistentes durante la actividad de escritura. Las escrituras continúan utilizando el sistema de transacciones compacto `pg_log`. Todo esto se realiza sin tener que reservar un bloqueo para cada registro como en los sistemas tradicionales de base de datos. Así que, básicamente, ya no estaremos restringidos por más tiempo por el bloqueo simple a nivel de tabla; tenemos algo mejor que el bloqueo a nivel de registro.

Copias de seguridad en caliente con `pg_dump`

`pg_dump` se beneficia de las nuevas prestaciones de MVCC para dar consistencia a un `dump`/backup mientras la base de datos permanece en línea y disponible para ser consultada.

Tipos de datos numéricos

Ahora tenemos tipos de datos verdaderamente numéricos, con precisión especificada por el usuario.

Tablas temporales

Se garantiza que las tablas temporales tienen nombres únicos durante una sesión en la base de datos, y que son destruidas al salir de la sesión.

Nuevas prestaciones SQL

Ahora tenemos soporte para declaraciones `CASE`, `INTERSECT`, and `EXCEPT`. Tenemos nuevos `LIMIT/OFFSET`, `SET TRANSACTION ISOLATION LEVEL`, `SELECT ... FOR UPDATE`, y un mejorado comando `LOCK TABLE`.

Aceleramiento

Continuamos acelerando PostgreSQL, gracias a la variedad de talentos que hay en nuestro equipo. Hemos acelerado la asignación de memoria, la optimización, las uniones de tablas (`table join`), y las rutinas de transferencias de registros.

Portes

Continuamos ampliando nuestra lista de portes, esta vez incluimos WinNT/i386 y NetBSD/arm32.

Interfaces

La mayoría de interfaces tienen una nueva versión, y la funcionalidad existente ha sido mejorada.

Documentación

Nuevo y actualizado material está presente por toda la documentación. Se han aportado nuevas FAQs para las plataformas SGI y AIX. El *Tutorial* tiene información introductoria sobre SQL de Stefan Simkovics. Para la *Guía del Usuario*, hay páginas de referencia cubriendo la utilidad `postmaster` y más programas de

utilidad, un apéndice nuevo contiene detalles sobre el comportamiento de `date/time`. La *Guía del Administrador* tiene un nuevo capítulo sobre resolución de problemas de Tom Lane. Y la *Guía del Programador* tiene una descripción del proceso de interrogación, también de Stefan, y detalles acerca de cómo obtener el árbol del código fuente de Postgres por CVS anónimo y CVSup.

Migración to v6.5

Un `dump/restore` utilizando `pg_dump` es necesario para aquellos que deseen migrar datos de cualquier versión previa de Postgres. `pg_upgrade` *not* puede ser utilizado para actualizar esta versión porque la estructura en disco de las tablas ha cambiado comparada con versiones precedentes.

La nueva característica de Control de Concurrencia Multi-Version (MVCC, en inglés) puede dar comportamientos un poco diferentes en entornos multiusuarios. *Lea y comprenda la sección siguiente para asegurar que sus aplicaciones existentes le proporcionarán el comportamiento que necesita.*

Control de Concurrencia Multi-Version

A causa de que las lecturas en 6.5 no bloquean los datos, a pesar del nivel de aislamiento de transacción, los datos leídos por una transacción pueden ser sobre escritos por otra. En otras palabras, si un registro es devuelto por **SELECT** eso no significa que este registro exista realmente en el momento en que es devuelto. (i.e algunas veces después de que la sentencia o la transacción comience) ni tampoco que el registro este protegido de ser borrado o actualizado por una transacción en concurrente antes de que la transacción en curso haga `commit` o `rollback`.

Para asegurar la existencia actual de un registro y protegerlo contra actualizaciones concurrentes se debe utilizar **SELECT FOR UPDATE** o una sentencia **LOCK TABLE** apropiada. Esto debería ser tenido en cuenta cuando se porten aplicaciones desde versiones precedentes de Postgres y otros entornos.

Tenga todo lo anterior en mente su utiliza `triggers contrib/refint.*` para integridad referencial. Ahora se requieren técnicas adicionales. Un modo es utilizar el comando **LOCK parent_table IN SHARE ROW EXCLUSIVE MODE** si una transacción va a actualizar/borrar una clave primaria y utilizar el comando **LOCK parent_table IN SHARE MODE** si una transacción va a actualizar/insertar una clave foránea.

Nota: Notese que si ejecuta una transacción en modo **SERIALIZABLE** entonces debe ejecutar el comando **LOCK** anterior antes de la ejecución de cualquier sentencia DML (**SELECT/INSERT/DELETE/UPDATE/FETCH/COPY_TO**) en la transacción.

Esto inconvenientes desaparecerán en el futuro cuando la habilidad para leer datos sucios (no confirmados) (a pesar del nivel de aislamiento) y la verdadera integridad referencial sea implementada.

Lista Detalla de Cambios

Correcciones de errores

Correccion de las funciones de conversion text<->float8 y text<->float4(Thomas)
 Correccion para creacion de tablas con constraints con mixed-case(Billy)
 Cambiado comportamiento de exp()/pow() para generar error en underflow/overflow(Jan)
 Correccion de error en pg_dump -z
 Limpiezas de invasiones de memoria(Tatsuo)
 Correccion para abortos de lo_import(Tatsuo)
 Ajustes en el manejo de nombres de tipo de datos para suprimir dobles comillas(Thomas)
 Uso de coercion de tipo para emparejar columnas y DEFAULT(Thomas)
 Correccion de deadlock de este modo solo verifica una vez despues de un segundo de espera(Bruce)
 Correcciones para agregaciones y PL/pgsql(Hiroshi)
 Correccion para aborto de subquery(Vadim)
 Correccion de libpq para la funcion PQfnnumber y nombres que no distinguen mayusculas-minusculas(Bahman Rafatjoo)
 Correccion para objetos grandes escritos-en-medio, no bloques extra, consumo de memoria(Tatsuo)
 Correccion para pg_dump -d o -D y entrecomillado de caracteres especiales en INSERT
 Reparados serios problemas con dynahash(Tom)
 Correjidos problemas de portabilidad para INET/CIDR
 Correccion de problema con error de selectividad en ALTER TABLE ADD COLUMN(Bruce)
 Correccion del ejecutor de ese modo trabaja merzejoin de diferentes tipos de columnas(Tom)
 Correccion de error para selectividad de OR en Alpha
 Correccion para problema de selectividad de indice OR(Bruce)
 Correccion \d para que muestre de ese modo la extension apropiada para char()/varchar()(I
 Correccion en el codigo del tutorial(Clark)
 Mejoras en la comprobacion de destroyuser(Oliver)
 Correccion para Kerberos(Rodney McDuff)
 Correccion para borrado de la base de datos mientras los buffers no se han limpiado(Bruce)
 Correccion la secuencia nextval() para que pueda asi distinguir mayusculas-minusculas
 Correccion para operador !=
 Borrado de buffers antes de destruir los ficheros de las base de datos(Bruce)
 Correccion del caso en que el ejecutor evalua las funciones dos veces(Tatsuo)
 Se permite a las acciones de secuencia nextval distinguir mayusculas-minusculas(Bruce)
 Correccion de optimizador de indexacion para que no trabaje para numeros negativos(Bruce)
 Correccion de perdidas de memoria en ejecuciones con fjisNull
 Correccion de perdidas de memoria para aggregate(Erik Riedel)
 Se permite que username contenga una almohadilla (#, dash en ingles) en los permisos GRANT
 Limpieza de NULL en los tipos inet
 Limpieza de errores en tablas del sistema(Tom)
 Correccion de problemas de PAGER y del comando \?(Masaaki Sakaida)
 Reducido el tamaño de fichero de multi-segment por defecto a 1GB(Peter)
 Correccion del volcado de CREATE OPERATOR(Tom)
 Correccion para escaneo hacia atras de cursores(Hiroshi Inoue)
 Correccion para COPY FROM STDIN cuando se utiliza \i(Tom)
 Correccion para una subselect cuando es comparada dentro de una expresion(Jan)
 Correccion para manejo de devolucion de error mientras se duelen registros(Tom)
 Correccion de problemas con referencia a tipos de array(Tom,Jan)
 Se previene oid de UPDATE SET (Jan)
 Correccion de pg_dump asi ña opcion -t puede manejar nombres de tabla en mayusculas-minusculas
 Correcciones para GROUP BY es casos especiales(Tom, Jan)
 Correccion de perdidas de memoria en queries falladas(Tom)
 DEFAULT soporta ahora identificadores mixed-case(Tom)
 Correccion para que multi-segment utilice DROP/RENAME de tabla, de indice(Ole Gjerde)
 Dehabilitacion de uso de pg_dump con las dos opciones -o y -d(Bruce)

Se permite a pg_dump volver adecuadamente permisos de GROUP(Bruce)
 Correccion para GROYP BY en INSERT INTO table SELECT * FROM table2(Jan)
 Correccion para computaciones en vistas(Jan)
 Correcciones para agregaciones en array con indices(Tom)
 Correccion para como maneja DEFAULT entrecomillado simple en valores que requieren demasiadas comillas
 Correccion de problema de seguridad con no super-usuarios que importan/exportan objetos de gran tamaño.(Tom)
 Vuelta atras de transaccion que crea table la limpia adecuadamente(Tom)
 Correccion para permitir que tablas largas y nombres de columnas generen nombres en serie adecuados(Tom)

Mejoras

Añadida la utilidad "vacuumdb"
 Se acelera libpq por mejor asignacion de memoria(Tom)
 EXPLAIN utiliza todos los indices(Tom)
 Implementadas las expresiones CASE, COALESCE, NULLIF(Thomas)
 Nuevo formato de salida de pg_dump(Constantin)
 Añadida la cadena min()/max() a las funciones(Thomas)
 Extendidas nuevo tipo de coersiones para agregaciones(Thomas)
 Nueva contribucion moddatetime (Terry)
 Actualizacion a pgaccess 0.96(Constantin)
 Añadida rutina para byte unico en tipo de caracter "char"(Thomas)
 Mejorada la funcion substr()(Thomas)
 Mejorado el manejo de multi-byte (Tatsuo)
 Control de concurrencia Multi-version /MVCC(Vadim)
 Nuevo modo Serialized(Vadim)
 Correccion para tablas por encima de 2gigs(Peter)
 Nuevo SET TRANSACTION ISOLATION LEVEL(Vadim)
 Nuevo LOCK TABLE IN ... MODE(Vadim)
 Actualizado el driver ODBC(Byron)
 Nuevo tipo de datos NUMERIC(Jan)
 Nueva SELECT FOR UPDATE(Vadim)
 Manejo de "NaN" e "Infinity" para valores de entrada(Jan)
 Mejorado el manejo de date/year(Thomas)
 Mejorado el manejo de conexiones con el motor de base de datos(backend)(Magnus)
 Nuevas opciones ELOG_TIMESTAMPS y la opcion USE_SYSLOG para ficheros de registro(Massimo)
 Nueva opcion TCL_ARRAYS (Massimo)
 Nueva INTERSECT y EXCEPT(Stefan)
 Nuevo pg_index.indisprimary para restro de claves primarias(D'Arcy)
 Nuevas opcion pg_dump para permitir el borrado de tablas antes de su creacion(Brook)
 Acelaracion de las rutinas de salida de registro(Tom)
 Nuevo nivel de aislamiento de READ COMMITTED (Vadim)
 Nuevas tablas/indices TEMP (Bruce)
 Se evita el ordenamiento si el resultado ya esta ordenado(Jan)
 Nueva optimizacin para la asignacion de memoria(Jan)
 Se permite a psql ejecutar \p\g(Bruce)
 Se permiten multiples reglas de acciones(Jan)
 Añadida funcionalidad LIMIT/OFFSET (Jan)
 Mejorado el optimizador cuando se unen un numero grande de tablas(Bruce)
 Nueva introduccion a SQL de La Tesis de Doctorado de S. Simkovics(Stefan, Thomas)
 Nueva introduccion a procesamiento del motor de base de datos (backend) de la Tesis de Doctorado de S. Simkovics(Stefan)
 Mejorado el soporte para int8(Ryan Bradetich, Thomas, Tom)
 Nuevas rutinas para convertir entre tipos int8 y text/varchar(Thomas)
 Nuevos planes arboreos, donde se unen meta-tablas(Bruce)
 Habilitadas consultas por la mano derecha por defecto(Bruce)
 Se permite que el numero maximo de procesos en servidor (backends) se parametrice en el momento de la configuracion
 (-with-maxbackends and postmaster switch (-N backends))(Tom)
 GEQO por defecto tenga ahora 10 tablas porque el optimizador se acelera(Tom)
 Se permite NULL=Var para MS-SQL portabilidad(Michael, Bruce)

Modificados contrib check_primary_key() y consecuentemente "automatic" or "dependent"(Ar)
 Se permite que psql \d en una vista muestre la consulta(Ryan)
 Se acelera el LIKE(Bruce)
 Correcciones/prestaciones EcpG , vease fichero src/interfaces/ecpg/ChangeLog(Michael)
 Correcciones/prestaciones JDBC , vease src/interfaces/jdbc/CHANGELOG(Peter)
 Se hace que el operador % tenga precedencia como /(Bruce)
 Añadida la nueva opcion postgres -O para permitir cambios en la estructura de tablas del sistema(Bruce)
 Actualizado el script contrib/pginterface/findoidjoins (Tom)
 Mayor aceleracion en vacuum de lineas borradas con indices(Vadim)
 Se permite la ejecucion de diferentes versiones de funciones no-SQL basadas en argumentos(Tom)
 Añadida la opcion -E que muestra las consultas actuales enviadas pro \dt y sus amigos(Masaaki Sakaida)
 Añadido numero de version en los banners de arranque de psql(Masaaki Sakaida)
 Nuevo contrib/vacuumlo que elimina objetos grandes no referenciados(Peter)
 Nueva inicializacion para tamaños de tablas, asi las tablas no vacuumeadas se ejecutan mejor(Tom)
 Mejorados los mensajes de error cuando una conexion es rechazada(Tom)
 Soporte para array de campos char() y varchar() (Massimo)
 Revision del codigo has para incrementar fiabilidad y prestaciones(Tom)
 Actualizacion a PyGreSQL 2.4(D'Arcy)
 Cambiadas las opciones de depuracion asi -d4 y -d5 producen diferentes displays de nodos(Jan)
 nuevas opciones: pretty_plan, pretty_parse, pretty_rewritten(Jan)
 Mejor optimizacion de estadisticas para los accesos a tablas del sistema(Tom)
 Mejor manejo de tamaño de bloque no por defecto(Massimo)
 Mejorado el optimizador de consumo de memoria GEQO(Tom)
 UNION soporta ahora ORDER BY de columnas que no estan en la lista de target(Jan)
 Mejoras grandes en libpq++ (Vince Vielhaber)
 pg_dump utiliza ahora -z(ACL's) por omision(Bruce)
 cache de procesos en servidor (backend), aceleracion de memoria(Tom)
 Se hace que pg_dump lo haga todo en una transaccion snapshot(Vadim)
 correccion de perdidas de memoria para objetos grandes, correccion para pg_dumping(Tom)
 INET escribe ahora respecto a la netmask para comparaciones
 Se hace que VACUUM ANALYZE solo utilice un readlock(Vadim)
 Se permiten vistas (VIEW) en UNIONS(Jan)
 pg_dump puede generar ahora snapshots consistentes en bases de datos activas(Vadim)

Cambios en el Arbol Fuente

Mejorado el emparejamiento en el porte(Tom)
 Correcciones de portabilidad para SunOS
 Añadido porte para NT/Win32 del proceso en el servidor (backend) y se habilita carga dinamica(Magnus and Daniel Horak)
 Nuevo porte a Cobalt Qube(Mips) ejecutando Linux(Tatsuo)
 Porte a NetBSD/m68k(Mr. Mutsuki Nakajima)
 Porte a NetBSD/sun3(Mr. Mutsuki Nakajima)
 Porte a NetBSD/macppc(Toshimi Aoki)
 Correccion para configuracion de tcl/tk(Vince)
 Eliminada la clave CURRENT para consultas por regla(Jan)
 carga dinamica en NT ahora funciona(Daniel Horak)
 Añadido soporte para ARM32(Andrew McMurry)
 Mejor soporte para HP-UX 11 y Unixware
 Mejorado el manejo de ficheros para ser mas uniforme, previene lagunas en los descriptors de ficheros.(Tom)
 Nuevos comandos de instalacion para plpgsql(Jan)

Version 6.4.2

La version 6.4.1 fue incorrectamente empaquetada. Esta tiene tambien una correccion adicional para un bug

Migracion a v6.4.2

No se requiere un dump/restores para aquellos que esten ejecutando una 6.4.*.

Lista Detallada de Cambios

Correccion para problema de constante de fecha y hora en algunas plataformas (Thomas)

Version 6.4.1

Esta es basicamente una limpieza de la version 6.4. Hemos corregio una variedad de problemas reportados por usuarios de 6.4

Migracion a v6.4.1

No se requiere un dump/restores para aquellos que esten ejecutando una 6.4.

Lista Detallada de Cambios

Añadida la opcion -N a pg_dump para forzar dobles comillas alrededor de los identificadores. Este es la opcion por omision (Thomas)
Correccion para NOT in donde la clausula causa abortos (Bruce)
Correccion para el coredump de EXPLAIN VERBOSE (Vadim)
Correccion para problemas de shared-library en Linux
Correccion del test de existencia de tabla para permitir combinacion de mayusculas y minusculas y
espacios en blanco en el nombre de tabla (Thomas)
Fijacion de un par de problemas de pg_dump
La configuracion empareja mejor entradas similares en la platilla template/. (Tom)
Cambios en la construccion de los nombre de la funcion de SPI_* a spi_*
Correccion para la clausula OR WHERE (Vadim)
Correcciones para nombres de tablas con mayusculas y minusculas (Billy)
Correccion para contrib/linux/postgres.init.csh/sh (Thomas)
rebasamiento de memoria en libpq corregido
Correcciones para SunOS (Tom)
Se cambia el comportamiento de exp() para generar error en negativos (Thomas)
pg_dump fixes for memory leak, inheritance constraints, layout change
Actualizado pgaccess a 0.93
Se corrige el prototipo para plataformas de 64-bit
Correcciones para Multi-byte (Tatsuo)
Nueva pagina de manual ecpg
Corregidos rebasamientos de memoria (Tatsuo)
Correccion para fallos en lo_import() (Bruce)
Mejores busquedas para el programa de instalacion (Tom)

Correcciones para zona horaria(Tom)
 Correcciones para HPUX(Tom)
 Se utiliza coersiones de tipo implicito para emparejar valores DEFAULT(Thomas)
 Anadidas rutinas para ayudar a tipo de caracter bytes-solo(single-byte) (internal)(Thomas)
 Correcciones para compilacion de libpq en Win32(Magnus)
 Actualizacion a PyGreSQL 2.2(D'Arcy)

Version 6.4

Hay *muchas* prestaciones nuevas y mejoras en esta version. Gracias a unestros desarrolladores y mantenedores, casi todos los aspectos del sistema han recibido alguna atencion desde la version anterior. He aqui un resumen, sumario incompleto:

- Las vistas y las reglas son ahora funcionales gracias al extensivo nuevo codigo en las reglas de re escritura de Jan Wieck. Tambien escribio un capitulo sobre ello en la *Guia del Programador*.
- Jan tambien contribuyo al segundo lenguaje procedural, PL/pgSQL, para ir con el original lenguaje procedural PL/pgTCL con el que el contribuyo la ultima version.
- Tenemos soporte opcional de caracter multiple-byte de Tatsuo Iisho para complementar nuestro soporte local.
- Las comunicaciones cliente/servidor han sido depuradas, con mejor soporte para mensajes asincronos e interrupciones, gracias a Tom Lane.
- El depurador de sintactico ejecuta ahora coersiones de tipo automatico para emparejar argumentos a los operadores y funciones disponibles, y para emparejar columnas y expresiones con columnas destino. Esto utiliza un mecanismo generico que soporta las prestaciones de extensibilidad de tipo de Postgres. Hay un nuevo capitulo en la *Guia de Usuario* que cubre este asunto.
- Tres nuevos tipos de datos han sido anadidos. Dos tipos, inet y cidr, soportan varias formas de trabajo en red IP, subred, y direccionamiento por maquina. Ahora hay un tipo entero de 8 byte disponible para algunas plataformas. Vease el capitulo de tipos de datos en la *Guia del Usuario* para mas detalles. Un cuarto tipo, serial, se soporta ahora por el depurador de sintactico como una amalgama de tipo int4, una secuencia, y un indice unico.
- Han sido añadidas varias prestaciones mas sintacticas compatibles con SQL92, incluyendo **INSERT DEFAULT VALUES** Several more SQL92-compatible syntax features have been added, including **INSERT DEFAULT VALUES**
- La instalacion y configuracion automatica del sistema ha recibido alguna atencion, y deberia ser mas robusta para mas plataformas de lo que nunca ha sido.

Migracion a v6.4

Se requiere un dump/restore utilizando `pg_dump` o `pg_dumpall` para aquellas que desen migrar datos desde cualquier version anterior de Postgres.

Lista Detallada de Cambios

Correcciones de errores

Correccion para una minuscula perdida en PQsetdb/PQfinish(Bryan)
 Se elimina char2-16 de los tipos de datos, se utiliza char/varchar(Darren)
 Pqfn no maneja un mensaje de NOTICE(Anders)
 Reducidas elevadas esperas por ocupacion a causa de bloqueos en transacciones con muchos procesos en servidor(backends) (dg)
 Deteccion de bloqueos de transacciones atascadas (dg)
 Correccion para masrcas de tiempo en estilo "ISO" en decodificacion y codificacion(Thomas)
 Correccion del problema con borrado de tabla (drop) despues de deshacer (rollback) una transaccion(Vadin)
 Cambiado mensaje de error y eliminado mensaje actualizado no funcional(Vadim)
 Correccion para verificacion de la matriz (array) de COPY
 Correccion para SELECT 1 UNION SELECT NULL
 Correccion para perdidas de buffer en llamadas a objetos grandes(Pascal)
 Cambio de propietario de tipo oid a int4(Bruce)
 Correccion de error en la compatibilidad con oracle de las funciones btrim() ltrim() y rtrim()
 Correccion de invalidacion en rebasamientos de cache compartida(Massimo)
 Prevencion de perdidas en descriptores de ficheros en COPY's fallidos(Bruce)
 Correccion para perdidas en la pg_select de libpgtcl(Constantin)
 Correccion de problemas con usuario/contrasena de mas de 8 caracteres(Tom)
 Correccion de problemas con manejo de NOTIFY asincronos en el proceso en servidor(backend)
 Correccion de muchas entradas de sistema malas(Tom)

Mejoras

Actualizacion de ecpg y ecpglib, vease src/interfaces/ecpg/ChangeLog(Michael)
 Se muestra en indice utilizado en un EXPLAIN(Zeugswette)
 EXPLAIN invoca una regla de sistema y muestra plan(es) para la reescritura de consultas(Jan)
 Conocimiento multi-byte de muchos tipos de datos y funciones, via configure(Tatsuo)
 Nuevo configure con la opcion -with-mb(Tatsuo)
 Nueva opcion initdb -pgencoding(Tatsuo)
 Nueva opcion createdb -E multibyte(Tatsuo)
 Select version(); ahora devuelve la version de PostgreSQL(Jeroen)
 Libpq permite ahora clientes asincronos(Tom)
 Se permite la cancelacion desde el cliente de una consulta en el proceso en servidor(backend)(Tom)
 Psql cancelas las consultas ahora con Control-C(Tom)
 Usuarios de Libpq no necesitan dar consultas dummy para obtener mensajes NOTIFY(Tom)
 NOTIFY envia ahora al PID del emisor, asi que puedes decir si eras tu mismo(Tom)
 La estructura de PGresult ahora incluye un mensaje de error asociado, si lo hay(Tom)
 Se definen los argumentos "tz_hour" y "tz_minute" como date_part()(Thomas)
 Se anaden rutinas para convertir entre varchar y bpchar(Thomas)
 Se anaden rutinas para permitir el dimensionamiento de varchar y bpchar dentro de las columnas de destino(Thomas)
 Se anade un bit a las etiquetas (flags) oara soportar zona horaria y minutos en la devolucion de fecha(Thomas)
 Se permiten mas variaciones en numeros de coma flotante (por ej. ".1", "1e6")(Thomas)
 Se corrigen el analisis sintactico de menores unarios empezando con espacios(Thomas)
 Se implementa TIMEZONE_HOUR, TIMEZONE_MINUTE por especificaciones SQL92(Thomas)
 Se verifica i se ignora adecuadamente constraints de columna FOREIGN KEY(Thomas)
 SE defina USER como sinonimo de CURRENT_USER por especificacines de SQL92(Thomas)
 Se habilita HAVING en clausulas pero no se corrije en ningun otro lugar aun.
 Se hace el tipo "char" un sinonimo de "char(1)" (actualmente implementado como bpchar)(Thomas)
 Se guarda el tipo de cadena si esta especificado por el manejo de la clausula DEFAULT(Thomas)

Operaciones de coercion abarcan diferentes tipos de datos(Thomas)
 Se permite a algunos indices utilizar columnas de diferentes tipos(Thomas)
 Anadidas capacidades para coersiones de tipo automatico(Thomas)
 Depuraciones para objetos grandes, de este modo un fichero es truncado en su apertura(Peter)
 Depuraciones de la lectura de lineas(Tom)
 Se permite que psql \f \ tomen los espacios en blanco como delimitadores(Bruce)
 Se pasa el pg_attribute.atttypmod al frontal de la aplicacion para las longitudes de los campos(Tom,Bruce)
 Libreria de compatibilidad con Msql en /contrib(Aldrin)
 Se elimina el requerimiento de que las clausulas identificadoras ORDER/GROUP BY estuvieran incluidas en la lista de la busqueda(David)
 Se convierten columnas para emparejarlas en las clausulas de UNION(Thomas)
 Se elimina fork()/ecec() y solo se hace fork()(Bruce)
 Depuraciones en Jdbc(Peter)
 Se muestra el estado del proceso en el servidor (backend) en la linea de comandos de ps(solo funciona en algunas plataformas)(Bruce)
 Pg_hba.conf tiene ahora una opcion sameuser en el campo de la base de datos
 Se hace que lo_unlink tome el parametro oid, no el int4
 Nueva DISABLE_COMPLEX_MACRO para compiladores que no pueden manejar nuestras macros(Bruce)
 Libpgtcl maneja los NOTIFY ahora como un evento Tcl, no se necesitan enviar consultas tontas(Tom)
 Depuraciones en libpgtcl(Tom)
 Anadida una opcion -error al comando pg_result de libpgtcl(Tom)
 Anadido parche locale, vease docs/README/locale(Oleg)
 Correccion para pg_dump con ella la sintaxis de CONSTRAINT y CHECK es correcta(ccb)
 Nuevo codigo contrib/lo para eliminar grandes objetos huérfanos(Peter)
 Nuevp comando psql "SET CLIENT_ENCODING TO 'encoding'" para prestaciones multi.byte, vease /doc/README.mb(Tatsuo)
 codigo /contrib/noupdate para revocar permisos de actualizacion en una columna
 Libpq puede ser compilada ahora en win32(Magnus)
 Anadido PQsetdbLogin() en libpq
 Nuevo tipo entero 8-byte, comprobado por el configure del soporte para OS(Thomas)
 Mejor soporte para nombres entrecomillados de tabla/columnas(Thomas)
 Se rodean los nombres de tabla y columnas con dobles comillas en pg_dump(Thomas)
 PQreset() trabaja ahora con contraseñas(Tom)
 Handle case of GROUP BY target list column number out of range(David)
 Se permite UNION en las subconsultas
 Anadido auto-dimensionamiento a la pantalla a los comandos \d?(Bruce)
 Se utiliza UNION para mostrar todos los resultados de \d? en una consulta(Bruce)
 Se anade la prestacion de busqueda de campo \d?(Bruce)
 Pg_dump utiliza menos peticiones \connect(Tom)
 Se hace que la opcion -z de pg_dump trabaje mejor, se documenta en la pagina de manual(Tom)
 Se anade la clausula HAVING con total soporte para subconsultas y uniones(Stephan)
 Texto completo de las rutinas de indexado en contrib/fulltextindex(Maarten)
 Los ids de transacciones se almacenan ahora en memoria compartida(Vadim)
 Nuevo PGCLIENTENCODING cuando ejecutan el comando COPY(Tatsuo)
 Soporte para la sintaxis SQL92 "SET NAMES"(Tatsuo)
 Soporte para LATIN2-5(Tatsuo)
 Anadido el caso UNICODE los test de refresion(Tatsuo)
 Depuracion de gestor de bloqueos, nuevos modos de bloqueos para LLL(Vadim)
 Se permite el uso de indice en clausulas OR(Bruce)
 Se permite "SELECT NULL ORDER BY 1;"
 La explicacion VERBOSE del plan lo imprime, y ahora imprime en bonito el plan al fichero de log del postmaster(Bruce)
 Se anaden indices al display para el comando \d(Bruce)
 Se permite el GROUP BY en funciones(David)
 Nuevo pg_class.relkind para objetos grandes(Bruce)
 Nuevo modo de enviar libpq mensajes NOTICE a diferentes localizaciones(Tom)
 Nuevo comando de escritura \w para psql(Bruce)

Nuevo /contrib/findoidjoins escanea columnas oid para encontrar relaciones de union(Bruce)
 Se permite que sean considerados indices compatibles binarios cuando se verifican indices validos para una clausula de restriccion conteniendo una constante(Thomas)
 Nuevo codigo ISBN/ISSN en /contrib/isbn_issn
 Se permite NOT LIKE, IN, NOT IN, BETWEEN, y NOT BETWEEN constraint(Thomas)
 Nuevo sistema de reescritura corrige muchos problemas con reglas y vistas(Jan)

- * Reglas en trabajos relacionados
- * Cualificaciones de eventos en trabajos de inserciones/actualizaciones/borrados
- * Nueva variable OLD para referirse a CURRENT, CURRENT se elimina en un futuro
- * Las reglas de actualizacion se pueden referir a NEW y OLD en la regla cualificacion/accion
- * Inserciones/actualizaciones/borrados en vistas de trabajo
- * Reglas multiples de accion se soportan ahora, rodeadas entre parentesis
- * Usuarios normales pueden crear vistas/reglas en las tablas en las que tengan permisos de RULE
- * Las reglas y las vistas heredan los permisos del creador
- * No hay reglas a nivel de columna
- * No hay reglas de ACTUALIZACION NUEVA/VIEJA (UPDATE NEW/OLD)
- * Nuevas vistas de sistema pg_tables, pg_indexes, pg_rules y pg_views
- * Solo se puede ejecutar una accion en las reglas de SELECT
- * Re escritura total revisada, tal vez para la 6.5
- * manejo de subselects
- * manejo de agregaciones en vistas
- * manejo de inserciones dentro de una seleccion desde una vista ahora funciona

 Los indices del sistema ahora son multi-clave(Bruce)
 Los tipos Oidint2, oidint4, y oidname types se eliminan(Bruce)
 Se utiliza cache del sistema para mas busquedas en tablas del sistema(Bruce)
 Nueva lenguaje de programacion en el proceso en servidor(backend) PL/pgSQL en backend/pg
 El nuevo tipo de datos SERIAL, auto crea la secuencia/indice(Thomas)
 Se permite la comprobacion de declaraciones sin recompilar(Massimo)
 Mejoras en el bloqueo de usuario(Massimo)
 Nuevo comando setval() para configurar el valor de una secuencia(Massimo)
 Auto eliminacion del fichero de socket de unixsi no hay un postmaster ejecutandose(Massimo)
 Paquete de traceo condicional(Massimo)
 Nuevo comando UNLISTEN(Massimo)
 Psql y libpq se compilan ahora bajo win32 utilizando win32.mak(Magnus)
 Lo_read ya no almacena rastros NULL (Bruce)
 Los identificadores son truncados ahora internamente a 31 caracteres(Bruce)
 Opciones de createuser estan disponibles ahora en la linea de comando
 Anadido soporte para codigo de enteros de 64-bit, configuracion comprobada, tipos de int8(Thomas)
 Se previene la perdida de un descriptor a causa de un COPY fallido(Bruce)
 Nuevo comando pg_upgrade(Bruce)
 Directorios Updated /contrib (Massimo)
 Nueva sentencia CREATE TABLE DEFAULT VALUES disponible(Thomas)
 Nueva sentencia INSERT INTO TABLE DEFAULT VALUES disponible(Thomas)
 Nueva prestacin DECLARE y FETCH(Thomas)
 Estructuras internas de libpq ahora no se exportan (Tom)
 Se permiten indices con mas de 8 claves(Bruce)
 Se elimina el teclado ARCHIVE, que ya no se utiliza(Thomas)
 La opcion -n de pg_dump para suprimir comillas alrededor de los identificadores se deshabilita las columnas del sistema para las vistas(Jan)
 nuevos tipos INET y CIDR para direcciones de red(TomH, Paul)
 No mas comillas en las salidas de psql
 pg_dump ahora vuelca las vistas(Terry)
 nuevo SET QUERY_LIMIT(Tatsuo,Jan)

Cambios en el Arbol Fuente

 Limpieza de /contrib (Jun)

Enlazadas algunas pequenas funciones llamadas para cada registro(Bruce)
 Inline some small functions called for every row(Bruce)
 Correcciones paraAlpha/linux
 Limpiezas para Hp/UX (Tom)
 Test de regresion para Multi-byte (Soonmyung.)
 Se elimina la opcion -disabled del configure
 Se define PGDOC para que utilice POSTGRES DIR por defecto
 Se hace la regresion opcional
 Se eliminan corchetes extras en el codigo de pgindent(Bruce)
 Se anade soporte para la libreria compartida bsdi(Bruce)
 Nueva soporte para opcion de configuracion -without-CXX support(Brook)
 Nueva FAQ_CVS
 Actualizado flowchart de proceso de servidor en tools/backend(backend)
 Cambiado atttymod de int16 a int32(Bruce, Tom)
 Se corrige Getrusage() para plataformas que no lo tienen(Tom)
 Se anade PQconnectdb, PGUSER, PGPASSWORD a la pagina de manual de libpq
 NS32K platform fixes(Phil Nelson, John Buller)
 Correcciones para Sco 7/UnixWare 2.x (Billy,others)
 Correcciones para Sparc/Solaris 2.5 (Ryan)
 Pgbuiltin.3 esta obsoleto, movido a los ficheros de documentacion(Thomas)
 Aun mas documentacion(Thomas)
 Soporte para Nexstep(Jacek)
 Soporte para Aix (David)
 Pagina de manual para pginterface(Bruce)
 Todas las librerias compartidas tienen numero de version
 Unidas todos los defines de las librerias compartidas de SO-especificos dentro de un unico fichero
 Comprobacion de configuracion TCL/TK mas inteligente(Billy)
 Configuracion de perl mas inteligente(Brook)
 configure utiliza install-sh facilitado si no se encuentra script de instalacion(Tom)
 nueva Makefile.shlib para configuracion de librerias compartidas(Tom)

Version 6.3.2

Esta es una version de correccion de errores para 6.3.x. Consultese las notas de la version v6.3 para un sumario completo de las nuevas prestaciones.

Sumario:

- Se repara soporte para configuracion automatica para algunas plataformas, incluyendo Linux, a causa de fallos introducidos sin advertirlo en v6.3.1.
- Se manejan correctamente las llamadas a funciones en la parte izquierda de las clausulas BETWEEN y LIKE.

No se requiere un dump/restore para aquellos que esten ejecutando 6.3 o 6.3.1. Un 'make distclean', 'make', y 'make install' es todo lo requerido. Este ultimo paso deberia ser ejecutado mientras que la postmaster no este ejecutandose. Deberia re-enlazar (re-link) cualquier aplicacion cliente que utilice librerias Postgres.

Para actualizaciones desde instalaciones pre-v6.3, consultese las instrucciones de instalacion y migracion para v6.3

Lista Detallada de Cambios

```

Cambios
-----
Mejoras en la deteccion del configure para tcl/tk(Brook Milligan, Alvin)
Mejoras en las paginas de manual(Bruce)
correcciones para BETWEEN y LIKE (Thomas)
correccion para psql \connect utilizado por pg_dump(Oliver Elphick)
Nuevo driver odbc
pgaccess, version 0.86
eliminado qsort, se utiliza ahora la version de libc, depuraciones(Jeroen)
correcciones para buffer excedidos que se han detectado(Maurice Gittens)
correcciones para buffer excedidos en libpgtcl(Randy Kunkee)
correccion para UNION con DISTICNT en ORDER BY(Bruce)
se comprueba en el configure el gettimeofday (Doug Winterburn)
Correccion para el error "indices no usados"(Vadim)
Adicciones a la documentcion(Thomas)
Correcciones para perdida de memoria en el proceso en el servidor(backend)(Bruce)
limpiezas en libreadline (Erwan MAS)
Eliminado DISTDIR(Bruce)
limpieza de las dependencias de Makefile (Jeroen van Vianen)
correcciones para ASSERT (Bruce)

```

Version 6.3.1

Sumario:

- Soporte adicional para conjuntos de caracteres multi-byte.
- Reparacion de la ordenacion de bytes para clientes y servidores de frontal mixto.
- Actualizaciones menores para permitir sintaxis SQL.
- Mejoras a la autodeteccion de la configuracion durante la instalacion.

No se requiere un dump/restore para aquellos que ejecuten 6.3. Un 'make distclean', 'make', y 'make install' es todo lo que se requiere. Deberia re-enlazar (re-link) cualquier aplicacion cliente que utilice librerias Postgres.

Para actualizaciones desde instalaciones pre-v6.3, consultese las instrucciones de instalacion y migracion para v6.3

Lista Detallada de Cambios

```

Cambios
-----
depuracion/correcciones para ecpg, ahora en version 1.1(Michael Meskes)
depuracion en pg_user(Bruce)
correccion para objetos largos para pg_dump y tclsh(alvin)
correccion de LIKE para multiples subrayados adyacentes
correccion para redefiniciones en la construccion de funciones(Thomas)
depuraciones para ultrix4
actualizacion a pg_access 0.83
actualizacion de la pagina de manual CLUSTER

```

Soporte para juego de caracteres multi-byte, vease doc/README.mb(Tatsuo)
 correccion para configure -with-pgport
 correccion para pg_ident
 correccion de big-endian para comunicaciones en el proceso servidor(backend)(Kataoka)
 correccion para SUBSTR() y substring() (Jan)
 varias correcciones para jdbc (Peter)
 libpgtcl improvements, see libpgtcl/README(Randy Kunkee)
 Correccion del error "Datasize = 0" (Vadim)
 Se previene la ocultacion de \do(Bruce)
 Eliminadas entradas de juegos de caracteres Ruso duplicadas
 depuracion para Sunos4
 Se permiten claves opcionales de TABLA en LOCK y SELECT INTO(Thomas)
 Opciones de CREATE SEQUENCE para permitir enteros negativos(Thomas)
 Se anade "PASSWORD" como un identificador de columna valido(Thomas)
 Se anade comprobacion de los campos objeto en UNION (Bruce)
 Correccion para el porte a Alpha(Dwayne Bailey)
 Correccion para matrices con texto contenido comillas(Doug Gibson)
 Correccion para la compilacion en Solaris(Albert Chin-A-Young)
 Se indentifica mejor librerias e includes de tcl y tk(Bruce)

Version 6.3

Hay *muchas* prestaciones nuevas y mejoras en esta version. He aqui un breve, incompleto resumen:

- Muchas prestaciones nuevas, incluyendo capacitacion total para subconsultas SQL92 (todo esta aqui excepto la lista destino en la subconsulta)
- Soporte para variables de entorno del lado del cliente para especificar zona horaria y estilo de fecha.
- Interfaz de conexion (socket) para conexiones cliente/servidor. Esto es por ahora defecto asi que necesitas iniciar postmaster con la opcion "-i"
- Mejores mecanismos de autorizacion con contrasena. Los permisos de tabla por defecto han cambiado.
- El viejo estilo de "time travel" ha sido eliminado. Se han mejorado los resultados.

Nota: Bruce Momjian escribio las siguientes notas para presentar la nueva version.

Hay algunas cosas generales en la 6.3 que quiero mencionar. Son solo los grandes items que no pueden ser descritos en una frase. Se aun necesita una revision de la lista de cambios detallado.

Primero, ahora tenemos subconsultas. Ahora que las tenemos, quiero decir que sin subconsultas, SQL es un lenguaje muy limitado. Las subconsultas son una prestacion mayor, y deberia revisar su codigo en los lugares en los que una subconsulta le provea de una mejor solucion para sus consultas. Creo que encontrara que hay mas usos para las subconsultas de los que usted cree. Vadim nos ha puesto en el gran mapa de SQL con las subconsultas, y con ellas funcionales totalmente. La unica cosa que no puede hacer con las subconsultas es utilizarlas en lista a obtener(target list).

Segundo, la 6.3 utiliza conexión de dominio unix(unix domain sockets) en vez TCP/IP por defecto. Para permitir conexiones desde otras máquinas, tiene que utilizar la nueva opción `postmaster -i`, y por supuesto editar `pg_hba.conf`. También por esta razón el formato de `pg_hba.conf` ha cambiado.

Tercero, los campos `char()` permitirán ahora un acceso más rápido que `varchar()` o que campos de texto. Específicamente, el texto y el `varchar()` tienen una penalización para el acceso a cualquier columna después de la primera columna de este tipo. `char()` utiliza también este acceso penalizado, pero ya no existe por más tiempo. Esto puede sugerir que rediseñe algunas de sus tablas, específicamente si tiene columnas de caracteres cortos que haya definido como `varchar()` o texto. Este y otros cambios hacen la 6.3 aún más rápida que versiones precedentes.

Tenemos ahora contraseñas definibles independientemente de cualquier fichero Unix. Hay nuevos comandos SQL `USER`. Véase la página de manual `pg_hba.conf` para más información. Hay una tabla nueva, `pg_shadow`, que es utilizada para almacenar información de los usuarios y las contraseñas de los usuarios, y que es, por defecto solo consultable (SELECT-able) por el super usuario de postgres. `pg_user` es ahora una vista de `pg_shadow`, y es consultable (SELECT-able) por PUBLIC. Debe seguir utilizando `pg_user` en sus aplicaciones sin cambios.

Las tablas creadas por usuarios ahora ya no tienen permisos de consulta (SELECT) para PUBLIC por defecto. Esto se hizo porque los estándares ANSI lo requieren. Puede por supuesto conceder (GRANT) cualquier permiso que quiera después de que la tabla sea creada. Las tablas del sistema continúan siendo consultables (SELECT-able) por PUBLIC.

Tenemos también verdaderos códigos de detección de bloqueos (deadlock). No más sesenta segundos de tiempo de expiración (timeouts). Y el nuevo código implementa un FIFO mejor, así que debería haber un menor agotamiento de recursos durante un uso fuerte.

Muchas quejas han sido hechas acerca de la inadecuada documentación en versiones anteriores. Thomas ha puesto mucho esfuerzo en muchos manuales nuevos para esta versión. Compruebe en directorio /doc.

Por razones de rendimiento, `time travel` se elimina, pero puede ser implementado utilizando disparadores (triggers) (véase `pgsql/contrib/spi/README`). Por favor, compruebe el nuevo comando `\d` para tipos, operadores, etc. También las vistas tienen ahora sus propios permisos, no basados en las tablas subyacentes, así que los permisos para ellas deben ser configurados separadamente. Compruebe `pgsql/interfaces` para ver algunas maneras de dialogar con Postgres.

Esta es la primera versión que realmente requiere una explicación para los usuarios existentes. De muchas maneras, fue necesario esto porque la nueva versión elimina muchas limitaciones, y las soluciones provisionales (work-around) que eran la gente estaba utilizando ya no se necesitan.

Migración a v6.3

Se requiere la utilización de un `pg_dump` o de un `pg_dumpall` para todos aquellos que deseen migrar datos de cualquier versión anterior de Postgres.

Lista Detallada de Cambios

Corrección de errores

Corrección de cursores binarios rotos por implementación de MOVE (Vadim)

Correccion para fallos de la libreria tcl(Jan)
 Correccion para el manejo de matrices (arrays), por Gerhard Hintermayer
 Correccion de error en acl, y se elimina pqtrace duplicado (Bruce)
 Correccion para psql \e con fichero vacio (Bruce)
 Correccion para texcat en campos varchar() (Bruce)
 Correccion para DBT Sendproc (Zeugswetter Andres)
 Correccion para problema de sintaxis en vacuum analyze(Bruce)
 Correccion para indentificadores internacionales(Tatsuo)
 Correccion para agregaciones en tablas heredadas (Bruce)
 Correccion fr substr() para datos fuera de rango
 Correccion para select 1=1 o 2=2, select 1=1 and 2=2, y selec sum (2+) (Bruce)
 Correccion para mostrar estado de del resultado cuando no hay tty de salida. La opcion -q
 aun no lo devuelve (Bruce)
 Correccion para count(*), argumentos con vistas y tablas multiples y sum(3)(Bruce)
 Correccion para cluster (Bruce)
 Correccion para PQtrace start/stop muchas veces (Bruce)
 Correccion para una variedad de problemas como bloqueos más recientes en espera
 que tienen bloqueo antes que procesos en espera mas viejos, y teniendo gente bloqueada
 en lectura no se comparte bloqueo si proceso que escribe esta esperando para cojer el bloqueo, y los procesos escritores que esperan no obtienen prioridad sobre los
 procesos lectores que esperan(Bruce)
 Correccion para abortos en psql cuando se ejecutan consultas desde ficheros externos (James)
 Correccion para el problema de multiples ordenaciones por columnas, cuando la primera
 tiene valores NULL (Jeroen)
 Utilización correcto soporte de funciones de la tabla hash para float8 e int4(Thomas)
 Re-enable JOIN= option in CREATE OPERATOR statement (Thomas)
 Se cambia la precedencia de los operadores booleanos para emparejarlos con el comportamiento esperado(Thomas)
 Se genera elog(ERROR) en enteros que sobre-crecen (Bruce)
 Se permite argumentos multiples en las funciones de clausulas de constraint(Thomas)
 Se comprueba los literales booleanos de entrada para 'true', 'false', 'yes', 'no', '1', '0' y se envian a elog(ERROR) si no se reconocen(Thomas)
 Correccion para objetos de gran tamaño importante
 Correccion para GROUP BY mostrando duplicados(Vadim)
 Correccion para exploracion de indice en MergeJoin(Vadim)

Mejoras

Sub consultas con las palabras claves EXISTS, IN, ALL y ANY (Vadim, Bruce, Thomas)
 Nuevo Manual del Usuario(Thomas, others)
 Aceleracion por alineamiento de algunas funciones frecuentemente llamadas
 Deteccion real de bloqueos(deadlock), no mas expiraciones por tiempo(timeout)(Bruce)
 se anaden las "constantes" SQL92 CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, CURRENT_USER(Thomas)
 Se modifica la sintaxis para ser compatibles con SQL-92(Thomas)
 Se implementa las clausulas SQL PRIMARY KEY y UNIQUE utilizando indices(Thomas)
 Se reconoce la sintaxis SQL92 para CLAVES FORANEAS (FOREIGN KEYS). Se pone una
 aviso (notice) en el elog (Thomas)
 Se permite la clausula de restriccion NOT NULL UNIQUE (cada una de ellas permitidas separadamente antes)(Thomas)
 Se permite el estilo de arrojar no constantes (::") Postgres (Thomas)
 Se anade soporte para las constantes booleanas de SQL3 TRUE y FALSE (Thomas)
 Se soporta la sintaxis SQL92 para IS TRUE/IS FALSE/IS NOT TRUE/IS NOT FALSE(Thomas)
 Se permiten cadenas mas cortas para literales booleanos(por ej. "t", "tr", "tru")(Thomas)
 Se permiten identificadores de delimitacion SQL92(Thomas)
 Se implementa decodificacion para cadenas binarias y hexadecimales SQL92 (b'10' and x'11

Support SQL92 syntax for type coercion of literal strings
 (e.g. "DATETIME 'now'")(Thomas)

Se añade conversiones para tipos int2, int4, y OID a y desde texto (Thomas)

Se utilizan bloqueos compartidos cuando se crean índices (Vadim)

La memoria libre reservada para una consulta de usuario dentro de un bloque de transacción
 después de esa consulta sea hecha, se había desactivado <= 6.2.1(Vadim)

Nueva declaración SQL, CREATE PROCEDURAL LANGUAGE (Jan)

Nuevo interfaz para el motor de base de datos en el servidor (backend)Lenguaje Procedural (PL)

Postgres(Jan)

Se renombra la opción pg_dump -H a -h(Bruce)

Se añade soporte Java para contraseñas, fechas europeas (Peter)

Se utilizan índices para operaciones LIKE y ~, !~ (Bruce)

Se añade funciones hash para fecha (datetime) y marca de tiempo (timespan) (Thomas)

Se elimina Time Travel(Vadim, Bruce)

Se añade paginación para \d y \z, y se corrige \i (Bruce)

Se añade soporte para conexiones de dominio Unix (Unix domain socket) al motor de base de datos y a
 la librería del frontal(Goran)

Se implementa CREATE DATABASE/WITH LOCATION y la utilidad initlocation(Thomas)

Se permiten más palabras reservadas SQL92 y/o Postgres como identificadores de
 columnas(Thomas)

Aumenta el soporte para SQL92 SET TIME ZONE...(Thomas)

SET/SHOW/RESET TIME ZONE utilizan la variable de entorno del motor de base de datos (backend) TZ(Thomas)

Se implementa SET keyword = DEFAULT y SET TIME ZONE DEFAULT(Thomas)

Se habilita SET TIME ZONE utilizando la variable de entorno (Thomas)

Se añade la variable de entorno PGDATESTYLE a la inicialización del frontal y del motor de base de
 datos(backend)(Thomas)

Se añaden las variables de entorno PGTZ, PGCOSTHEAP, PGCOSTINDEX, PGRPLANS, PGGEQO a la librería de
 inicialización del frontal(Thomas)

Se configura automáticamente la zona horaria en los test de regresión con "setenv PGTZ PST8PDT"(Thomas)

Se añade la tabla pg_descripcion para información sobre tablas, columnas, operadores, tipos y
 agregaciones (Bruce)

Se incrementa el límite de caracteres de 16 a 32 en los nombres de tablas/índices de sistema(Bruce)

Se renombran los índices del sistema(Bruce)

Se añade la opción 'GERMAN' para SET DATESTYLE(Thomas)

Se define un "ISO-style" formato de salida de fecha extendida con los campos "hh:mm:ss"(Thomas)

Se permite valores fraccionales para tiempos delta (por ej. '2.5 días')(Thomas)

Se valida la entrada de números más cuidadosamente para tiempos delta(Thomas)

Se implementa el día del año como una entrada posible para date_part()(Thomas)

Se definen las funciones timespan_finite() y text_timespan() (Thomas)

Se elimina materia archivada(Bruce)

Se habilita una base de datos de autenticación pg_password que esta separada del fichero de contraseñas
 del sistema(Todd)

Se vuelcan los permisos ACLs, GRANT, REVOKE(Matt)

Se definen funciones de longitud de cadena text, varchar y bpchar(Thomas)

Corrección para el manejo de herencia en consultas, y computos de coste(Bruce)

Se implementa CREATE TABLE/AS SELECT (una alternativa a SELECT/INTO)(Thomas)

Se permite NOT, IS NULL, IS NOT NULL en restricciones(Thomas)

Se implementa UNIONes para SELECT(Bruce)

Se añade UNION, GROUP, DISTINCT a INSERT(Bruce)

varchar() almacena solo los bytes necesarios en disco(Bruce)

Correccion para BLOBs(Peter)
 Mega-Parche para JDBC... vease README_6.3 para lista de cambios(Peter)
 Se elimina "opcion" no utilizada de PQconnectdb()
 Nuevo comando LOCK y pagina de manual describiendo bloqueos(deadlocks)(Bruce)
 Se anaden nuevos comandos psql \da, \dd, \df, \do, \dS, and \dT(Bruce)
 Mejora de psql \z para mostrar secuencias(Bruce)
 Se muestran NOT NULL y DEFAULT en tabla psql \d (Bruce)
 Nuevo fichero de arranque de psql .psqlrc(Andrew)
 Se modifican el script de muestra del arranque en contrib/linux para mostrar syslog(Thomas)
 Nuevos tipos para direcciones IP y MAC en contrib/ip_and_mac(TomH)
 Conversiones de tiempo en sistema Unix con tipos date/time en contrib/unixdate(Thomas)
 Actualizacion del material contrib(Massimo)
 Se anade soporte para conexion Unix (Unix socket) para DBD::Pg(Goran)
 Nueva interfaz para python (PyGreSQL 2.0)(D'Arcy)
 El nuevo protocolo para frontal/motor de base de datos(backend) tiene un numero version, byte de orden de red (Phil)
 Caracteristicas de seguridad en pg_hba.conf mejorada y documentada, muchas depuraciones(Phil)
 CHAR() tiene ahora acceso mas rapido que VARCHAR() o TEXT
 preprocesador SQL embebido en ecpg
 Se reduce la carga de columnas del sistemas(Vadim)
 Se elimina la tabla pg_time(Vadim)
 Se anade el atributo pg_type para indentificar tipos que necesitan longitud (bpchar, varchar)
 Se anade informe de final de linea fuera de termino cuando el comando COPY falla
 Se permiten que los permisos en VIEW se configuren separadamente de las tabla subyacentes.
 Por seguridad, se utiliza como apropiado GRANT/REVOKE en vistas(Jan)
 Las tablas ahora no tienen por omision GRANT SELECT TO PUBLIC. Debes conceder explicitamente esos permisos.
 Limpieza de ejemplos en el tutorial(Darren)

Cambios en el Arbol Fuente

 Se anaden nuevas herramientas de desarrollo html, y caracteres flotantes en /tools/backend
 Correccion para compilacion en SCO
 Porte a computador Stratus, Robert Gillies
 Anadido soporte para shlib para BSD44_derived & i386_solaris
 Se hace la configuracion de Make mas automatizada(Brook)
 Se anade script para comprobar los resultados del test de regresion
 Se separan las funciones de verificacion sintactica en ficheros mas pequenos, agrupados juntos(Bruce)
 Se renombran heap_create a heap_create_and_catalog, rename heap_create a heap_create()(Bruce)
 Parche para bloqueos en Sparc/Linux (TomS)
 Se elimina PORTNAME y se reorganiza el material especifico de puertos(Marc)
 Se anade fichero de optimizador README(Bruce)
 Se eliminan algunas recursiones en el optimizados y se limpia algo de codigo ahi(Bruce)
 Correccion para bloqueo en NetBSD(Henry)
 Correccion para el make libptcl(Tatsuo)
 Parche para AIX(Darren)
 Cambio IS TRUE, IS FALSE, ... a la expresiones utilizando "=" mas que a llamadas a funcion
 istrue() o isfalse() para permitir optimizacion(Thomas)
 Varias correcciones relacionadas con NetBSD/Sparc(TomH)
 Bloqueos para linux Alpha(Travis,Ryan)
 Se cambian elog(ALARMAS)(WARM) a elog(ERROR)(Bruce)
 FAQ para FreeBSD(Marc)

```

Bring in the PostODBC source tree as part of our standard distribution(Marc)
Un parche menor para HP/UX vs 9(Stan)
New pg_attribute.atttypmod for type-specific info like varchar length(Bruce)
Parches para Unixware(Billy)
Nuevo 'bloqueo' i386 para giro de bloqueo en asm(Billy)

Soporte para procesos en servidor (backends) multiplexados se elimina

Comienza un porte a OpenBSD
Comienza un porte a AUX
Comienza un porte a Cygnus
Se anaden funciones de cadena para la suite de regresion(Thomas)
Se expanden unos pocos nombres de funcion anteriormente truncados a 16 caracteres(Thomas)
Emilinaada llamadas a malloc() no necesitadas y reemplazadas con palloc()(Bruce)

```

Version 6.2.1

La v6.2.1 es un a version de corrección de error y de usabilidad sobre la v6.2

Summary:

- Se permite a las cadenas extender líneas, por SQL92.
- Se incluyen ejemplos de funciones desencadenadoras(triggers) para insertar nombres de usuario en actualizaciones de tablas.

esta es una version menor de correccion de error sobre la v6.2 Para actualizaciones desde sistemas pre-v6.2, se requiere un dump/reload completo. Mire las notas de la version v6.2 para instrucciones.

Migracion desde v6.2 a v6.2.1

Esta es una version menor de correccion de error. No se requiere un dump/reload desde la v6.2, pero se requiere desde cualquier version anterior a la v6.2

Al actualizar desde v6.2, si escojes un dump/reload encontraras que avg(money) se calcula ahora correctamente. Todos las otras correcciones de errores tendran efecto al actualizar los ejecutables.

Otra manera de evitar el dump/reload es utilizar el siguiente comando SQL desde psql para actualizar la tabla de sistema existente:

```

update pg_aggregate set aggfinalfn = 'cash_div_flt8'
where aggrname = 'avg' and aggbasetype = 790;

```

Se necesita hacer esto a todas las bases de datos exitentes, incluyendo template1.

Lista Detalladas de Cambios

Cambios en esta version

Se permite bombres de columna TIME y TYPE (Thomas)

Se permite un rango grande de true/false como valores booleanos(Thomas)

Se soporta la salida de "now" y "current"(Thomas)

Manejo de DEFAULT con INSERT de NULL adecuado(Vadim)

Correccion para el proble de contar relaciones de referencia en el gestor de buffer(Vadim)

Se permite que las cadenas (strings) se extiendan a lineas, como ANSI(Thomas)

Correccion para cursor recursivo con ORDER BY(Vadim)

Correccion de computacion de avg(cash)(Thomas)

Correccion para especificacion de una columna dos veces en un ORDER/GROUP BY(Vadim)

Documentada nueva funcion de libpq que devuelve las lineas afectadas, PQcmdTuples(Bruce)

Funcion desencadenadora (trigger) para insertar nombres de usuario para INSERT/UPDATE(Brook Milligan)

Version 6.2

Se requiere un dump /restore para aquellos que deseen migrar datos desde versiones anteriores de Postgres.

Migracion desde v6.1 a v6.2

Esta version requiere un volcado (dump) completo de la base de datos en 6.1 y una retauracion de la base de datos en 6.2

Notese que la utilidad pg_dump y pg_dumpall de 6.2 deberia utilizarse para volcar (dump) la base de datos 6.1.

Migracion desde v1.x a v6.2

Aquellas migraciones desde versiones anteriores a 1.* deberian actualizarse primero a 1.09 porque el formato de salida de COPY fue mejorado desde la version 1.02.

Lista Detallada de Cambios

Correccion de errores

Correccion de problemas con pg_dump para herencia, secuencias, tablas archivo(Bruce)

Correccion de errores de compilacion en desbordamientos para shifts, unsigned y prototipos malos

de Solaris(Diab Jerius)

Correccion de errores en lineas aritmeticas geometricas (malos calculos de intersecciones)(Thomas)

Comprobacion de intersecciones geometricas en los puntos de finalizacion para evitar lo feo del redondeo(Thomas)

Se cojen tentativas no-funcionales de borrado(Vadim)

Se cambia los nombres de la funcion de tiempo para ser mas consistentes(Michael Reifenberg)

Se comprueba las divisiones por cero(Michael Reifenberg)

Correccion para error muy antiguo que hacia que tuplas cambiadas/insertadas por un comando fueran visibles para el propio comando (de ese modo teniamos multiples actualizaciones

de tuplas actualizadas, etc)(Vadim)

Correccion para SELECT null, 'fail' FROM pg_am(Patrick)

No se permite SELECT NULL como EMPTY_FIELD(Patrick)

Se elimina senas innecesario de contrib/pginterface

Correccion OR(donde x!= 1 o x sea nulo no devolvera tuplas con x NULL)(Vadim)

Correccion para la funcion time_cmp (Vadim)

Correccion en el manejo de funciones con argumentos que no tiene el atributo primero en

las clausulas WHERE (Vadim)

Correccion de GROUP BY cuando el orden de las entradas es diferente del orden en la lista de

seleccion (Vadim)

Correccion de pg_dump para agrgaciones sin sfunc1(Vadim)

Mejoras

El parametro del optimizador genetico por defecto GEQO es ahora 8(Bruce)

Se permite el uso de parametros en la lista de consulta teniendo agrgaciones en las funciones(Vadim)

Se anade driver JDBC como una interfaz(Adrian & Peter)

Utilidad pg_password

Se devuelve el numero de tuplas insertadas/afectadas por una INSERT/UPDATE/DELETE etc.(Vadim)

Los disparadores (triggers) son implementados con CREATE TRIGGER (SQL3) (Vadim)

SPI (Interfaz de Programacion en el Servidor) permite la ejecución de consultas dentro de

funciones C (Vadim)

Implementado NOT NULL (SQL92) (Robson Paniago de Miranda)

Se incluyen palabras reservadas para el manejo de cadenas (strings), uniones externas (outer

joins) y uniones (unions) (Thomas)

Implementados comentarios extendidos ("/* ... */") utilizando estados exclusivos(Thomas)

Anadido comentarios de una sola linea "///" (Bruce)

Eliminadas algunas restricciones de caracteres en nombres de operadores(Thomas)

Implementado DEFAULT y CONSTRAINT para tablas (SQL92)(Vadim & Thomas)

Anadido operador de concatenacion de texto y funcion (SQL92) (Thomas)

Soporte para sintaxis WITH TIME ZONE (SQL92) (Thomas)

Soporte para sintaxis INTERVAL unidad TO unidad (SQL92)(Thomas)

Se definen los tipos (de datos) DOUBLE PRECISION, INTERVAL, CHARACTER,

y CHARACTER VARYING (SQL92)(Thomas)

Se define el tipo FLOAT(p) y DECIMAL(p,s), NUMERIC(p,s) rudimentaria-
mente (SQL92)(Thomas)

Se define EXTRACT(), POSITION(), SUBSTRING(), y TRIM() (SQL92)(Thomas)

Se define CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP (SQL92)(Thomas)

Se anade sintaxis y avisos para UNION, HAVING, INNER y OUTER JOIN (SQL92)(Thomas)

Se anaden mas palabras reservadas, la mayoria para cumplir con SQL92(Thomas)

Se permite la entrada de tiempo hh:mm:ss para tipos de marca de tiempo/tiempo

relativo (timespan/reftime) (Thomas)

Se anaden rutinas center() para lseg, path, polygon(Thomas)

Se anaden rutinas distance() para circle-polygon, polygon-polygon(Thomas)

Se verifica explicitidad para puntos y poliginis contenidos en los poligonos

utilizando un algoritmo de cruzamiento de ejes(Thomas)

Se anade rutina para convertir circle-box(Thomas)

Se unen operadores conflictivos para diferentes tipos de datos geometricos(Thomas)

Se reemplaza el operador de distancia "<==>" con "<->"(Thomas)

Se reemplaza el operador "above", "!^" con ">^", y el operador "below", "!" con "<^"(Thomas)

Se anaden rutinas para recorte de texto en ambos extremos, subcadenas y posiciones de cadena(Thomas)

Se han anadido rutinas de conversion circle(box) and poly(circle)(Thomas)

Se permite que las ordenaciones se almacenen en memoria antes que en ficheros(Bruce & Vadim)

Se permite que funciones y operadores con tipos internamente identicos finalicen con exito(Bruce)

Speed up backend startup after profiling analysis(Bruce)

Se alinean frecuentes llamadas a funciones para mejorar las prestaciones(Bruce)

Se reducen las llamadas open() (Bruce)

psql: Se anade correccion para PAGER \h y \?,\C

Correccion para el paginador psql cuando no hay tty(Bruce)

Nueva utilidad entab (Bruce)

Funciones generales de disparadores para integridad referencial (Vadim)

Funciones generales de disparadores para tiempo transcurrido (Vadim)

Funciones generales de disparadores para la prestacion AUTOINCREMENT/IDENTITY (Vadim)

Se implemente MOVE (Vadim)

Cambios en el Arbol Fuente

Parches para HPUX 10 (Vladimir Turin)

Anadido soporte para SCO (Daniel Harris)

parches para mkLinux(Tatsuo Ishii)

Se cambia la terminologia geometrica de caja de "length" a "width"(Thomas)

Se desaprueban campos incrementales temporales no almacenados en el codigo geometrico(Thomas)

Se eliminan instrucciones de reinicio de INSTALL(Bruce)

Se mira en /usr/ucb antes de instalar(Bruce)

Correccion para el codigo de ejemplo de c++(Thomas)

Anadido -o a la pagina de manual de psql(Bruce)

Se previene que la longitud de la cadena relname no asignada sea copiada dentro de la base de datos (Bruce)

Depuracion del uso de NAMEDATALEN (Bruce)

Correccion para nombres pg_proc por encima de los 15 caracteres de salida(Bruce)

Anadida la funcion strNcpy() (Bruce)

eliminados algunos (void) repartidos que son innecesarias(Bruce)
 nuevo directorio de interfaces(Marc)
 Se remplazan las llamadas fopen() con llamadas a las funciones fd.c(Bruce)
 Se hacen las funciones estaticas donde es posible(Bruce)
 Se encierran las funciones no usadas con #ifdef NOT USED(Bruce)
 Se eliminan las llamadas a difftime() en el soporte de estampacion de fecha para corregir SunOS(Bruce & Thomas)
 Cambios para Digital Unix
 Correccion de portabilidad para pg_dumpall(Bruce)
 Se renombra pg_attribute.attnvals a attdisbursion(Bruce)
 Pagina de manual "intro/unix" ahora es "pgintro"(Bruce)
 Pagina de manual "built-in" ahora es "pgbuiltin"(Bruce)
 Pagina de manual "drop" ahora es "drop_table"(Bruce)
 Anadidas paginas de manual "create_trigger", "drop_trigger"(Thomas)
 Anadidas constraints a los test de regresion(Vadin 6 Thomas)
 Anadidos comentarios a la sintaxis del test de regresion(Thomas)
 Anadido PGIDENT y programa de soporte(Bruce)
 Commit masivo al ajecutar PGIDENT en todos los ficheros *.c y *.h(Bruce)
 Ficheros movidos al directorio /src/tools(Bruce)
 Guias de programacion de SPI y disparadores(triggers)(Vadim & D'Arcy)

Version 6.1.1

Migración desde v6.1 a v6.1.1

Esta es una versión de correcciones menores. No se necesita volcado/recarga desde la versión 6.1, pero sí para cualquier versión previa. Diríjase a las notas de versión de v6.1 para conseguir más detalles.

Lista Detallada de Cambios

Cambios en esta versión

Correcciones para SET con opciones (Thomas)
 se permite a pg_dump/pg_dumpall preservar la propiedad de todos los objetos/tablas (Bruce)
 la nueva opción \connect de psql permite cambiar el código de usuario sin cambiar de base de datos.
 corrección de la opción -debug de initdb (Yoshihiko Ichikawa)
 limpieza de lexttest(Bruce)
 correcciones en la rutina hash (Vadim)
 corrección en la aritmética de construcción del mes en fecha/hora (Thomas)
 corrección en la manipulación del horario en zonas horarias en algunas migraciones (Thomas, Bruce, Tatsuo)
 timestamp sobrecargado al utilizar funciones standard (Thomas)
 otras limpiezas del código en rutinas de fecha y hora (Thomas)
 \d de psql ahora es insensible a las mayúsculas (Bruce)
 los comandos con \ de psql ahora pueden tener ; final (Bruce)
 corrección de huecos de memoria en psql cuando se utiliza \g (Bruce)
 correcciones mayores para la manipulación final de la comunicación con el servidor (Thomas, Tatsuo)
 Correcciones para el ensamblador de Solaris y los ficheros de include (Yoshihiko Ichikawa)

pg_dumpall devuelve ahora el status correcto, corregida la portabilidad (Bruce)

Version 6.1

Las pruebas de regresión se han adaptado y modificado extensamente para la versión v6.1 de Postgres.

Se han añadido tres nuevos tipos de datos (datetime, timespan, and circle) al conjunto nativo de tipos de Postgres. Puntos, cajas, rutas y polígonos tienen sus formatos de salida consistentes por encima de los tipos de datos. La salida polígono en misc.out sólo se ha revisado para hacer correcciones relativas a la salida de regresión original.

Postgres v6.1 introduce un nuevo optimizador alternativo que utiliza algoritmos *genéticos*. Estos algoritmos introducen un comportamiento aleatorio en la ordenación de los resultados de la consulta cuando la consulta contiene múltiples calificadores o múltiples tablas (dando el optimizador una elección en el orden de evaluación). Se han modificado varias pruebas de regresión para ordenar explícitamente el resultado, y hacerlo así insensible a las elecciones del optimizador. Unas pocas pruebas de regresión corresponden a tipos de datos que son inherentemente desordenados (como puntos o intervalos de tiempo), y las pruebas que involucran estos tipos se fuerzan explícitamente con **set geqo to 'off'** y **reset geqo**.

La interpretación de los especificadores de vectores (los corchetes alrededor de valores atómicos) parece haber cambiado alguna vez tras las pruebas de regresión originales. Los ficheros actuales `./expected/*.out` reflejan esta nueva interpretación, ¡que puede no ser correcta!

Las pruebas de regresión con float8 fallan al menos en algunas plataformas. Esto se debe a las diferencias en las implementaciones de `pow()` y `exp()` y a los mecanismos de señalización utilizados para las condiciones de sobrecarga y subcarga.

Los resultados "aleatorios" en la prueba aleatoria deberían provocar que la prueba "aleatoria" resultase "fallida", puesto que los test de regresión se evalúan con un simple diff. Sin embargo, la prueba "aleatoria" no parece producir resultados aleatorios en mi máquina de pruebas (Linux/gcc/i686).

Migración a v6.1

Esta migración requiere un volcado completo de la base de datos 6.0 y su restauración en la base de datos en 6.1.

Aquellos que quieran migrar desde versiones iniciales 1.* deberían primero actualizarse a 1.09, porque el formato de salida de COPY se mejoró en la versión 1.02.

Lista Detallada de Cambios

Corrección de errores

comprobaciones en la longitud empaquetada en rutinas de la librería
 parche de prioridad en el gestor de bloqueos
 comprobaciones para sub/sobrecarga de float8 (Bruce)
 correcciones en cruces de múltiples tablas (Vadim)
 corrección de un aborto de SIGPIPE (Darren)

correcciones sobre objetos grandes (Sven)
 se permite que los índices btree manipulen NULL,s (Vadim)
 correcciones en timezone (D'Arcy)
 select SUM(x) puede devolver NULL cuando no hay filas (Thomas)
 se corrigen errores del optimizador interno y del ejecutor (Vadim)
 se corrigen problemas cuando bucles internos en < o <= no tienen filas (Vadim)
 se previene la re-inversión de cláusulas join de índices (Vadim)
 corregida la clausula join para múltiples tablas (Vadim)
 corregidos hash y hashjoin para vectores (Vadim)
 corrección en btree para el tipo abstime (Vadim)
 correcciones para objetos grandes (Raymond)
 corregidos huecos en el buffer en índices hash (Vadim)
 corregido rtree para el uso de barridos internos (Vadim)
 correcciones esenciales para el uso de barridos internos, limpiezas (Vadim, Andrea)
 se impide la colocación de buffers locales innecesarios (Vadim, Massimo)
 corregidos huecos de buffers locales en abortos de transacciones (Vadim)
 corregidos huecos de memoria del gestor de ficheros, limpiados (Vadim, Massimo)
 corregidos huecos de memoria del gestor de almacenamiento (Vadim)
 corregido que btree duplique la manipulación (Vadim)
 corregida la reencarnación de tuplas borradas causada por vacuum (Vadim)
 corregido SELECT varchar()/char() INTO TABLE hace campos de longitud cero (Bruce)
 corregidas muchos huecos de memoria de psql, pg_dump y libpq utilizando Purify (Igor)

Mejoras

estadísticas de optimización de atributos (Bruce)
 nuevo código de carga masivo btree mucho más rápido (Paul)
 añadido BTREE UNIQUE para código de carga masiva (Vadim)
 nuevo aspecto del código de depuración (Massimo)
 cambios masivos en libpq++ (Leo)
 el nuevo optimizador GEQO acelera la optimización en tablas multi-tabla (Martin)
 nuevo mensaje de alarma para inserciones no únicas en claves únicas (Marc)
 update x=-1, sin espacios, ahora es válido (Bruce)
 se elimina la manipulación de identificadores sensibles a las mayúsculas (Bruce, Thomas, Dan)
 la depuración del servidor ahora imprime un árbol agradable (Darren)
 nuevas funciones de caracteres de Oracle (Edmund)
 nuevas funciones de palabra clave con texto plano (Dan)
 se cambian a mensajes diferentes no tal clase y insuficientes privilegios (Dan)
 nueva función ANSI timestamp (Dan)
 nuevos tipos ANSI Time y Date (Thomas)
 mueve grandes grupos de datos en el servidor (Martin)
 índices btree multicolumna (Vadim)
 nuevo comando SET var TO valor (Martin)
 status de transacción de actualización en las lecturas (Dan)
 nuevos ajustes locales para tipos de caracteres (Oleg)
 nuevo generador de series de números SEQUENCE (Vadim)
 ahora es posible GROUP BY una función (Vadim)
 reorganizada la prueba de regresión (Thomas, Marc)
 nuevos pesos de operaciones en el optimizador (Vadim)
 nueva opción psql \z grant/permit (Marc)
 nuevo tipo de datos MONEY (D'Arcy, Thomas)
 incrementada la velocidad de comunicación por el socket tcp (Vadim)
 nueva opción VACUUM para estadísticas de atributos, y para ciertas columnas (Vadim)
 muchas potenciaciones de tipos geométricos (Thomas, Keith)
 Pruebas de regresión adicionales (Thomas)
 nuevas variables de estilo de datos (Thomas, Vadim, Martin)

más operadores de comparación para ordenar tipos (Thomas)
 nuevas funciones de conversión (Thomas)
 no más formato btree compacto (Thomas)
 se permite a pg_dumpall preservar la propiedad de las bases de datos (Bruce)
 nuevas variables SET GEQO=# y R_PLANS (Vadim)
 el viejo optimizador (!GEQO) puede utilizar planes del lado derecho (VADIM)
 mejorado el control de tipos en el traductor de SQL (Bruce)
 nuevos comandos SET, SHOW y RESET (Thomas, Vadim)
 nueva opción USER \connect basededatos
 nueva opción destroydb -i (Igor)
 nuevos comandos de psql \dt y \di (Darren)
 SELECT "\n" ahora genera nueva línea (A. Duursma)
 nuevas funciones de conversión de la geometría desde el formato anterior (Thomas)

Cambios en el árbol fuente

nuevo guión de configuración (Marc)
 añadida la opción de configuración de lectura de línea (Marc)
 nuevos ficheros de plantillas específicas del Sistema Operativo (Marc)
 ya no se necesita editar Makefile.global (Marc)
 se reordenan los ficheros de include (Marc)
 parche para nextstep (Gregor Hoffleit)
 se elimina código específico de WIN32 (Bruce)
 se elimina la opción -e de postmaster, se mantiene sólo la opción postgres -e (Bruce)
 se mezcla el código de librerías duplicadas en clientes y servidores (Martin)
 ahora trabaja con eBones, Kerberos internacional (Jun)
 más soporte a librerías compartidas
 c++ incluye limpieza de fichero (Bruce)
 Aviso sobre flex erróneo (Bruce)
 Correcciones en la portabilidad para DG-UX, Ultrix, Irix, AIX.

Version v6.0

Aquellos que quieran migrar datos desde versiones previas de Postgres necesitarán hacer un volcado/recuperación.

Migración desde v1.09 a v6.0

Esta migración necesita un volcado completo de la base de datos 1.09 y una recuperación de los datos en 6.0.

Migración desde versiones previas a v1.09 hasta v6.0

Quienes quieran migrar desde las iniciales versiones 1.* deberían actualizarse primero a la versión 1.09, ya que el formato de salida de COPY se mejoró a partir de la versión 1.02.

Lista Detallada de Cambios

Corrección de errores

Error ALTER TABLE - corriendo el proceso postgres se necesita re-leer la definición de la tabla.

Se permite que vacuum se ejecute sobre una tabla o sobre la base entera (Bruce)

Correcciones en tablas.

Corregido una sobre-escritura en tabla en escritura de memoria (Kurt)

Corregido un error en btree elusivo en rango/no en rango (Dan)

Correcciones en los índices hash para algunos tipos como time y date.

Correcciones para la explosión del tamaño de pg_log.

Corregidos los permisos en lo_export()(Bruce).

Corregidas lecturas no inicializadas de memoria (Kurt).

Corregido un error ALTER TABLE ... char(3) (Bruce)

Corregidas una pocas lagunas de memoria pequeñas.

Corregida la manipulación de EXPLAIN de opciones y cambiado el nombre de opción del path completo.

Corregida la salida de permisos de grupos de acl

Lagunas en la memoria (localizadas y eliminadas con herramientas como Purify (Kurt))

Mejoras menores de las reglas del sistema.

Se corrige NOTIFY

Nuevas instrucciones para ejecutar-comprobar.

Repaso general del código del analizador/traductor para informar correctamente de los errores e incrementar la velocidad.

Pg_dump -d ahora manipula correctamente los NULL (Bruce)

Se evita que SELECT NULL mate el servidor (Bruce)

Se informan adecuadamente errores cuando las columnas de INSERT ... SELECT no casan.

Se informan adecuadamente errores cuando se están insertando nombres de columna que no son correctos.

Psql \g nombrefichero ahora trabaja (Bruce)

Corregido un problema de psql con instrucciones múltiples en una línea con múltiples salidas.

Eliminados oid's de sistema duplicados

SELECT * INTO TABLE . GROUP/ORDER BY daba un error de enlace si la tabla existía (Bruce)

Varias correcciones a consultas que mataban el servidor

Las comillas al principio de una cadena a insertar produce un error (Bruce)

El lanzamiento de una consulta vacía ahora devuelve un status de vacío, no sólo la consulta " " (Bruce)

Mejoras

Se añade una página de manual para EXPLAIN (Bruce)

Se añade la capacidad de índice UNIQUE (Dan)

Se añade control de acceso vigilando nombre_host/usuario, más que sólo nombre_host y usuario.

Se añade el sinónimo != para <> (Bruce)

Se permite "select oid,* from table"

Se permite a ORDER BY especificar columnas por número, o por tabla.columna que no son alias (Bruce)

Se permite el comando COPY desde la aplicación cliente (Bryan)

Se permite a GROUP BY que utilice alias de nombres de columnas (Bruce)

Se permite la compresión actual, no sólo en la misma página (Vadim)

Se permite la opción de instalación-configuración para auto-ayudar a todos los usuarios locales (Bryan)

Se permite a libpq que distinga entre textos con valor " y nulo (Bruce)
 Se permite a los usuarios diferentes de postgres con privilegios de createdb ejecutar destroydb.
 Se permiten restricciones sobre quién puede crear funciones C (Bryan)
 Se permiten restricciones sobre quien puede hacer COPY del servidor (Bryan)
 Se pueden reducir tablas, pg_timer y pg_log (Vadim & Erich)
 Cambiado el nivel de debug 2 para imprimir sólo consultas, cambiado el formato de
 la cabecera del debug (Bruce)
 Se cambia la representación de las constantes decimales desde float4 a float8 (Bruce)
 Ahora se fija el formato de fecha europeo cuando se arranca el postmaster.
 Se ejecutan las funciones con el nombre en minúscula si no se encuentran con el nombre exacto.
 Las correcciones del procesamiento de agregados/GROUP, permiten
 'select sum(func(x),sum(x+y) from z'
 Gist está ahora incluido en la distribución (Marc)
 Autenticación Idendde usuarios locales (Bryan)
 Se implementa el calificador BETWEEN (Bruce)
 Se implementa el calificador IN (Bruce)
 Libpq tiene PQgetisnull()(Bruce)
 Mejoras de Libpq++
 Nuevas opciones en initdb(Bryan)
 Pg_dump permite volcar los oid's (Bruce)
 Pg_dump crea los índices tras cargar las tablas, por velocidad (Bruce)
 Pg_dumpall vuelca todas las bases de datos, y la tabla de usuarios.
 Adiciones a Pginterface para los valores NULL (Bruce)
 Se previene la ejecución de postmaster como root
 \h y \? son ahora legibles (Bruce)
 Psql permite punto y coma escapados (\;) en cualquier parte de la línea (Bruce)
 Se cambia el prompt de comandos de Psql para líneas intermedias en consultas o en líneas entre comillas (Bruce)
 Las variables char(3) de Psql se muestran ahora como (bp)char en salidas \d (Bruce)
 El código de retorno de Psql es ahora más ajustado (Bryan?)
 Se actualiza la sintaxis de la ayuda de Psql (Bruce)
 Se re-visita y corrige vacuum (Vadim)
 Se reduce el tamaño de las diferencias de regresión, se elimina la diferencia del nombre de la zona horaria (Bruce)
 Se eliminan parámetros de tiempo de compilación para capacitar distribuciones binarias (Bryan)
 Gestión inversa de máscaras HBA (Bryan)
 Autenticación segura de usuarios locales (Bryan)
 Se incrementa la seguridad de vacuum (Vadim)
 Vacuum ahora tiene opción VERBOSE (Bruce)

Cambios en el árbol fuente

Todas las funciones tienen ahora prototipos que se comparan contra las llamadas.
 Se permite inhabilitar fácilmente las declaraciones en Makefile.global (Bruce).
 Se cambian las constantes oid utilizadas en el código para los nombres de #define
 Se desacoplan las defines de sparc y solaris (Kurt)
 gcc -Wall compila limpiamente con avisos (warnings) sólo a partir de construcciones no corregidas.
 Gran reorganización/reducción del fichero de include (Marc).
 Make ahora para en fallos de compilación (Bryan)
 Reestructuración del Makefile (Bryan, Marc).
 Se mezcla bsd_2_1 con bsd (Bruce)
 Se elimina el programa Monitor-
 Se cambia el nombre de Postgres95 a PostgreSQL
 Nuevo fichero config.h (Marc, Bryan)
 PG_VERSION se fija ahora a 6.0 y lo utiliza el postmaster.
 Adiciones a la portabilidad, incluyendo Ultrix, DG/UX, AIX, y Solaris

Se reduce el número de #define's, se centralizan las #define's
 Se eliminan OIDS duplicadas en las tablas del sistema(Dan)
 Se elimina información duplicada en el catálogo del sistema o errores de informe(Dan)
 Se eliminan muchas #define's específicas del sistema operativo.
 Generación/localización del fichero de objetos reestructurada(Bryan, Marc)
 Reestructuradas las localizaciones de ficheros específicas de la
 migración(Bryan, Marc)
 Corregidas variables no utilizadas/no inicializadas.

Version v1.09

Lo siento, hemos parado de registrar los cambios desde 1.02 a 1.09. Algunos de los cambios listados en 6.0 estaban ya incluidos en las versiones 1.02.1 a 1.09.

Version v1.02

Migración de v1.02 a v1.02.1

Aquí tenemos un nuevo fichero de migración para 1.02.1. Este el cambio de 'copy' y un guión para convertir los antiguos ficheros ascii.

Nota: Las siguientes notas son para el beneficio de los usuarios que quieren migrar bases de datos desde postgres95 1.01 y 1.02 a postgres95 1.02.1.

Si está usted arrancando con postgres95 1.02.1 de nuevas y no necesita migrar una base de datos antigua, no necesita leer lo que sigue.

Para actualizar anteriores bases de datos postgres95 versiones 1.01 o 1.02 a la versión 1.02.1, se requieren los siguientes pasos:

1. Arranque un nuevo postmaster 1.02.1
2. Añadi las nuevas funciones y operadores incluidos de 1.02.1 a bases de datos 1.02 o 1.02. Esto se hace ejecutando el nuevo servidor 1.02.1 contra su propia base de datos 1.01 o 1.02, y aplicando las consultas incluidas al final de este fichero. Se puede hacer esto muy facilmente con psql. Si su base de datos 1.01 o 1.02 se llama "testdb", y ha cortado los comandos del final de este fichero y los ha salvado en addfunc.sql:

```
% psql testdb -f addfunc.sql
```

Aquellos que estén actualizado bases de datos 1.02 obtendrán un aviso cuando ejecuten las dos últimas instrucciones en el fichero, pues ya están presentes en 1.02. No hay motivo para preocuparse.

Procedimiento de Volcado/Recarga Procedure

Si está intentando recargar un pg_dump o 'copy tablename to stdout' en modo texto generados con una versión previa, necesitará ejecutar el guión de sed siguiente sobre el fichero ASCII antes de cargarlo en la base de datos. El formato antiguo utilizaba ';

como end-of-data, mientras que el nuevo marcador de end-of-data (fin de los datos) es '\.'. También, las cadenas vacías se cargan ahora como "" en lugar de como NULL. Vea la página del manual de copy para obtener detalles completos.

```
sed 's/^\.$/\\./g' <in_file >out_file
```

Si está usted cargando una copia binaria más vieja, o una copia que no procede de stdout, no hay caracter end-of-data, y por ello no se necesita conversión.

```
- following lines added by agc to reflect the case-insensitive
- añadidas las siguientes líneas por agc para que no sea sensible a las mayúsculas
- regexp searching for varchar (in 1.02), and bpchar (in 1.02.1)
- regexp buscando varchar (en 1.02) y bpchar (en 1.02.1)
create operator ~* (leftarg = bpchar, rightarg = text, procedure = texticregexeq);
create operator !~* (leftarg = bpchar, rightarg = text, procedure = texticregexne);
create operator ~* (leftarg = varchar, rightarg = text, procedure = texticregexeq);
create operator !~* (leftarg = varchar, rightarg = text, procedure = texticregexne);
```

Lista Detallada de Cambios

Mantenimiento y desarrollo del código fuente.

- * equipo de voluntarios extendido por todo el mundo.
- * el árbol fuente se encuentra ahora en CVS en ftp.ki.net

Mejoras

- * psql (y la librería libpq que subyace) tiene ahora muchas más opciones para dar forma a sus salidas, incluyendo HTML
- * pg_dump ahora extrae el esquema y/o los datos, con muchas mejoras para asegurar que se complete.
- * se utiliza psql en lugar de monitor en los guiones de órdenes de administración. monitor será despreciado en la siguiente versión.
- * mejoradas las funciones de fecha/hora
- * la comparación/inserción/actualización de NULL corregidos/potenciados.
- * la librería y el interprete de órdenes de TCL/TK corregidos para que trabajen tanto con tcl7.4/tk4.0 como con tcl7.5/tk4.1

Errores corregidos (aunque demasiados numerosos para mencionarlos)

- * índices
- * gestión de almacenamiento
- * comprobación de punteros a NULL antes de dereferenciarlos
- * Correcciones en el Makefile.

Nuevas Migraciones

- * añadida la migración a SolarisX86
- * añadida la migración a BSDI 2.1
- * añadida la migración a DGUX

Version v1.01

Migración desde v1.0 a v1.01

Las siguientes notas son para beneficio de los usuarios que quieren migrar bases de datos de postgres95 1.0 a postgres95 1.01

Si está usted arrancando de nuevas con postgres95 1.01 y no necesita migrar bases de datos anteriores, no necesita usted leer lo siguiente.

Para que postgres95 versión 1.01 funcione con bases de datos creadas con postgres95 versión 1.0, se requieren los siguientes pasos:

1. Fije la definición de NAMEDATALEN en src/Makefile.global a 16 y OIDNAMELEN a 20.
2. Decida si quiere usted autenticación basada en el ordenador.
 - a. Si lo hace, debe usted crear un fichero llamado "pg_hba" en su directorio de datos de nivel superior (típicamente el valor de su \$PGDATA). src/libpq/pg_hba muestra un ejemplo de sintaxis.
 - b. Si no quiere autenticación basada en el ordenador, puede usted comentar la línea

```
HBA = 1
```

en src/Makefile.global

Compruebe que la autenticación basada en el ordenador se ha activado por defecto, y si no sigue los pasos A o B anteriores, el out-of-the-box 1.01 no le permitirá conectar a las bases de datos 1.0

3. Compile e instale 1.01, pero NO ejecute el paso initdb.
4. Antes de hacer ninguna otra cosa, pare su postmaster 1.0, y respalde su directorio \$PGDATA existente.
5. Fije su variable de entorno PGDATA a sus bases de datos 1.0, pero fijela de forma que los binarios 1.01 sean los que se utilizan.
6. Modifique el fichero \$PGDATA/PG_VERSION de 5.0 a 5.1
7. Arranque un nuevo postmaster 1.01
8. Añada las nuevas funciones y operadores incluidas en 1.01 sobre bases de datos 1.0. Esto se hace ejecutando el nuevo servidor 1.01 contra su propia base de datos 1.0, y aplicando las consultas unidas y salvadas en el fichero 1.0_to_1.01.sql. Se puede hacer facilmente desde psql. Si su base de datos se llama "testdb":

```
% psql testdb -f 1.0_to_1.01.sql
```

y entonces ejecute los siguientes comandos(copiar y pegar desde aquí):

- funciones incluidas añadidas que son nuevas en 1.01

```
create function int4eqoid (int4, oid) returns bool as 'foo'
language 'internal';
create function oideqint4 (oid, int4) returns bool as 'foo'
language 'internal';
create function char2icregexeq (char2, text) returns bool as 'foo'
language 'internal';
create function char2icregexne (char2, text) returns bool as 'foo'
language 'internal';
create function char4icregexeq (char4, text) returns bool as 'foo'
```

```

language 'internal';
create function char4icregexne (char4, text) returns bool as 'foo'
language 'internal';
create function char8icregexeq (char8, text) returns bool as 'foo'
language 'internal';
create function char8icregexne (char8, text) returns bool as 'foo'
language 'internal';
create function char16icregexeq (char16, text) returns bool as 'foo'
language 'internal';
create function char16icregexne (char16, text) returns bool as 'foo'
language 'internal';
create function texticregexeq (text, text) returns bool as 'foo'
language 'internal';
create function texticregexne (text, text) returns bool as 'foo'
language 'internal';

```

- funciones incluidas añadidas que son nuevas en 1.01

```

create operator = (leftarg = int4, rightarg = oid, procedure = int4eqoid);
create operator = (leftarg = oid, rightarg = int4, procedure = oideqint4);
create operator ~* (leftarg = char2, rightarg = text, procedure = char2icregexeq);
create operator !~* (leftarg = char2, rightarg = text, procedure = char2icregexne);
create operator ~* (leftarg = char4, rightarg = text, procedure = char4icregexeq);
create operator !~* (leftarg = char4, rightarg = text, procedure = char4icregexne);
create operator ~* (leftarg = char8, rightarg = text, procedure = char8icregexeq);
create operator !~* (leftarg = char8, rightarg = text, procedure = char8icregexne);
create operator ~* (leftarg = char16, rightarg = text, procedure = char16icregexeq);
create operator !~* (leftarg = char16, rightarg = text, procedure = char16icregexne);
create operator ~* (leftarg = text, rightarg = text, procedure = texticregexeq);
create operator !~* (leftarg = text, rightarg = text, procedure = texticregexne);

```

Lista Detallada de Cambios

Incompatibilidades:

- * 1.01 es compatible hacia atrás con 1.0 si el usuario sigue los pasos marcados en el fichero `MIGRATION_from_1.0_to_1.01`.
En caso contrario, 1.01 no es compatible con la base de datos 1.0

Potenciaciones:

- * se añade `PQdisplayTuples()` a `libpq` y se modifican `monitor` y `psql` para que lo utilicen.
- * se añade la migración a `NEXT` (requiere la implementación de `SysVIPC`)
- * se añade la sintaxis `CAST .. AS ...`
- * se añaden las palabras clave `ASC` y `DESC`
- * se añade `'internal'` como un posible lenguaje para `CREATE FUNCTION`
las funciones internas son funciones en C que se han linkado estáticamente en el servidor de postgres.
- * se ha añadido un nuevo tipo `"name"` para identificadores de sistema (nombres de tablas, nombres de atributos, etc.) Esto reemplaza al viejo tipo `char16`. La longitud del nombre se fija en la definición (`#define`) `NAMEDATELEN` de `src/Makefile.global`
- * un manual de referencia legible que describe el lenguaje de consulta.
- * se ha añadido un control de acceso basado en el ordenador. Se utiliza un fichero de configuración (`$PGDATA/pg_hba`) para almacenar los datos de configuración.

Si el control de acceso basado en el ordenador no es deseable, comente HBA=1 en src/Makefile.global.

- * cambiada la manipulación de reges para hacerla uniforme con el código regex de Henry Spencer sin mirar la plataforma. Es código regex está incluido en la distribución.
- * añadidas funciones y operadores para expresiones regulares insensibles a las mayúsculas.
- Los operadores son ~* y !~*.
- * pg_dump utiliza COPY en lugar de un bucle de SELECT para incrementar el rendimiento.

Errores corregidos:

- * corregido un error del optimizador que provocaba volcados de memoria cuando se utilizaban llamadas a funciones en comparaciones en la cláusula WHERE.
- * se cambian todos los usos de getuid a geteuid se modo que se utilicen los uids efectivos.
- * psql devuelve ahora status distintos de cero en los errores cuando se usa -c
- * se aplican los parches públicos 1-14

Version v1.0

Lista Detallada de Cambios

Cambio en el Copyright:

- * Se ha aflojado el copyright de Postgres 1.0 para que sea libremente modificable para cualquier propósito. Lea por favor el fichero COPYRIGHT.
- Gracias al Profesor Michael Stonebraker por hacerlo posible.

Incompatibilidades:

- * los formatos de fecha tienen que ser MM-DD-YYYY (o DD-MM-YYYY si está usted utilizando EUROPEAN STYLE). Esto sigue las especificaciones SQL-92.
- * "delimiters" es ahora una palabra clave.

Potenciaciones:

- * se ha añadido la sintaxis LIKE de sql
- * el comando copy ahora recibe una especificación opcional USING DELIMITER. Los delimitadores pueden ser cualquier cadena de un único carácter.
- * se ha añadido la migración a IRIX 5.3.
- Gracias a Paul Walmsley y otros.
- * se ha actualizado pg_dump para trabajar con una nueva libpq.
- * se ha añadido \d a psql
- Gracias a Keith Parks
- * se ha incrementado el rendimiento de regexp para arquitecturas que utilizan regex de POSIX regex debido a la memorización (caching) de patrones precompilados.
- Gracias a Alistair Crooks
- * una nueva versión de libpq++
- Gracias a William Wanders

Errores corregidos:

- * se pueden especificar usuarios arbitrarios en el guión (script) createuser.
- * ya funciona \c para conectar a otras bases de datos.
- * se ha corregido una mala entrada en pg_proc para float4inc()
- * los usuarios con el campo usecreatedb fijado, ya pueden crear bases de datos sin ser el superusuario.
- * se eliminan las entradas del control de acceso cuando la entrada ya no tiene ningún permiso.
- * corregida la implementación de formatos de fecha/hora no portables.
- * añadidas marcas (banderas=flags) kerberos en src/backend/Makefile
- * libpq ya trabaja con kerberos.
- * se han corregido errores tipográficos en el manual de usuario.
- * btree con índices múltiples no ha trabajado nunca, pero ahora le decimos que no trabajan cuando intenta utilizarlo.

Postgres95 Beta 0.03

Lista Detallada de Cambios

Cambios incompatibles:

- * BETA-0.3 ES INCOMPATIBLE CON BASES DE DATOS CREADAS CON VERSIONES PREVIAS (debido a cambios en el catálogo del sistema y a cambios en la estructura de los índices).
- * las dobles comillas (") se desprecian como un carácter de limitación para cadenas literales; necesitarás convertirlas a apostrofes (').
- * el nombre de los agregados (como int4sum) se renombran de acuerdo con el SQL estándar (por ejemplo sum).
- * la sintaxis CHANGE ACL se reemplaza por la sintaxis GRANT/REVOKE.
- * los literales flotantes (como 3.14) son ahora del tipo float4 (en lugar del float8 de versiones anteriores); deberá realizar un transformado de tipos si su instalación depende de que siga siendo float8.
¡Si rechaza realizar el transformado de tipos, y asigna un literal flotante a un campo de tipo float8, es posible que los valores almacenados sean incorrectos!
- * se ha recompuesto totalmente LIBPQ para que las aplicaciones cliente (frontend) puedan conectarse a múltiples servidores (backend).
- * el campo usesysid de pg_user se ha cambiado de int2 a int4 para permitir mayores rangos de identificadores de usuarios de Unix.
- * los portes a los sistemas operativos netbsd/freebsd/bsd se han consolidado en un único port derivado de BSD44. (Gracias a Alistair Crooks).

Cumplimento del estándar SQL (los siguientes detalles cambian para hacer a postgres95

más ajustado al estándar SQL-92):

- * se han incluido los siguientes tipos SQL: smallint, int(eger), float, real, char(N), varchar(N), date y time.

Los siguientes son alias de los tipos postgres existentes:

```

        smallint -> int2
        integer, int -> int4
        float, real -> float4
    char(N) y varchar(N) se han implementado como tipos text truncados.
    Además, char(N) rellena a blancos el espacio no utilizado.
    * se utiliza el apóstrofe (') para limitar cadenas literales; " (además de \') se soportan
    para permitir insertar un único límite en una cadena.
    * se utilizan los nombres de agregados de SQL estándar (MAX, MIN, AVG, SUM, COUNT)
    (También, se pueden ahora sobrecargar los agregados, es decir, puede usted definir
    su propio agregado MAX para disponer de un tipo definido por el usuario).
    * se ha eliminado CHANGE ACL. Se añade la sintaxis GRANT/REVOKE.
    - Se pueden dar privilegios a un grupo utilizando la palabra clave "GROUP".
      Por ejemplo:
          GRANT SELECT ON mi_tabla TO GROUP mi_grupo;
    La palabra clave 'PUBLIC' también está soportada para autorizar a
    todos los usuarios.

```

Sólo se pueden otorgar o retirar privilegios a un usuario o grupo cada vez.

"WITH GRANT OPTION" no está soportado. Sólo los propietarios de clases pueden cambiar el control de acceso.

- El control de acceso de defecto es autorizar a los usuarios sólo a leer. Deberá usted autorizar explícitamente el acceso en inserción/actualización a los usuarios. Para cambiar esto, deberá modificar la línea


```
src/backend/utils/acl.h
```

 que define ACL_WORLD_DEFAULT

Errores corregidos:

- * se ha corregido el error según el cual los agregados de tablas vacías no trabajaban adecuadamente. Ahora, los agregados ejecutados sobre tablas vacías devuelven las condiciones iniciales de los agregados. Así, COUNT de una tabla vacía devuelve correctamente el valor 0. MAX/MIN de una tabla vacía devolverá una tupla de valor NULL.
- * se permite el uso de \; dentro del monitor.
- * el mecanismo de notificación asíncrono LISTEN/NOTIFY ya trabaja.
- * NOTIFY en el cuerpo de las reglas de acción ya trabaja.
- * ya funcionan los índices hash, y los métodos de acceso en general deberían funcionar mejor. La creación de grandes índices btree debería ser mucho más rápida. (Gracias a Paul Aoki).

Otros cambios y potenciaciones:

- * se añade la instrucción EXPLAIN utilizada para analizar el plan de ejecución de una consulta. (es decir: "EXPLAIN SELECT * FROM EMP" muestra el plan de ejecución de la consulta).
- * los mensajes WARN y NOTICE ya no muestran tiempo de ejecución en sí mismos. Para activar el tiempo de ejecución en los mensajes de error, descomente en


```
src/backend/utils/elog.h:
```

 la línea


```
/* define ELOG_TIMESTAMPS */
```
- * En una violación del control de acceso, se dará el mensaje


```
"Either no such class or insufficient privilege"
```

 Este es el mismo mensaje que se devuelve cuando no se encuentra una clase. Esto

disuade a los usuarios no privilegiados de sospechar la existencia de clases privilegiadas.

- * se han hecho algunos cambios adicionales en el catálogo del sistema que no son visibles para el usuario.

cambios en libpgtcl:

- * se añade la opción `-oid` al comando de tcl `"pg_result"`. `pg_result -oid` devolverá el oid de la última tupla insertada. Si el último comando no fue una inserción, `pg_result -oid` devuelve `""`.
- * el interface de objetos largos está utilizable como comandos tcl `pg_lo*`: `pg_lo_open`, `pg_lo_close`, `pg_lo_creat`, etc.

Potenciaciones de la Portabilidad y Nuevas Migraciones:

- * Se han limpiado problemas con flex/lex. Ahora, se debería poder utilizar flex en lugar de lex en cualquier plataforma. Ya se harán asunciones sobre la forma de analizador sintáctico elegido basadas en la plataforma que utilice.
- * Ahora se soporta la migración a Linux-ELF. Se han probado varias configuraciones: Se sabe que la siguiente configuración funciona:
kernel 1.2.10, gcc 2.6.3, libc 4.7.2, flex 2.5.2, bison 1.24
todo en formato ELF.

Nuevas utilidades:

- * `ipcclean` añadido a la distribución
habitualmente no se necesita ejecutar `ipcclean`, pero si cae su servidor y deja segmentos de memoria ocupados, `ipcclean` los limpiara para usted.

Nueva documentación:

- * se ha revisado el manual del usuario y se ha añadido la documentación de `libpq`.

Postgres95 Beta 0.02

Lista Detallada de Cambios

Cambios incompatibles:

- * La declaracion SQL para crear una base de datos en `'CREATE DATABASE'` en lugar de `'CREATEDB'`. De modo similar, el borrado de una base de datos en `'DROP DATABASE'` en lugar de `'DESTROYDB'`. Sin embargo los nombres de los ejecutables `'createdb'` y `'destroydb'` permanecen igual.

Nuevas herramientas:

- * `pgperl` - una interfaz Perl (4.036) para Postgres95
- * `pg_dump` - una utilidad para volcar una base de datos postgres en un fichero
fichero de script conteniendo los comandos de creacion. Los ficheros script estan en formato ASCII y pueden ser usados para reconstruir una base de datos, incluso

```

en otras maquinas y otras arquitecturas. (Tambien es bueno pa-
ra convertir
una base de datos Postgres 4.2 a base de datos Postgres95.)

```

Los siguientes portes han sido incorporados en postgres95-beta-0.02:

- * el porte a NetBSD por Alistair Crooks
- * el porte a AIX por Mike Tung
- * el porte a Windows NT por Jon Forrest (hay mas gente pero aun no han he-cho nada)
- * el porte a Linux ELF por Brian Gallew

Los siguientes errores han sido corregidos en postgres95-beta-0.02:

- * lineas nuevas no acaban en escape en COPY OUT y problemas con COPY OUT cuando la primera linea es un '.'
- * no se puede escribir un "retur" para utilizar el id del usuario por omi-sion en "createuser"
- * SELECT DISTINCT en tablas grandes aborta
- * Problemas en la instalacion en Linux
- * monitor no acepta el uso de 'localhost' como PGHOST
- * psql genera un volcado de memoria cuando ejecuta \c or \l
- * la etiqueta "pgtclsh" desaparecida de src/bin/pgtclsh/Makefile
- * libpgtcl tiene un número de puerto de defecto codificado en duro
- * SELECT DISTINCT INTO TABLE se cuelga
- * CREATE TYPE no acepta 'variable' como longitud interna ("internallenght")
- * resultados incorrectos utilizando mas de 1 agregado en una SELECT

Postgres95 Beta 0.01

Version inicial.

Tiempos Resultantes

Estos son los tiempos resultantes de ejecutar los test de regresion con los comandos

```

% cd src/test/regress
% make all
% time make runtest

```

Los tiempos bajo Linux 2.0.27 parecen tener una variacion de aproximadamente 5% de ejecucion a ejecucion, presumiblemente debido a vaguedades de planificacion en los sistemas multitareas.

v6.5

Como ha sido el caso para versiones precedentes, los tiempos entre versiones no son directamente comparables puesto que se han añadido nuevos test de regresion. En general, la v6.5 es mas rapida que versiones precedentes.

Tiempo con `fsync()` desabilitado:

Tiempo	Sistema
--------	---------

```

02:00 Dual Pentium Pro 180, 224MB, UW-SCSI, Linux 2.0.36, gcc 2.7.2.3 -
O2 -m486
04:38 Sparc Ultra 1 143MHz, 64MB, Solaris 2.6

```

Tiempos con `fsync()` habilitado:

```

Tiempo Sistema
04:21 Dual Pentium Pro 180, 224MB, UW-SCSI, Linux 2.0.36, gcc 2.7.2.3 -
O2 -m486

```

Para el sistema Linux anterior, la utilizacion de discos UW-SCSI mejores que los (viejos) IDE conducen a una mejora de un 50% en la velocidad de los test de regresion.

v6.4beta

Los tiempos para esta version no son directamente comparables a aquellos de versiones precedentes puesto que algunos test de regresion adicionales han sido incluidos. No obstante, en general, la v6.4 puede ser levemente mas rapida que versiones precedentes (gracias, Bruce!).

```

Tiempo Sistema
02:26 Dual Pentium Pro 180, 96MB, UW-SCSI, Linux 2.0.30, gcc 2.7.2.1 -
O2 -m486

```

v6.3

Los tiempos para esta version no son directamente comparables a aquellos de versiones precedentes puesto que algunos test de regresion adicionales han sido incluidos y algunos test obsoletos incluyendo los tiempos de viaje han sido eliminados. No obstante, en general, la v6.3 es sustancialmente mas rapida que versiones precedentes (gracias, Bruce!).

```

Tiempo Sistema
02:30 Dual Pentium Pro 180, 96MB, UW-SCSI, Linux 2.0.30, gcc 2.7.2.1 -
O2 -m486
04:12 Dual Pentium Pro 180, 96MB, EIDE, Linux 2.0.30, gcc 2.7.2.1 -
O2 -m486

```

v6.1

```

Tiempo Sistema
06:12 Pentium Pro 180, 32MB, EIDE, Linux 2.0.30, gcc 2.7.2 -O2 -m486
12:06 P-100, 48MB, Linux 2.0.29, gcc
39:58 Sparc IPC 32MB, Solaris 2.5, gcc 2.7.2.1 -O -g

```


Capítulo 31. Arquitectura

Conceptos de Arquitectura de Postgres

Antes de continuar, debería usted conocer la arquitectura básica del sistema Postgres. El conocimiento de como interactúan las partes de Postgres debería aclararse algo durante el siguiente capítulo. En la jerga de las bases de datos, Postgres utiliza un simple modelo cliente/servidor de "proceso por usuario". Una sesión de Postgres consiste en los siguientes procesos Unix (programas) cooperando:

- Un proceso demonio supervisor (`postmaster`),
- la aplicación de interface del usuario (frontend en inglés) (por ejemplo, el programa `psql`), y
- los uno o más procesos servidores de acceso a la base de datos (backend en inglés) (el proceso `postgres` mismo).

Un único `postmaster` maneja una colección dada de bases de datos en único host. Tal colección se denomina una instalación o un site. Las aplicaciones de frontend que quieren acceder a una base de datos dada en una instalación realizan llamadas a la librería. La librería envía el requerimiento del usuario a través de la red al `postmaster` (*Cómo se establece una conexión(a)*), quien en su turno arranca un nuevo proceso servidor de backend (*Cómo se establece una conexión(b)*)

Figura 31-1. Cómo se establece una conexión

y se conecta el proceso cliente al nuevo servidor (*Cómo se establece una conexión(c)*). > A partir de aquí, el proceso cliente y el servidor se comunican entre ellos sin intervención del `postmaster`. En consecuencia, el proceso `postmaster` está siempre corriendo, esperando llamadas, mientras que los procesos cliente y servidor vienen y van. La librería `libpq` permite a un único proceso cliente tener múltiples conexiones con procesos servidores. Sin embargo, la aplicación cliente sigue siendo un proceso mono-hebra. Las conexiones con multihebrado cliente/servidor no están actualmente soportadas en `libpq`. Una implicación de esta arquitectura es que el `postmaster` y los servidores siempre corren en la misma máquina (el servidor de base de datos), mientras que el cliente puede correr en cualquier sitio. Debe usted tener esto en cuenta, ya que los ficheros que pueden estar accesibles en una máquina cliente, pueden no estarlo (o estarlo sólo con un nombre de fichero diferente) en la máquina servidor. Debería tener también en cuenta que `postmaster` y los servidores `postgres` corren bajo el user-id del "superusuario" de Postgres. Nótese que el superusuario de Postgres no tiene porqué ser un usuario especial (es decir, un usuario llamado "postgres"), aunque en muchos sistemas esté instalado así. Más aún, el superusuario de Postgres

definitivamente ¡no debe de ser el superusuario de Unix, "root"! En cualquier caso, todos los ficheros relacionados con una base de datos deben encontrarse bajo este superusuario de Postgres.

Capítulo 32. Extensor SQL: Preludio

En la seccion que sigue, trataremos como añadir extensiones al Postgres SQL usando peticiones del lenguaje:

- funciones
- tipos
- operadores
- añadidos

Como hacer extensible el trabajo

Postgres es extensible porque las operaciones son catalogos en disco. Si está familiarizado con los estandares de sistemas relacionales, sabe que la informacion se almacena en bases de datos, tablas, columnas, etc., en lo que se comunmente conoce como sistema de catalogos. (Algunos sistemas lo llaman diccionario de datos). El catalogo aparece al usuario como una clase, como otro objeto cualquiera, pero DBMS lo almacena en una biblioteca. Una diferencia clave entre Postgres y el estandar de sistemas relacionales es que Postgres almacena muchas mas informacion en su catalogos – no solo informacion de tablas y columnas, sino tambien informacion sobre sus tipos, funciones y metodos de acceso. Estas clases pueden ser modificadas por el usuario, y dado que Postgres basa la operacion interna en todas sus clases, esto significa que Postgres puede ser extendido por los usuarios. Por comparacion, la convencion es que los sistemas de base de datos pueden ser extendidos solamente cambiando los procedimientos codificados del DBMS o cargando modulos especialmente escritos por el vendedor de DBMS.

Postgres es tambien distinto a otros gestores de datos en que el servidor puede incorporar codigo escrito por el usuario a traves de bibliotecas de carga dinamicas. O sea, el usuario puede especificar un fichero de codigo objeto (p. ej., un fichero compilado .o o bibliotecas de intercambio) con lo que se implementa un nuevo tipo o funciones y Postgres cargara lo que requiera. El codigo escrito en SQL es mas dificil de añadir al servidor. Esta habilidad para modificar la operacion 'al vuelo' hace de Postgres la unica suite para prototipos rapidos de nuevas aplicaciones y estructuras de almacenamiento.

El Tipo de Sistema de Postgres

El tipo de sistema de Postgres puede entrar en crisis por varios medios. Los tipos estan divididos en tipos base y tipos compuestos. Los tipos base son precisamente eso, como *int4*, que es implementado en leguajes como C. Generalmente se corresponde a lo comunmente conocido como "abstract data types"; Postgres puede operar solo con los tipos de metodos provistos por el usuario y solo se entiende el comportamiento de los tipos de la extension que el usuario describe. Los tipos compuestos son los creados cuando el usuario crea una clase. EMP es un ejemplo de un tipo de composicion.

Postgres almacena estos tipos en solo un sentido (que el fichero que almacena todas las instancias de las clases) pero el usuario puede "mirar dentro" de estos tipos desde el lenguaje de peticion y optimizar sus recuperacion por (por ejemplo) definir indices en los atributos. La base de tipos de Postgres esta mas dividida en tipos y tipos definidos por el usuario. Los tipos de construccion (como *int4*) son los que son compilados

dentro del sistema. Los tipos definidos por el usuario son creados por el usuario de la manera descrita abajo.

Acerca de los Sistema de Catalogo de Postgres

Para introducirnos en los conceptos basicos de la extensibilidad, hemos de estudiar como se diseñan los catalogos. Puede saltarse esta seccion ahora, pero algunas secciones mas tarde no seran comprendidas sin la informacion dada aqui, asi que marque esta página como posterior referencia. Todos los sistemas de catalogos tienen un nombre que empieza por *pg_*. Las siguientes clases contienen informacion que debe de ser util para los usuarios finales. (Hay muchas otros sistemas de catalogo, pero estos raramente son pedidos directamente.)

Tabla 32-1. Sistema de Catalogos de Postgres

Nombre del Catalogo	Descripcion
pg_database	base de datos
pg_class	clases
pg_attribute	atributos de clases
pg_index	indices secundarios
pg_proc	procedimientos (ambos C y SQL)
pg_type	tipos (ambos base y complejo)
pg_operator	operadores
pg_aggregate	conjunto y conjunto de funciones
pg_am	metodo de acceso
pg_amop	operador de metodo de acceso
pg_amproc	soporte de operador de metodo de acceso
pg_opclass	operador de clases de metodo de acceso

Figura 32-1. El principal sistema de catalogo de Postgres

El manual de referencia da mas detalle de explicacion de estos catalogos y sus atributos. De cualquier manera, *El principal sistema de catalogo de Postgres* muestra su mayor entidades y sus relacionamiento en el sistema de catalogo. (Los atributos que no se

refieren a otras entidades no son mostrados si no son parte primaria de la llave. Este diagrama puede ser mas o menos incomprensible hasta que realmente comience a mirar los contenidos de los catalogos y vea como se relacionan entre si. Por ahora, lo principal seguir este diagrama:

- En varias de las secciones que vienen a continuacion, presentaremos varias consultas compuestas en los catalogos del sistema que presentan informacion que necesitamos para extender el sistema. Mirado este diagrama podremos hacer que algunas de estas consultas compuestas (que a menudo se componen de tres o cuatro partes) sean Más comprensibles mas faciles de entender, dado que sera capaz de ver los atributos usados en las claves importadas de los formularios de consulta de otras clases.
- Muchas características distintas (clases, atributos, funciones, tipos, metodos de acceso, etc) estan estrechamente relacionadas en este esquema. Un simple comando de creacion puede modificar muchos de estos catalogos.
- Los tipos y procedimientos son elementos fundamentales de este esquema.

Nota: Usamos las palabras *procedure* y *function* de forma mas o menos indistinta.

Practicamente todo catalogo contiene alguna referencia a instancias en una o ambas clases. Por ejemplo, Postgres frecuentemente usa firmas de tipo (por ejemplo, de funciones y operadores) para identificar instancias unicas en otros catalogos.

- Hay muchos otros atributos y relaciones que tienen significados obvios, pero hay otros muchos (particularmente aquellos que tienen que ver con los metodos de acceso) que no los tienen. Las relaciones entre `pg_am`, `pg_amop`, `pg_amproc`, `pg_operator` y `pg_opclass` son particularmente dificiles de comprender, y seran descritas en profundidad (en la seccion sobre tipos de interfase y operadores de indice) antes de que estudiemos las extensiones basicas.

Capítulo 33. 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.

Postgres SQL proporciona tres tipos de funciones:

- funciones de lenguaje de consultas (funciones escritas en SQL)
- funciones de lenguaje procedural (funciones escritas en, por ejemplo, PLTCL o PLSQL)
- funciones de lenguaje de programación (funciones escritas en un lenguaje de programación compilado tales como C)

Cada clase de función puede tomar un tipo base, un tipo compuesto o alguna combinación como argumentos (parámetros). Además, cada clase de función puede devolver un tipo base o un tipo compuesto. Es más fácil definir funciones SQL, así que empezaremos con ellas. Los ejemplos en esta sección se puede encontrar también en `funcs.sql` y `funcs.c`.

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 `set of`, entonces un elemento arbitrario del resultado de la última consulta será devuelto.

El cuerpo de una función SQL que sigue a `AS` debería ser una lista de consultas separadas por caracteres espacio en blanco y entre paréntesis 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.

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:

```
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';
```

Funciones SQL sobre Tipos Base

La función SQL más simple posible no tiene argumentos y sencillamente devuelve un tipo base, tal como int4:

```
CREATE FUNCTION one() RETURNS int4
AS 'SELECT 1 as RESULT' LANGUAGE 'sql';
```

```
SELECT one() AS answer;
```

```
+-----+
|answer |
+-----+
| 1      |
+-----+
```

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 sobrescribió 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:

```
CREATE FUNCTION add_em(int4, int4) RETURNS int4
AS 'SELECT $1 + $2;' LANGUAGE 'sql';
```

```
SELECT add_em(1, 2) AS answer;
```

```
+-----+
|answer |
+-----+
| 3      |
+-----+
```

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:

```
CREATE FUNCTION double_salary(EMP) RETURNS int4
AS 'SELECT $1.salary * 2 AS salary;' LANGUAGE 'sql';

SELECT name, double_salary(EMP) AS dream
FROM EMP
WHERE EMP.cubicle ~= '(2,1)::point;
```

```
+---+-----+
|name | dream |
+---+-----+
|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 |
+--+
```

Funciones de Lenguaje Procedural

Los lenguajes procedurales no están contruidos dentro de Postgres. Se proporcionan como módulos cargables. Por favor diríjase a la documentación para el PL en cuestión para los detalles sobre la sintaxis y cómo la cláusula AS se interpreta por el manejador del PL.

Hay dos lenguajes procedurales disponibles con la distribución estándar de Postgres (PLTCL y PLSQL), y otros lenguajes se pueden definir. Diríjase a *Lenguajes Procedurales* para más información.

Funciones Internas

Las funciones internas son funciones escritas en C que han sido enlazadas estáticamente en el proceso backend de Postgres. La cláusula da el nombre en lenguaje C de la función, que no necesita ser el mismo que el nombre que se declara para el uso de SQL. (Por razones de compatibilidad con versiones anteriores, una cadena AS vacía se acepta con el significado de que el nombre de la función en lenguaje C es el mismo que el nombre en SQL.) Normalmente, todas las funciones internas presentes en el backend se declaran como funciones SQL durante la inicialización de la base de datos, pero un usuario podría usar **CREATE FUNCTION** para crear nombres de alias adicionales para una función interna.

Funciones de Lenguaje Compilado (C)

Las funciones escritas en C se pueden compilar en objetos que se pueden cargar de forma dinámica, y usar para implementar funciones SQL definidas por el usuario. La primera vez que la función definida por el usuario es llamada dentro del backend, el cargador dinámico carga el código objeto de la función en memoria, y enlaza la función con el ejecutable en ejecución de Postgres. La sintaxis SQL para **CREATE FUNCTION** enlaza la función SQL a la función en código C de una de dos formas. Si la función SQL tiene el mismo nombre que la función en código C se usa la primera forma. El argumento cadena en la cláusula AS es el nombre de camino (pathname) completo del fichero que contiene el objeto compilado que se puede cargar de forma dinámica. Si el nombre de la función C es diferente del nombre deseado de la función SQL, entonces se usa la segunda forma. En esta forma la cláusula AS toma dos argumentos cadena, el primero es el nombre del camino completo del fichero objeto que se puede cargar de forma dinámica, y el segundo es el símbolo de enlace que el cargador dinámico debería buscar. Este símbolo de enlace es solo el nombre de función en el código fuente C.

Nota: Después de que se use por primera vez, una función de usuario, dinámicamente cargada, se retiene en memoria, y futuras llamadas a la función solo incurren en la pequeña sobrecarga de una búsqueda de tabla de símbolos.

La cadena que especifica el fichero objeto (la cadena en la cláusula AS) debería ser el *camino completo* del fichero de código objeto para la función, unido por comillas simples. Si un símbolo de enlace se usa en la cláusula AS, el símbolo de enlace se debería unir por comillas simples también, y debería ser exactamente el mismo que el nombre de la función en el código fuente C. En sistemas Unix la orden **nm** imprimirá todos los símbolos de enlace de un objeto que se puede cargar de forma dinámica. (Postgres no compilará una función automáticamente; se debe compilar antes de que se use en una orden **CREATE FUNCTION**. Ver abajo para información adicional.)

Funciones de Lenguaje C sobre Tipos Base

La tabla siguiente da el tipo C requerido para los parámetros en las funciones C que se cargarán en Postgres. La columna "Defined In" da el fichero de cabecera real (en el directorio `.../src/backend/`) en el que el tipo C equivalente se define. Sin embargo, si incluye `utils/builtins.h`, estos ficheros se incluirán de forma automática.

Tabla 33-1. Tipos de C equivalentes para los tipos internos de Postgres

Built-In Type	C Type	Defined In
abstime	AbsoluteTime	utils/nabstime.h
bool	bool	include/c.h
box	(BOX *)	utils/geo-decls.h
bytea	(bytea *)	include/postgres.h
char	char	N/A
cid	CID	include/postgres.h
datetime	(DateTime *)	include/c.h or include/postgres.h
int2	int2	include/postgres.h
int2vector	(int2vector *)	include/postgres.h
int4	int4	include/postgres.h
float4	float32 or (float4 *)	include/c.h or include/postgres.h
float8	float64 or (float8 *)	include/c.h or include/postgres.h
lseg	(LSEG *)	include/geo-decls.h
name	(Name)	include/postgres.h
oid	oid	include/postgres.h
oidvector	(oidvector *)	include/postgres.h
path	(PATH *)	utils/geo-decls.h
point	(POINT *)	utils/geo-decls.h
regproc	regproc or REGPROC	include/postgres.h
reltime	RelativeTime	utils/nabstime.h
text	(text *)	include/postgres.h
tid	ItemPointer	storage/itemptr.h
timespan	(TimeSpan *)	include/c.h or include/postgres.h
tinterval	TimeInterval	utils/nabstime.h
uint2	uint16	include/c.h
uint4	uint32	include/c.h
xid	(XID *)	include/postgres.h

Internamente, Postgres considera un tipo base como un "blob de memoria". Las fun-

ciones definidas por el usuario que usted define sobre un tipo en turn definen la forma en que Postgres puede operar sobre él. Esto es, Postgres solo almacenará y recuperará los datos desde disco y solo usará sus funciones definidas por el usuario para introducir y procesar los datos, así como para obtener la salida de los datos. Los tipos base pueden tener uno de los tres formatos internos siguientes:

- pass by value, fixed-length
- pass by reference, fixed-length
- pass by reference, variable-length

Los tipos por valor solo pueden tener 1, 2 o 4 bytes de longitud (incluso si su computadora soporta tipos por valor de otros tamaños). Postgres mismo solo pasa los tipos entero por valor. Debería tener cuidado al definir sus tipos para que tengan el mismo tamaño (en bytes) en todas las arquitecturas. Por ejemplo, el tipo `long` es peligroso porque es de 4 bytes en algunas máquinas y de 8 bytes en otras, mientras que el tipo `int` es de 4 bytes en la mayoría de las máquinas Unix (aunque no en la mayoría de computadores personales). Una implementación razonable del tipo `int4` en las máquinas Unix podría ser:

```
/* 4-byte integer, passed by value */
typedef int int4;
```

En el otro lado, los tipos de longitud fija de cualquier tamaño se pueden pasar por referencia. Por ejemplo, aquí se presenta una implementación de ejemplo de un tipo de Postgres:

```
/* 16-byte structure, passed by reference */
typedef struct
{
    double x, y;
} Point;
```

Solo los punteros a tales tipos se pueden usar a la hora de pasarlos como argumentos de entrada o de retorno en las funciones de Postgres. Finalmente, todos los tipos de longitud variable se deben pasar también por referencia. Todos los tipos de longitud variable deben comenzar con un campo `length` de exactamente 4 bytes, y todos los datos que se tengan que almacenar dentro de ese tipo deben estar situados en la memoria inmediatamente a continuación de ese campo `length`. El campo `length` es la longitud total de la estructura (es decir, incluye el tamaño del campo `length` mismo). Podemos definir el tipo `texto` como sigue:

```
typedef struct {
    int4 length;
    char data[1];
} text;
```

Obviamente, el campo `data` no es suficientemente largo para almacenar todas las cadenas posibles; es imposible declarar tal estructura en C. Al manipular tipos de longitud variable, debemos tener cuidado de reservar la cantidad de memoria correcta

y de inicializar el campo `length`. Por ejemplo, si quisiéramos almacenar 40 bytes en una estructura `text`, podríamos usar un fragmento de código como este:

```
#include "postgres.h"
...
char buffer[40]; /* our source data */
...
text *destination = (text *) malloc(VARHDRSZ + 40);
destination->length = VARHDRSZ + 40;
memmove(destination->data, buffer, 40);
...
```

Ahora que hemos visto todas las estructuras posibles para los tipos base, podemos mostrar algunos ejemplos de funciones reales. Suponga que `funcs.c` es así:

```
#include <string.h>
#include "postgres.h"

/* By Value */

int
add_one(int arg)
{
    return(arg + 1);
}

/* By Reference, Fixed Length */

Point *
makepoint(Point *pointx, Point *pointy )
{
    Point      *new_point = (Point *) malloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    return new_point;
}

/* By Reference, Variable Length */

text *
copytext(text *t)
{
    /*
     * VARSIZE is the total size of the struct in bytes.
     */
    text *new_t = (text *) malloc(VARSIZE(t));
    memset(new_t, 0, VARSIZE(t));
    VARSIZE(new_t) = VARSIZE(t);
    /*
     * VARDATA is a pointer to the data region of the struct.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),    /* source */
           VARSIZE(t)-VARHDRSZ);   /* how many bytes */
    return(new_t);
}

text *
```

```

concat_text(text *arg1, text *arg2)
{
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    memset((void *) new_text, 0, new_text_size);
    VARSIZE(new_text) = new_text_size;
    strncpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1)-
VARHDRSZ);
    strncat(VARDATA(new_text), VARDATA(arg2), VARSIZE(arg2)-
VARHDRSZ);
    return (new_text);
}

```

On OSF/1 we would type:

```

CREATE FUNCTION add_one(int4) RETURNS int4
AS 'PGROOT/tutorial/funcs.so' LANGUAGE 'c';

CREATE FUNCTION makepoint(point, point) RETURNS point
AS 'PGROOT/tutorial/funcs.so' LANGUAGE 'c';

CREATE FUNCTION concat_text(text, text) RETURNS text
AS 'PGROOT/tutorial/funcs.so' LANGUAGE 'c';

CREATE FUNCTION copytext(text) RETURNS text
AS 'PGROOT/tutorial/funcs.so' LANGUAGE 'c';

```

En otros sistemas, podríamos tener que especificar la extensión del nombre del fichero como .sl (para indicar que es una librería (o biblioteca) compartida).

Funciones del Lenguaje C sobre Tipos Compuestos

Los tipos compuestos no tienen un formato fijo como las estructuras de C. Las instancias de un tipo compuesto pueden contener campos null. Además, los tipos compuestos que son parte de una jerarquía de herencia pueden tener campos diferentes respecto a otros miembros de la misma jerarquía de herencia. Por ello, Postgres proporciona una interfaz procedural para acceder a los campos de los tipos compuestos desde C. Cuando Postgres procesa un conjunto de instancias, cada instancia se pasará a su función como una estructura opaca de tipo `TUPLE`. Suponga que queremos escribir una función para responder a la consulta

```

* SELECT name, c_overpaid(EMP, 1500) AS overpaid
FROM EMP
WHERE name = 'Bill' or name = 'Sam';

```

En la consulta anterior, podemos definir `c_overpaid` como:

```

#include "postgres.h"
#include "executor/executor.h" /* for GetAttributeByName() */

bool
c_overpaid(TupleTableSlot *t, /* the current instance of EMP */

```

```

        int4 limit)
    {
        bool isnull = false;
        int4 salary;
        salary = (int4) GetAttributeByName(t, "salary", &isnull);
        if (isnull)
            return (false);
        return(salary > limit);
    }

```

GetAttributeByName es la función de sistema de Postgres que devuelve los atributos fuera de la instancia actual. Tiene tres argumentos: el argumento de tipo TUPLE pasado a la función, el nombre del atributo deseado, y un parámetro de retorno que describe si el atributo es null. GetAttributeByName alineará los datos apropiadamente de forma que usted pueda convertir su valor de retorno al tipo deseado. Por ejemplo, si tiene un atributo name que es del tipo name, la llamada a GetAttributeByName sería así:

```

char *str;
...
str = (char *) GetAttributeByName(t, "name", &isnull)

```

La consulta siguiente permite que Postgres conozca a la función c_overpaid:

```

* CREATE FUNCTION c_overpaid(EMP, int4) RETURNS bool
  AS 'PGROOT/tutorial/obj/funcs.so' LANGUAGE 'c';

```

Aunque hay formas de construir nuevas instancias o de modificar las instancias existentes desde dentro de una función C, éstas son demasiado complejas para discutir las en este manual.

Escribiendo código

Ahora volvemos a la tarea más difícil de escribir funciones del lenguaje de programación. Aviso: esta sección del manual no le hará un programador. Debe tener un gran conocimiento de C (incluyendo el uso de punteros y el administrador de memoria malloc) antes de intentar escribir funciones C para usarlas con Postgres. Aunque sería posible cargar funciones escritas en lenguajes distintos a C en Postgres, eso es a menudo difícil (cuando es posible hacerlo completamente) porque otros lenguajes, tales como FORTRAN y Pascal a menudo no siguen la misma *convención de llamada* que C. Esto es, otros lenguajes no pasan argumentos y devuelven valores entre funciones de la misma forma. Por esta razón, asumiremos que las funciones de su lenguaje de programación están escritas en C.

Las funciones C con tipos base como argumentos se pueden escribir de una forma sencilla. Los equivalentes C de los tipos internos de Postgres son accesibles en un fichero C si *PGROOT/src/backend/utils/builtins.h* se incluye como un fichero de cabecera. Esto se puede conseguir escribiendo

```
#include <utils/builtins.h>
```

al principio del fichero fuente C.

Las reglas básicas para construir funciones C son las siguientes:

- La mayoría de los ficheros cabecera (include) para Postgres deberían estar ya instalados en `PGROOT/include` (ver Figura 2). Debería incluir siempre
`-I$PGROOT/include`
 en sus líneas de llamada a cc. A veces, podría encontrar que necesita ficheros cabecera que están en el código fuente del servidor mismo (es decir, necesita un fichero que no hemos instalado en include). En esos casos puede necesitar añadir uno o más de
`-I$PGROOT/src/backend`
`-I$PGROOT/src/backend/include`
`-I$PGROOT/src/backend/port/<PORTNAME>`
`-I$PGROOT/src/backend/obj`
 (donde `<PORTNAME>` es el nombre del puerto, por ejemplo, `alpha` or `sparc`).
- Al reservar memoria, use las rutinas de Postgres `palloc` y `pfree` en vez de las rutinas de la librería de C correspondientes `malloc` y `free`. La memoria reservada por `palloc` se liberará automáticamente al final de cada transacción, previniendo fallos de memoria.
- Siempre céntrese en los bytes de sus estructuras usando `memset` o `bzero`. Varias rutinas (tales como el método de acceso `hash`, `hash join` y el algoritmo `sort`) computan funciones de los bits puros contenidos en su estructura. Incluso si usted inicializa todos los campos de su estructura, puede haber varios bytes de relleno de alineación (agujeros en la estructura) que pueden contener valores incorrectos o basura.
- La mayoría de los tipos internos de Postgres se declaran en `postgres.h`, por eso es una buena idea incluir siempre ese fichero también. Incluyendo `postgres.h` incluirá también `elog.h` y `palloc.h` por usted.
- Compilar y cargar su código objeto para que se pueda cargar dinámicamente en Postgres siempre requiere flags (o banderas) especiales. Ver *Enlazando funciones de carga dinámica* para una explicación detallada de cómo hacerlo para su sistema operativo concreto.

Sobrecarga de funciones

Se puede definir más de una función con el mismo nombre, siempre que los argumentos que tomen sean diferentes. En otras palabras, los nombres de las funciones se pueden *sobrecargar*. Una función puede tener además el mismo nombre que un atributo. En el caso de que haya ambigüedad entre una función sobre un tipo complejo y un atributo del tipo complejo, se usará siempre el atributo.

Conflictos en el Espacio de Nombres

A partir de Postgres v6.6, la forma alternativa de la cláusula `AS` para la orden de SQL **CREATE FUNCTION** desempareja el nombre de la función SQL del nombre de función en el código fuente C. Esta es ahora la técnica preferida para realizar la sobrecarga de funciones.

Pre-v6.6

Para funciones escritas en C, el nombre SQL declarado en **CREATE FUNCTION** debe ser exactamente el mismo que el nombre real de la función en el código C (debido a esto debe ser un nombre de función de C legal).

Hay una sutil consecuencia de esta restricción: mientras las rutinas de carga dinámicas en la mayoría de los sistemas operativos están más que felices de permitirle cargar cualquier número de librerías compartidas que contienen nombres de funciones conflictivos (con idénticos nombres), pueden, de hecho, chapucear la carga de formas interesantes. Por ejemplo, si usted define una función dinámicamente cargada que resulta tener el mismo nombre que una función perteneciente a Postgres, el cargador DEC OSF/1 dinámico hace que Postgres llame a la función dentro de él mismo preferiblemente a dejar que Postgres llame a su función. Por esto, si quiere que su función se use en diferentes arquitecturas, recomendamos que no sobrecargue los nombres de las funciones C.

Hay un truco ingenioso para resolver el problema que se acaba de describir. Dado que no hay problemas al sobrecargar funciones SQL, usted puede definir un conjunto de funciones C con nombres diferentes y entonces definir un conjunto de funciones SQL con idénticos nombres que tomen los tipos de argumentos apropiados y llamen a la función C correspondiente.

Otra solución es no usar la carga dinámica, sino enlazar sus funciones al backend estáticamente y declararlas como funciones **INTERNAL**. Entonces, las funciones deben tener todos los nombres C distintos pero se pueden declarar con los mismos nombres SQL (siempre que los tipos de sus argumentos difieran, por supuesto). Esta forma evita la sobrecarga de una función wrapper (o envoltente) SQL, con la desventaja de un mayor esfuerzo para preparar un ejecutable del backend a medida. (Esta opción está disponible sólo en la versión 6.5 y posteriores, dado que las versiones anteriores requerían funciones internas para tener el mismo nombre en SQL que en el código C.)

Capítulo 34. Extendiendo SQL: Tipos

Como se mencionó anteriormente, hay dos clases de tipos en Postgres: tipos base (definidos en un lenguaje de programación) y tipos compuestos (instancias). Los ejemplos en esta sección hasta los de índices de interfaz se pueden encontrar en `complex.sql` y `complex.c`. Los ejemplos compuestos están en `funcs.sql`.

Tipos Definidos por el Usuario

Funciones Necesarias para un Tipo Definido por el Usuario

Un tipo definido por el usuario debe tener siempre funciones de entrada y salida. Estas funciones determinan cómo aparece el tipo en las cadenas (para la entrada por el usuario y la salida para el usuario) y cómo se organiza el tipo en memoria. La función de entrada toma una cadena de caracteres delimitada por null como su entrada y devuelve la representación interna (en memoria) del tipo. La función de salida toma la representación interna del tipo y devuelve una cadena de caracteres delimitada por null. Suponga que queremos definir un tipo complejo que representa números complejos. Naturalmente, elegimos representar un complejo en memoria como la siguiente estructura en C:

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

y una cadena de la forma (x, y) como la representación externa de la cadena. Estas funciones normalmente no son difíciles de escribir, especialmente la función de salida. Sin embargo, hay varios puntos a recordar:

- Al definir su representación externa (cadena), recuerde que al final debe escribir un parser completo y robusto para esa representación como su función de entrada!

```
Complex *
complex_in(char *str)
{
    double x, y;
    Complex *result;
    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2) {
        elog(WARN, "complex_in: error in parsing");
        return NULL;
    }
    result = (Complex *)palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    return (result);
}
```

La función de salida puede ser sencillamente:

```
char *
complex_out(Complex *complex)
{
    char *result;
    if (complex == NULL)
        return(NULL);
    result = (char *) palloc(60);
```

```

        sprintf(result, "(%g,%g)", complex->x, complex-
>y);
        return(result);
    }

```

- Debería intentar hacer las funciones de entrada y salida inversas la una a la otra. Si no lo hace, tendrá problemas serios cuando necesite volcar sus datos en un fichero y después leerlos (por ejemplo, en la base de datos de otra persona en otra computadora). Este es un problema particularmente común cuando hay números en punto flotante de por medio.

Para definir el tipo complejo, necesitamos crear las dos funciones definidas por el usuario `complex_in` y `complex_out` antes de crear el tipo:

```

CREATE FUNCTION complex_in(opaque)
RETURNS complex
AS 'PGROOT/tutorial/obj/complex.so'
LANGUAGE 'c';

CREATE FUNCTION complex_out(opaque)
RETURNS opaque
AS 'PGROOT/tutorial/obj/complex.so'
LANGUAGE 'c';

CREATE TYPE complex (
    internallength = 16,
    input = complex_in,
    output = complex_out
);

```

Como se discutió antes, Postgres soporta totalmente vectores (o arrays) de tipos base. Además, Postgres soporta vectores de tipos definidos por el usuario también. Cuando usted define un tipo, Postgres automáticamente proporciona soporte para vectores de ese tipo. Por razones históricas, el tipo vector tiene el mismo nombre que el tipo definido por el usuario con el carácter subrayado `_` antepuesto. Los tipos compuestos no necesitan ninguna función definida sobre ellos, dado que el sistema ya comprende cómo son por dentro.

Objetos Grandes

Los tipos discutidos hasta este punto son todos objetos "pequeños" – esto es, son menores que 8KB en tamaño.

Nota: 1024 longwords == 8192 bytes. De hecho, el tipo debe ser considerablemente menor que 8192 bytes, dado que las páginas y tuplas de sobrecarga de Postgres deben caber en esta limitación de 8KB también. El valor real que cabe depende de la arquitectura de la máquina.

Si usted necesita un tipo más grande para algo como un sistema de recuperación de documentos o para almacenar bitmaps, necesitará usar la interfaz de grandes objetos de Postgres.

Capítulo 35. Extendiendo SQL: Operadores

Postgres soporta operadores unitarios izquierdos, unitarios derechos y binarios. Los operadores pueden ser sobrecargados; eso es, el mismo nombre de operador puede ser usado para diferentes operadores que tengan diferentes número y tipo de argumentos. Si hay una situación ambigua y el sistema no puede determinar el operador correcto a usar, retornará un error. Tu puedes tener los tipos de operadores izquierdo y/o derecho para ayudar a entender que operadores tienen significado usar.

Cada operador es "azúcar sintáctico" por una llamada hacia una función subyacente que hace el trabajo real; entonces tu debes primero crear la función subyacente antes de que puedas crear el operador. Sin embargo, un operador *no* es sólo un azúcar sintáctico, porque carga información adicional que ayuda al planeador de consultas a optimizar consultas que usa el operador. Mucho de este capítulo, será dedicado a explicar esa información adicional. merely syntactic sugar, because it carries additional information

Este es un ejemplo de crear un operador para sumar dos números complejos. Nosotros aumimos que ya hemos creado la definición del tipo complejo. Primero necesitamos una función que haga el trabajo; entonces podemos crear el operador.

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS '$PWD/obj/complex.so'
    LANGUAGE 'c';

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_add,
    commutator = +
);
```

Ahora podemos hacer:

```
SELECT (a + b) AS c FROM test_complex;
```

```
+-----+
| c      |
+-----+
| (5.2,6.05) |
+-----+
| (133.42,144.95) |
+-----+
```

Nosotros hemos mostrado como crear un operador binario aquí. Para crear un operador unario, solamente omite un argumento izquierdo (para unario izquierdo) o un argumento derecho (para unario derecho). El procedimiento y las sentencias del argumento son los únicos items requeridos para CREATE OPERATOR (Crear operador). La sentencia COMMUTATOR (conmutador) mostrada en esta ejemplo es una indicación opcional para optimizar la consulta. Además de los detalles acerca de COMMUTATOR, aparecen otras indicaciones de optimización.

Información de optimización de operador

Autor: Escrito por Tom Lane.

Una definición de un operador Postgres puede incluir muchas sentencias opcionales que dicen al sistema cosas útiles acerca de como el operador se comporta. Estas sentencias deben ser proveídas siempre que sea apropiado, porque ellas pueden hacer considerablemente una más rápida ejecución de la consulta que usa el operador. Pero si tu las provees, debes estar seguro que están bien! Un incorrecto uso de una sentencia de optimización puede resultar en choques finales, salidas sutilmente erróneas, o otras cosas malas. Tu puedes no poner las sentencias de optimización si no estás seguro de ellas, la única consecuencia es que las consultas pueden ejecutarse más lentamente de lo necesario.

Sentencias de optimización adicionales pueden ser agregadas en versiones posteriores de Postgres. Las descritas aquí son todas aquellas que la versión 6.5 entinede.

COMMUTATOR (conmutador)

La sentencia COMMUTATOR ,si es proveída, nombra un operador que es el conmutador del operador que está siendo definido. Decimos que el operador A es el conmutador del operador B si $(x \text{ A } y)$ es igual $(y \text{ B } x)$ para todos los posibles valores de entrada x,y . Nota que B es también el conmutador de A. Por ejemplo, operadores ' $<$ ' and ' $>$ ' (y lógico) para un tipo de dato particular son usualmente entre sí conmutadores, y el operador '+' es usualmente conmutativo con sigo mismo. Pero el operador '-' no es conmutativo con nada.

El tipo de argumento izquierdo de un operador conmutado es el mismo que el tipo de argumento derecho de su conmutador, y viceversa. Entonces el nombre del operador conmutador es todo lo que Postgres necesita para ser dado de alta el conmutador y eso es todo lo que necesita ser proveído en la sentencia COMMUTATOR.

Cuando tu estas definiendo un operador conmutador por si mismo (self-conmutative), tu solamente hazlo. Cuando tu estas definiendo un par de operadores conmutadores, las cosas son un poquito mas engañosas: Cómo pede el primero ser definido refiriéndose a otro, que no ha sido definido todavía? hay dos soluciones a este problema:

- Un método es omitir la sentencia COMMUTATOR en el primer operador que tu defines, y entonces proveer uno en la segunda definición de operador. Desde que Postgres sabe que operadores conmutativos vienen de a pares, cuando ve la segunda definición automaticamente volverá y llenará en la sentencia COMMUTATOR faltante en la primera definición.
- La otra, una manera mas honesta es solamente incluir la sentencia COMMUTATOR en ambas definiciones. Cuando Postgresprocesa la primera definición y se da cuenta COMMUTATOR hace referencia a un opereador inexistenete, el sistema hará una entrada silenciosa para ese operador en la tabla (relación) pg_operator del sistema. Esta entrada silenciosa tendrá datos válidos solamente para el nombre del operador, tipos de argumentos izquierdo y derechos, y un tipo de resultado, debido queu es todo lo que Postgrespuede deducir en ese punto. La primera entrada la catálogo del operador enlazará hacia esa entrada silenciosa. Mas tarde, cuando tu definas el segundo operador, el sistema actualiza la entrada silenciosa con la información adicional de la segunda definición. Si tu tratas de usar el operador silenciosa antes de que sea llenado, tu solo obtendras un mensaje de error. (Nota:

Este procedimiento no funciona correctamente en versiones de Postgres anteriores a la 6.5, pero es ahora la manera recomendada de hacer las cosas.)

NEGATOR(negador)

La sentencia NEGATOR, si es proveída, nombra a un operador que es el negador del operador que está siendo definido. Nosotros decimos que un operador A es el negador de un operador B si ambos retornan resultados booleanos y $(x \text{ A } y)$ no es igual $(x \text{ B } y)$ para todas las posibles entradas x, y . Nota que B es también el negador de A. Por ejemplo, ' $<$ ' and ' $>=$ ' son un par de negadores para la mayoría de los tipos de datos.

A diferencia de COMMUTATOR, un par de operadores unarios, pueden ser válidamente marcados como negadores entre si; eso significaría $(A \text{ x})$ no es igual $(B \text{ x})$ para todo x , o el equivalente para operadores unarios derechos.

Un operador de negación debe tener el mismo tipo de argumento derecho y/o izquierdo como el operador en si mismo, entonces al igual que con COMMUTATOR, solo el nombre del operador necesita ser dado en la sentencia NEGATOR.

Proveer NEGATOR es de mucha ayuda para la optimización de las consultas desde que permite expresiones como NOT $(x=y)$ ser simplificadas en $x <> y$. Esto aparece mas seguido de los que tu debes pensar, porque NOTs pueden ser insertados como una consecuencia de otras reconstrucciones.

Pares de operadores negadores pueden ser definidos usando el mismo método explicado para pares de conmutadores.

RESTRICT (Restringir)

La sentencia RESTRICT, si es proveída, nombra una función de estimación selectiva de restricción para el operador (nota que esto es un nombre de función, no un nombre de un operador). Las sentencias RESTRICT solamente tienen sentido para operadores binarios que retornan valores booleanos. La idea detrás de un estimador selectivo de restricción es suponer que fracción de las filas en una tabla satisfacen una sentencia de condición WHERE en el formulario RESTRICT clauses only make sense for

```
field OP constant
```

para el operador corriente y un valor constante particular. Esto asiste al optimizador dándole alguna idea de como tantas filas van a ser eliminadas por la sentencia WHERE que tiene este formulario. (Qué pasa si la constante está a la izquierda, debes puedes estar preguntándote? bien, esa es una de las cosas para las que sirve COMMUTATOR...)

Escribiendo nuevas funciones de estimación selectivas de restricción está más allá del alcance de este capítulo, pero afortunadamente tu puedes usualmente solamente usar un estimador estandar del sistema para muchos de tus propios operadores. Estos son los estimadores estandars:

```
eqsel for =
neqsel for <>
scalartsel for < or <=
scalargtsel for > or >=
```

Puede parecer un poco curioso que estas son las categorías, pero ellas tienen sentido si tu piensas acerca de ellas. '=' típicamente aceptará solamente una fracción pequeña de las filas en la tabla; '<>' típicamente rechazará solamente una fracción pequeña. '<' aceptará una fracción que depende de donde las constantes dadas quedan en el rango de valores para esa columna de la tabla (la cual, solo ha pasado, es información recogida por el ANALIZADOR VACUUM y puesta disponible para el estimador selectivo). '<=' aceptará una fracción un poquito más laraga que '<' para las mismas constantes comparadas, pero son lo suficientemente cerradas para no ser costosas, especialmente desde que nosotros no estamos probablemente haciendo mejor que una aspera suposición de cualquier manera. Los mismos comentarios son aplicados a '>' y '>='.

Tu puedes frecuentemente escaparse de usar eqsel o neqsel para operadores que tienen muy alta o muy baja selectividad, incluso si ellas no son realmente equivalentes o no equivalentes. Por ejemplo, la expresión regular operadores emparejados (~, ~*, etc.) usa eqsel sobre la suposición que ellos usualmente solo emparejen una pequeña fracción de entradas en una tabla.

Tu puedes usar scalarltsel y scalargtsel para comparaciones sobre tipos de datos que tienen cierto significado sensible de ser convertido en escalares numéricos para comparaciones de rango. Es posible, añadir el tipo de dato a aquellos entendidos por la rutina `convert_to_scalar()` in `src/backend/utils/adt/selfuncs.c`. (Eventualmente, esta rutina debe ser reemplazada por funciones per-datatype identificadas a través de una columna de la tabla `pg_type`; pero eso no ha pasado todavía.) Si tu no haces esto, las cosas seguirán funcionando, pero la estimación del optimizador no estarán tan bien como podrían.

Hay funciones adicionales de selectividad diseñadas para operadores geométricos en `src/backend/adt/geo_selfuncs.c`: `areasel`, `positionsel`, y `contsel`. En este escrito estas son solo nombradas, pero tu puedes querer usarlas (o incluso mejorarlas).

JOIN (unir)

La sentencia JOIN, si es proveída, nombra una función de estimación selectiva join para el operador (nota que esto es un nombre de una función, no un nombre de un operador). Las sentencias JOIN solamente tienen sentido para operadores binarios que retorna valores booleanos. La idea detrás de un estimador selectivo join es suponer que fracción de las filas de un par de tablas satisfacen la condición de la sentencia WHERE del formulario

```
table1.field1 OP table2.field2
```

para el operador corriente. Como la sentencia RESTRICT, esta ayuda al optimizador muy sustancialmente permitiéndole deducir cual de las posibles secuencias join es probable que tome el menor trabajo.

como antes, este capítulo no procurará explicar como escribir una función estimadora selectiva join, pero solamente sugeriremos que tu uses uno de los estimadores estandar si alguna es aplicable:

```
eqjoinselect for =
neqjoinselect for <>
scalarltjoinselect for < or <=
scalargtjoinselect for > or >=
areajoinselect for 2D area-based comparisons
positionjoinselect for 2D position-based comparisons
contjoinselect for 2D containment-based comparisons
```

HASHES(desmenusamiento)

La sentencia HASHES, si está presente, le dice al sistema que está bien usar un método hash join para uno basado en join sobre este operador. HASHES solamente tiene sentido para operadores binarios que retornan valores binario, y en la práctica el operador tiene que estar igualado a algún tipo de datos.

La suposición subyacente de hash join es que el operador join puede solamente retornar TRUE (verdadero) para pares de valores izquierdos o derechos. Si dos valores puestos en diferentes recipientes hash, El join nunca los comparará a ellos del todo, implícitamente asumiendo que el resultado del operador join debe ser FALSE (falso). Entonces nunca tiene sentido especificar HASHES para operadores que no se representan igualmente.

De hecho, la igualdad lógica no es suficientemente buena; el operador tuvo mejor representación por igualdad pura bit a bit, porque la función hash será computada sobre la representación de la memoria de los valores sin tener en cuenta que significan los bits. Por ejemplo, igualdad de intervalos de tiempo no es igualdad bit a bit; el operador de igualdad de intervalo considera dos intervalos de tiempos iguales si ellos tienen la misma duración, si son o no son sus puntos finales idénticos. Lo que esto significa es que el uso de join "=" entre campos de intervalos produciría resultados diferentes si es implementado como un hash join o si es implementado en otro modo, porque una fracción larga de los pares que deberían igualar resultarán en valores diferentes y nunca serán comparados por hash join. Pero si el optimizador elige usar una clase diferente de join, todos los pares que el operador de igualdad diga que son iguales serán encontrados. No queremos ese tipo de inconsistencia, entonces no marcamos igualdad de intervalos como habilitados para hash.

Hay también modos de dependencia de máquina en cuales un hash join puede fallar en hacer las cosas bien. Por ejemplo, si tu tipo de dato es una estructura en la cual puede haber bloque de bits sin interés, es inseguro marcar el operador de igualdad HASHES. (al menos, quizás, tu escribes tu otro operador para asegurarte que los bits sin uso son iguales a zero). Otro ejemplo es que los tipos de datos de punto flotante son inseguros para hash joins. Sobre máquinas que cumplan los estándares de la IEEE de puntos flotantes, menos cero y más cero son dos valores diferentes (diferentes patrones de bit) pero ellos están definidos para compararlos igual. Entonces, si la igualdad de punto flotante estuviese marcada, hashes, un menos cero y un más cero probablemente no serían igualados por hash join, pero ellos serían igualados por cualquier otro proceso join.

La última línea es para la cual tu probablemente deberías usar únicamente HASEHES para igualdad de operadores que son (o podría ser) implementada por memcmp().

SORT1 and SORT2 (orden1 y orden2)

La sentencia SORT, si está presente, le dice al sistema que está permitido usar el método merge join (unir join) para un join basado sobre el operador corriente. Ambos deben ser especificados si tampoco está. El operador corriente debe ser igual para algunos pares de tipos de datos, y las sentencias SORT1 Y SORT2 nombran el operador de orden ('<' operador) para tipos de datos izquierdo y derecho respectivamente.

Merge join está basado sobre la idea de ordenar las tablas izquierdas y derechas en un orden y luego inspeccionarlas en paralelo. Entonces, ambos tipos de datos deben ser aptos de ser ordenados completamente, y el operador join debe ser uno que pueda solamente tener éxito con pares de valores que caigan en el mismo lugar en la busque-

da de orden. En práctica esto significa que el operador join debe comportarse como igualdad. Pero distinto de hashjoin, cuando los tipos de datos izquierdos y derechos tuvieron que ser mejor el mismo. (o al menos equivalentes bit a bit), es posible unir dos tipos de datos distintos tanto como sean ellos compatibles lógicamente. Por ejemplo, el operador de igualdad int2-versus-int4 es usable. Solo necesitamos operadores de orden que traigan ambos tipos de datos en una secuencia lógica compatible.

Cuando se especifican operadores operadores sort merge, el operador corriente y ambos operadores referenciados deben retornar valores booleanos; el operador SORT1 debe tener ambos tipos de datos de entrada iguales al tipo de argumento izquierdo del operador corriente y el operador SORT2 debe tener ambos tipos de datos de entrada iguales al tipo de argumento derecho del operador corriente. (como con COMMUTATOR y NEGATOR, esto significa que el nombre del operador es suficiente para especificar el operador, y el sistema es capaz de hacer entradas de operador silenciosas si tú definiste el operador de igualdad antes que los otros.

En práctica tú debes solo escribir sentencias SORT para un operador '=', y los dos operadores referenciados deben ser siempre nombrados '<'. Tratando de usar merge join con operadores nombrados nada más resultará en confusiones inesperadas, por razones que veremos en un momento.

Hay restricciones adicionales sobre operadores que tú marcas mergejoinables. Estas restricciones no son corrientemente chequeadas por CREATE OPERATE, pero un merge join puede fallar en tiempo de ejecución si alguna no es verdad:

- El operador de igualdad mergejoinable debe tener un conmutador (El mismo si los dos tipos de datos son iguales, o un operador de igualdad relativo si son diferentes.).
- Debe haber operadores de orden '<' and '>' teniendo los mismos tipos de datos izquierdo y derecho de entrada como el operador mergejoinable en sí mismo. Estos operadores *deben* ser nombrados '<' and '>'; tú no tienes opción en este problema, desde que no hay provisión para especificarlos explícitamente. Nota que si los tipos de datos izquierdo y derecho son diferentes, ninguno de estos operadores es el mismo que cualquier operador SORT. pero ellos tuvieron mejores ordenados la compatibilidad de los valores de dato con los operadores SORT, o o mergejoin fallará al funcionar.

Capítulo 36. Extensiones de SQL: Agregados

Los agregados en Postgres están expresados en términos de funciones de transición de estado. Es decir, un agregado puede estar definido en términos de un estado que es modificado cuando una instancia es procesada. Algunas funciones de estado miran un valor particular en la instancia cuando calculan el nuevo estado (sfunc1 en la sintaxis de create aggregate) mientras que otras sólo se preocupan de su estado interno (sfunc2). Si definimos un agregado que utiliza solamente sfunc1, definimos un agregado que computa una función de los atributos de cada instancia. "Sum" es un ejemplo de este tipo de agregado. "Sum" comienza en cero y siempre añade el valor de la instancia actual a su total. Utilizaremos int4pl que está integrado en Postgres para realizar esta adición.

```
CREATE AGGREGATE complex_sum (
    sfunc1 = complex_add,
    basetype = complex,
    stype1 = complex,
    initcond1 = '(0,0)'
);

SELECT complex_sum(a) FROM test_complex;
```

```
+-----+
|complex_sum |
+-----+
| (34,53.9)  |
+-----+
```

Si solamente definimos sfunc2, estamos especificando un agregado que computa una función que es independiente de los atributos de cada instancia. "Count" es el ejemplo más común de este tipo de agregado. "Count" comienza a cero y añade uno a su total para cada instancia, ignorando el valor de instancia. Aquí, utilizamos la rutina integrada int4inc para hacer el trabajo por nosotros. Esta rutina incrementa (añade uno) su argumento.

```
CREATE AGGREGATE my_count (
    sfunc2 = int4inc, - add one
    basetype = int4,
    stype2 = int4,
    initcond2 = '0'
);

SELECT my_count(*) as emp_count from EMP;
```

```
+-----+
|emp_count |
+-----+
| 5         |
+-----+
```

"Average" es un ejemplo de un agregado que requiere tanto una función para calcular la suma actual y una función para calcular el contador actual. Cuando todas las instancias han sido procesadas, la respuesta final para el agregado es la suma actual dividida por el contador actual. Utilizamos las rutinas int4pl y int4inc que utilizamos anteriormente así como también la rutina de división entera de Postgres , int4div, para calcular la división de la suma por el contador.

```

CREATE AGGREGATE my_average (
    sfunc1 = int4pl,      - sum
    basetype = int4,
    stype1 = int4,
    sfunc2 = int4inc,    - count
    stype2 = int4,
    finalfunc = int4div, - division
    initcond1 = '0',
    initcond2 = '0'
);

SELECT my_average(salary) as emp_average FROM EMP;

```

```

+-----+
|emp_average |
+-----+
|1640      |
+-----+

```


Capítulo 37. El Sistema de reglas de Postgres

Los sistemas de reglas de producción son conceptualmente simples, pero hay muchos puntos sutiles implicados en el uso actual de ellos. Algunos de estos puntos y los fundamentos teóricos del sistema de reglas de Postgres se pueden encontrar en [Stonebraker et al, ACM, 1990].

Algunos otros sistemas de base de datos definen reglas de base de datos activas. Éstas son habitualmente procedimientos y disparadores (a partir de aquí utilizaré el término más habitual de "trigger") almacenados y se implementan en Postgres como funciones y triggers.

El sistema de reglas de reescritura de queries (el "sistema de reglas" a partir de ahora) es totalmente diferente a los procedimientos almacenados y los triggers. Él modifica las queries para tomar en consideración las reglas y entonces pasa la query modificada al optimizador para su ejecución. Es muy poderoso, y puede utilizarse de muchas formas, tales como procedimientos, vistas y versiones del lenguaje de query. El poder de este sistema de reglas se discute en [Ong and Goh, 1990] y en [Stonebraker et al, ACM, 1990].

¿Qué es un árbol de query?

Para comprender como trabaja el sistema de reglas, es necesario conocer cuándo se invoca y cuáles son sus inputs y sus resultados.

El sistema de reglas se sitúa entre el traductor de la query y el optimizador. Toma la salida del traductor, un árbol de la query, y las reglas de reescritura del catálogo `pg_rewrite`, los cuales son también árboles de queries con alguna información extra, y crea cero o muchos árboles de query como resultado. De este modo, su input y su output son siempre tales como el traductor mismo podría haberlos producido y, de este modo, todo aparece básicamente representable como una instrucción SQL.

Ahora, ¿qué es un árbol de query? Es una representación interna de una instrucción SQL donde se almacenan de modo separado las partes menores que la componen. Estos árboles de query son visibles cuando arrancamos el motor de Postgres con nivel de debug 4 y tecleamos queries en el interface de usuario interactivo. Las acciones de las reglas almacenadas en el catálogo de sistema `pg_rewrite` están almacenadas también como árboles de queries. No están formateadas como la salida del debug, pero contienen exactamente la misma información.

Leer un árbol de query requiere experiencia y era bastante duro cuando empecé a trabajar en el sistema de reglas. Puedo recordar que mientras estaba esperando en la máquina de café asimilaba el vaso a una lista de objetivos, el agua y el polvo del café a una tabla de rangos, y todos los botones a expresiones de cualificación. Puesto que las representaciones de SQL de árboles de queries son suficientes para entender el sistema de reglas, este documento no le enseñará como leerlo. Él debería ayudarle a aprenderlo, con las convenciones de nombres requeridas en las descripciones que siguen más adelante.

Las partes de un árbol de query

Cuando se leen las representaciones de SQL de los árboles de queries en este documento, es necesario ser capaz de identificar las partes de la instrucción que se ha roto en ella, y que está en la estructura del árbol de query. Las partes de un árbol de query son:

El tipo de commando (commandtype)

Este es un valor sencillo que nos dice el comando que produjo el árbol de traducción (SELECT, INSERT, UPDATE, DELETE).

La tabla de rango (rangetable)

La tabla de rango es una lista de las relaciones que se utilizan en la query. En una instrucción SELECT, son las relaciones dadas tras la palabra clave FROM.

Toda entrada en la tabla del rango identifica una tabla o vista, y nos dice el nombre por el que se la identifica en las otras partes de la query. En un árbol de query, las entradas de la tabla de rango se indican por un índice en lugar de por su nombre como estarían en una instrucción SQL. Esto puede ocurrir cuando se han mezclado las tablas de rangos de reglas. Los ejemplos de este documento no muestran esa situación.

La relación-resultado (resultrelation).

Un índice a la tabla de rango que identifica la relación donde irán los resultados de la query.

Las queries SELECT normalmente no tienen una relación resultado. El caso especial de una SELECT INTO es principalmente idéntica a una secuencia CREATE TABLE, INSERT ... SELECT y no se discute aquí por separado.

En las queries INSERT, UPDATE y DELETE, la relación resultado es la tabla (¡o vista!) donde tendrán efecto los cambios.

La lista objetivo (targetlist).

La lista objetivo es una lista de expresiones que definen el resultado de la query. En el caso de una SELECT, las expresiones son las que construyen la salida final de la query. Son las expresiones entre las palabras clave SELECT y FROM (* es sólo una abreviatura de todos los nombres de atributos de una relación).

Las queries DELETE no necesitan una lista objetivo porque no producen ningún resultado. De hecho, el optimizador añadirá una entrada especial para una lista objetivo vacía. Pero esto ocurre tras el sistema de reglas y lo comentaremos más tarde. Para el sistema de reglas, la lista objetivo está vacía.

En queries INSERT la lista objetivo describe las nuevas filas que irán a la relación resultado. Las columnas que no aparecen en la relación resultado serán añadidas por el optimizador con una expresión constante NULL. Son las expresiones de la cláusula VALUES y las de la cláusula SELECT en una INSERT SELECT.

En queries UPDATE, describe las nuevas filas que reemplazarán a otras viejas. Ahora el optimizador añadirá las columnas que no aparecen insertando expresiones que recuperan los valores de las filas viejas en las nuevas. Y añadirá una entrada especial como lo hace DELETE. Es la parte de la query que recoge las expresiones del atributo SET atributo = expresión.

Cada entrada de la lista objetivo contiene una expresión que puede ser un valor constante, una variable apuntando a un atributo de una de las relaciones en la tabla de rango, un parámetro o un árbol de expresiones hecho de llamadas a funciones, constantes, variables, operadores, etc.

La cualificación.

La cualificación de las queries es una expresión muy similar a otra de las contenidas en las entradas de la lista objetivo. El valor resultado de esta expresión es un booleano que dice si la operación (INSERT, UPDATE, DELETE o SELECT) para las filas del resultado final deberá ser ejecutada o no. Es la cláusula WHERE de una instrucción SQL.

the others

Las otras partes de un árbol de query, como la cláusula ORDER BY, no tienen interés aquí. El sistema de reglas sustituye las entradas aquí presentes mientras está aplicando las reglas, pero aquellas no tienen mucho que hacer con los fundamentos del sistema de reglas. GROUP BY es una forma especial en la que aparece una definición de una vista, y aún necesita ser documentado.

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 semánticas de las otras en las que modifican el árbol 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:

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS
'BEGIN
    IF $1 < $2 THEN
        RETURN $1;
    END IF;
    RETURN $2;
END;'
LANGUAGE 'plpgsql';
```

Las tablas reales que necesitaremos en las dos primeras descripciones del sistema de reglas son estas:

```
CREATE TABLE shoe_data (           - datos de zapatos
    shoename    char(10),           - clave primaria (primary key)
    sh_avail    integer,            - número de pares utilizables
    slcolor     char(10),           - color de cordón preferido
    slminlen    float,              - longitud mínima de cordón
    slmaxlen    float,              - longitud máxima del cordón
    slunit      char(8)             - unidad de longitud
);

CREATE TABLE shoelace_data (       - datos de cordones de zapatos
    sl_name     char(10),           - clave primaria (primary key)
    sl_avail    integer,            - número de pares utilizables
    sl_color    char(10),           - color del cordón
    sl_len      float,              - longitud del cordón
    sl_unit     char(8)             - unidad de longitud
);

CREATE TABLE unit (                - unidades de longitud
    un_name     char(8),            - clave primaria (primary key)
    un_fact     float               - factor de transformación a cm
);
```

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 fácil la vida ignorarlos.

Las vistas las crearemos como:

```
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:: 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 interés 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      |        |black     |80    |cm      |80
sl2      |        |black     |100   |cm      |100
sl7      |        |brown     |60    |cm      |60
sl3      |        |black     |35    |inch    |88.9
sl4      |        |black     |40    |inch    |101.6
sl8      |        |brown     |40    |inch    |101.6
sl5      |        |brown     |1     |m       |100
sl6      |        |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,
       s.sl_color, s.sl_len, s.sl_unit,
       float8mul(s.sl_len, u.un_fact) AS sl_len_cm
FROM shoelace *OLD*, shoelace *NEW*,
     shoelace_data s, unit u
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

```

SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
FROM shoelace shoelace, shoelace *OLD*,
     shoelace *NEW*,
     shoelace_data s,
     unit u;

```

En el paso 2, añade la cualificación de la acción de las reglas al árbol de traducción resultante en

```

SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,

```

```

        shoelace.sl_unit, shoelace.sl_len_cm
FROM shoelace shoelace, shoelace *OLD*,
     shoelace *NEW*, shoelace_data s,
     unit u
WHERE bpchareq(s.sl_unit, u.un_name);

```

Y en el paso 3, reemplaza todas las variables en el árbol 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:

```

SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len,
       s.sl_unit,
       float8mul(s.sl_len, u.un_fact) AS sl_len_cm
FROM shoelace shoelace, shoelace *OLD*,
     shoelace *NEW*, shoelace_data s,
     unit u
WHERE bpchareq(s.sl_unit, u.un_name);

```

Para realizar esta salida en una instrucción SQL real, un usuario humano debería teclear:

```

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;

```

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 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 ignora. 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 árbol 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 árbol 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
WHERE min(sh.sh_avail, s.sl_avail) >= 2
      AND s.sl_color = sh.slcolor
      AND s.sl_len * u.un_fact >= sh.slminlen * un.un_fact
      AND s.sl_len * u.un_fact <= sh.slmaxlen * un.un_fact
      AND sh.sl_unit = un.un_name
      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: 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 límite 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 árbol 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:

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

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 contienen 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:

```
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

```
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

```
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 transacción 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.

El poder de las vistas en Postgres

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).

Beneficios

Los beneficios de implementar las vistas con el sistema de reglas están en que el optimizador 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.

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 difícil 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 difícil 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.

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 razón 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.

Reglas sobre INSERT, UPDATE y DELETE

Diferencias con las reglas de las vistas.

Las reglas que se definen para ON INSERT, UPDATE y DELETE son totalmente diferentes de las que se han descrito en la sección anterior para las vistas. Primero, su comando CREATE RULE permite más:

- Pueden no tener acción.
- Pueden tener múltiples acciones.
- La palabra clave INSTEAD es opcional.
- Las pseudo-relaciones NEW y OLD se vuelven utilizables.
- Puede haber cualificaciones a las reglas.

Segundo, no modifican el árbol de traducción en el sitio. En lugar de ello, crean cero o varios árboles de traducción nuevos y pueden desechar el original.

Cómo trabajan estas reglas

Mantenga en mente la sintaxis

```
CREATE RULE rule_name AS ON event
    TO object [WHERE rule_qualification]
    DO [INSTEAD] [action | (actions) | NOTHING];
```

En lo que sigue, "las reglas de update" muestran reglas que están definidas ON INSERT, UPDATE o DELETE.

Update toma las reglas aplicadas por el sistema de reglas cuando la relación resultado y el tipo de comando de un árbol de traducción son iguales al objeto y el acontecimiento dado en el comando CREATE RULE. Para reglas de update, el sistema de reglas crea una lista de árboles de traducción. Inicialmente la lista de árboles de traducción está vacía. Puede haber cero (palabra clave NOTHING), una o múltiples acciones. Para simplificar, veremos una regla con una acción. Esta regla puede tener una cualificación o no y puede ser INSTEAD o no.

¿Qué es una cualificación de una regla? Es una restricción que se dice cuándo las acciones de una regla se deberían realizar y cuándo no. Esta cualificación sólo se puede referir a las pseudo-relaciones NEW y/o OLD, que básicamente son la relación dada como objeto (pero con unas características especiales).

De este modo tenemos cuatro casos que producen los siguientes árboles de traducción para una regla de una acción:

- Sin cualificación ni INSTEAD:
 - El árbol de traducción para la acción de la regla a la que se ha añadido cualificación a los árboles de traducción originales.
- Sin cualificación pero con INSTEAD:
 - El árbol de traducción para la acción de la regla a la que se ha añadido cualificación a los árboles de traducción originales.
- Se da cualificación y no se da INSTEAD:
 - El árbol de traducción de la acción de la regla, a la que se han añadido la cualificación de la regla y la cualificación de los árboles de traducción originales.
- Se da cualificación y se da INSTEAD:
 - El árbol de traducción de la acción de la regla a la que se han añadido la cualificación de la regla y la cualificación de los árboles de traducción originales.
 - El árbol de traducción original al que se le ha añadido la cualificación de la regla negada.

Finalmente, si la regla no es INSTEAD, el árbol de traducción original sin cambiar se añade a la lista. Puesto que sólo las reglas INSTEAD cualificadas se añaden al árbol de traducción original, terminamos con un máximo total de dos árboles de traducción para una regla con una acción.

Los árboles de traducción generados a partir de las acciones de las reglas se colocan en el sistema de reescritura de nuevo, y puede ser que otras reglas aplicadas resulten en más o menos árboles de traducción. De este modo, los árboles de traducción de las acciones de las reglas deberían tener bien otro tipo de comando, bien otra relación resultado. De otro modo, este proceso recursivo terminaría en un bucle. Hay un límite de recursiones compiladas actualmente de 10 iteraciones. Si tras 10 iteraciones aún sigue habiendo reglas de update para aplicar, el sistema de reglas asumirá que se ha producido un bucle entre muchas definiciones de reglas y aborta la transacción.

Los árboles de traducción encontrados en las acciones del catálogo de sistema `pg_rewrite` son sólo plantillas. Una vez que ellos pueden hacer referencia a las entradas de tabla de rango para NEW u OLD, algunas sustituciones habrán sido hechas antes de ser utilizadas. Para cualquier referencia a NEW, la lista objetivo de la query original se revisa buscando una entrada correspondiente. Si se encuentra, esas entradas de la expresión se sitúan en la referencia. De otro modo, NEW se mantiene igual que OLD. Cualquier referencia a OLD se reemplaza por una referencia a la entrada de la tabla de rango que es la relación resultado.

Una primera regla paso a paso.

Queremos trazar los cambios en la columna `sl_avail` de la relación `shoelace_data`. Para ello, crearemos una tabla de log, y una regla que escriba las entradas cada vez que se realice una UPDATE sobre `shoelace_data`.

```
CREATE TABLE shoelace_log (
    sl_name    char(10),      - shoelace changed
    sl_avail   integer,       - new available value
    log_who    name,          - who did it
    log_when   datetime       - when
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
WHERE NEW.sl_avail != OLD.sl_avail
DO INSERT INTO shoelace_log VALUES (
    NEW.sl_name,
    NEW.sl_avail,
    getpgusername(),
    'now'::text
);
```

Un detalle interesante es la caracterización de 'now' en la reglas de la acción INSERT para teclear texto. Sin ello, el traductor vería en el momento del CREATE RULE, que el tipo objetivo en `shoelace_log` es un dato de tipo fecha, e intenta hacer una constante de él... con éxito. De ese modo, se almacenaría un valor constante en la acción de la regla y todas las entradas del log tendrían la hora de la instrucción CREATE RULE. No es eso exactamente lo que queremos. La caracterización lleva al traductor a construir un "fecha-hora" que será evaluada en el momento de la ejecución (`datetime('now'::text)`).

Ahora Al hace

```
al_bundy=> UPDATE shoelace_data SET sl_avail = 6
al_bundy-> WHERE sl_name = 'sl7';
```

y nosotros miramos en la tabla de log.

```
al_bundy=> SELECT * FROM shoelace_log;
sl_name    |sl_avail|log_who|log_when
-----+-----+-----+-----
sl7         |        6|Al      |Tue Oct 20 16:14:45 1998 MET DST
```

```
(1 row)
```

Que es justo lo que nosotros esperábamos. Veamos qué ha ocurrido en la sombra. El traductor creó un árbol de traducción (esta vez la parte del árbol de traducción original está resaltado porque la base de las operación es la acción de la regla para las reglas de update)

```
UPDATE shoelace_data SET sl_avail = 6
FROM shoelace_data shoelace_data
WHERE bpchareq(shoelace_data.sl_name, 'sl7');
```

Hay una regla para 'log_shoelace' que es ON UPDATE con la expresión de cualificación de la regla:

```
int4ne(NEW.sl_avail, OLD.sl_avail)
```

y una acción

```
INSERT INTO shoelace_log SELECT
    *NEW*.sl_name, *NEW*.sl_avail,
    getpgusername(), datetime('now'::text)
FROM shoelace_data *NEW*, shoelace_data *OLD*,
    shoelace_log shoelace_log;
```

No detallaremos la salida de la vista del sistema pg_rules. Especialmente manipula la situación de que aquí sólo se haga referencia a NEW y OLD en la INSERT, y las salidas del formato de VALUES de INSERT. De hecho, no hay diferencia entre una INSERT ... VALUES y una INSERT ... SELECT al nivel del árbol de traducción. Ambos tienen tablas de rango, listas objetivo, pueden tener cualificación, etc. El optimizador decide más tarde si crear un plan de ejecución de tipo resultado, barrido secuencial, barrido de índice, join o cualquier otro para ese árbol de traducción. Si no hay referencias en entradas de la tabla de rango previas al árbol de traducción, éste se convierte en un plan de ejecución (la versión INSERT ... VALUES). La acción de las reglas anterior puede ciertamente resultar en ambas variantes.

La regla es una regla no-*INSTEAD* cualificada, de modo que el sistema de reglas deberá devolver dos árboles de traducción. La acción de la regla modificada y el árbol de traducción original. En el primer paso, la tabla de rango de la query original está incorporada al árbol de traducción de la acción de las reglas. Esto da como resultado

```
INSERT INTO shoelace_log SELECT
    *NEW*.sl_name, *NEW*.sl_avai,
    getpgusername(), datetime('now'::text)
FROM shoelace_data shoelace_data,
    shoelace_data *NEW*,
    shoelace_data *OLD*,
    shoelace_log shoelace_log;
```

En el segundo paso, se añade la cualificación de la regla, de modo que el resultado se restringe a las filas en las que sl_avail cambie.

```
INSERT INTO shoelace_log SELECT
    *NEW*.sl_name, *NEW*.sl_avai,
    getpgusername(), datetime('now'::text)
FROM shoelace_data shoelace_data, shoelace_data *NEW*,
    shoelace_data *OLD*, shoelace_log shoelace_log
WHERE int4ne(*NEW*.sl_avail, *OLD*.sl_avail);
```

En el tercer paso, se añade la cualificación de los árboles de traducción originales, restringiendo el juego de resultados más aún, a sólo las filas tocadas por el árbol de traducción original.

```
INSERT INTO shoelace_log SELECT
    *NEW*.sl_name, *NEW*.sl_avai,
    getpgusername(), datetime('now'::text)
FROM shoelace_data shoelace_data, shoelace_data *NEW*,
    shoelace_data *OLD*, shoelace_log shoelace_log
WHERE int4ne(*NEW*.sl_avail, *OLD*.sl_avail)
    AND bpchareq(shoelace_data.sl_name, 'sl7');
```

En el paso cuatro se sustituyen las referencias NEW por las entradas de la lista objetivo del árbol de traducción original o con las referencias a variables correspondientes de la relación resultado.

```
INSERT INTO shoelace_log SELECT
    shoelace_data.sl_name,
6,
    getpgusername(), datetime('now'::text)
FROM shoelace_data shoelace_data, shoelace_data *NEW*,
    shoelace_data *OLD*, shoelace_log shoelace_log
WHERE int4ne(6, *OLD*.sl_avail)
    AND bpchareq(shoelace_data.sl_name, 'sl7');
```

El paso 5 reemplaza las referencias OLD por referencias en la relación resultado.

```
INSERT INTO shoelace_log SELECT
    shoelace_data.sl_name, 6,
    getpgusername(), datetime('now'::text)
FROM shoelace_data shoelace_data, shoelace_data *NEW*,
    shoelace_data *OLD*, shoelace_log shoelace_log
WHERE int4ne(6, shoelace_data.sl_avail)
    AND bpchareq(shoelace_data.sl_name, 'sl7');
```

Y esto es. De modo que la máxima reducción de la salida del sistema de reglas es una lista de dos árboles de traducción que son lo mismo que las instrucciones:

```
INSERT INTO shoelace_log SELECT
    shoelace_data.sl_name, 6,
    getpgusername(), 'now'
FROM shoelace_data
WHERE 6 != shoelace_data.sl_avail
    AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
WHERE sl_name = 'sl7';
```

Estas con ejecutadas en este orden y eso es exactamente lo que la regla define. Las sustituciones y las cualificaciones añadidas aseguran que si la query original fuese una

```
UPDATE shoelace_data SET sl_color = 'green'
WHERE sl_name = 'sl7';
```

no se habría escrito ninguna entrada en la tabla de log, ya que esta vez el árbol de traducción original no contiene una entrada de la lista objetivo para sl_avail, de modo que NEW.sl_avail será reemplazada por shoelace_data.sl_avail resultando en la query adicional

```
INSERT INTO shoelace_log SELECT
```



```

        shoelace_data.sl_name,
    shoelace_data.sl_avail,
    getpgusername(), 'now'
FROM shoelace_data
WHERE shoelace_data.sl_avail !=
    shoelace_data.sl_avail
AND shoelace_data.sl_name = 'sl7';

```

cuya cualificación nunca será cierta. Una vez que no hay diferencias a nivel de árbol de traducción entre una INSERT ... SELECT, y una INSERT ... VALUES, trabajará también si la query original modificaba múltiples columnas. De modo que si Al hubiese pedido el comando

```

UPDATE shoelace_data SET sl_avail = 0
WHERE sl_color = 'black';

```

serán actualizadas cuatro filas (sl1, sl2, sl3 y sl4). Pero sl3 ya tiene sl_avail = 0. Esta vez, la cualificación de los árboles de traducción originales es diferente y como resultado tenemos el árbol de traducción adicional

```

INSERT INTO shoelace_log SELECT
    shoelace_data.sl_name, 0,
    getpgusername(), 'now'
FROM shoelace_data
WHERE 0 != shoelace_data.sl_avail
AND shoelace_data.sl_color = 'black';

```

Este árbol de traducción seguramente insertará tres nuevas entradas de la tabla de log. Y eso es absolutamente correcto.

Es importante recordar que el árbol de traducción original se ejecuta el último. El "agente de tráfico" de Postgres incrementa el contador de comandos entre la ejecución de los dos árboles de traducción, de modo que el segundo puede ver cambios realizados por el primero. Si la UPDATE hubiera sido ejecutada primero, todas las filas estarían ya a 0, de modo que la INSERT del logging no habría encontrado ninguna fila para las que shoelace_data.sl_avail != 0: no habría dejado ningún rastro.

Cooperación con las vistas

Una forma sencilla de proteger las relaciones vista de la mencionada posibilidad de que alguien pueda INSERT, UPDATE y DELETE datos invisibles es permitir a sus árboles de traducción recorrerlas de nuevo. Creamos las reglas

```

CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
DO INSTEAD NOTHING;

```

Si Al ahora intenta hacer cualquiera de estas operaciones en la relación vista shoe, el sistema de reglas aplicará las reglas. Una vez que las reglas no tienen acciones y son INSTEAD, la lista resultante de árboles de traducción estará vacía, y la query no devolverá nada, debido a que no hay nada para ser optimizado o ejecutado tras la actuación del sistema de reglas.

Nota: Este hecho debería irritar a las aplicaciones cliente, ya que no ocurre absolutamente nada en la base de datos, y por ello, el servidor no devuelve nada para la query. Ni siquiera un PGRES_EMPTY_QUERY o similar será utilizable en libpq. En psql, no ocurre nada. Esto debería cambiar en el futuro.

Una forma más sofisticada de utilizar el sistema de reglas es crear reglas que reescriban el árbol de traducción en uno que haga la operación correcta en las tablas reales. Para hacer esto en la vista `shoelace`, crearemos las siguientes reglas:

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
DO INSTEAD
INSERT INTO shoelace_data VALUES (
    NEW.sl_name,
    NEW.sl_avail,
    NEW.sl_color,
    NEW.sl_len,
    NEW.sl_unit);

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
DO INSTEAD
UPDATE shoelace_data SET
    sl_name = NEW.sl_name,
    sl_avail = NEW.sl_avail,
    sl_color = NEW.sl_color,
    sl_len = NEW.sl_len,
    sl_unit = NEW.sl_unit
WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
DO INSTEAD
DELETE FROM shoelace_data
WHERE sl_name = OLD.sl_name;
```

Ahora llega un paquete de cordones de zapatos a la tienda de Al, y el tiene una gran lista de artículos. Al no es particularmente bueno haciendo cálculos, y no lo queremos actualizando manualmente la vista `shoelace`. En su lugar, creamos dos tablas pequeñas, una donde él pueda insertar los datos de la lista de artículos, y otra con un truco especial. Los comandos `CREATE` completos son:

```
CREATE TABLE shoelace_arrive (
    arr_name    char(10),
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     char(10),
    ok_quant    integer
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
DO INSTEAD
UPDATE shoelace SET
    sl_avail = sl_avail + NEW.ok_quant
WHERE sl_name = NEW.ok_name;
```

ahora Al puede sentarse y hacer algo como:

```
al_bundy=> SELECT * FROM shoelace_arrive;
```

```

arr_name |arr_quant
-----+-----
sl3      |          10
sl6      |          20
sl8      |          20
(3 rows)

```

Que es exactametine lo que había en la lista de artículos. Daremos una rápida mirada en los datos actuales.

```

al_bundy=> SELECT * FROM shoelace ORDER BY sl_name;
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      |        6|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)

```

trasladamos los cordones recién llegados:

```
al_bundy=> INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

y comprobamos los resultados:

```

al_bundy=> SELECT * FROM shoelace ORDER BY sl_name;
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      |        6|brown     |  60|cm      |      60
sl4      |        8|black     |  40|inch    |    101.6
sl3      |       10|black     |  35|inch    |     88.9
sl8      |       21|brown     |  40|inch    |    101.6
sl5      |        4|brown     |   1|m       |     100
sl6      |       20|brown     |  0.9|m      |      90
(8 rows)

```

```

al_bundy=> SELECT * FROM shoelace_log;
sl_name  |sl_avail|log_who|log_when
-----+-----+-----+-----
sl7      |        6|Al     |Tue Oct 20 19:14:45 1998 MET DST
sl3      |       10|Al     |Tue Oct 20 19:25:16 1998 MET DST
sl6      |       20|Al     |Tue Oct 20 19:25:16 1998 MET DST
sl8      |       21|Al     |Tue Oct 20 19:25:16 1998 MET DST
(4 rows)

```

Esta es una larga vía desde la primera INSERT ... SELECT a estos resultados. Y su descripción será la última en este documento (pero no el último ejemplo :-). Primero estaba la salida de los traductores:

```

INSERT INTO shoelace_ok SELECT
    shoelace_arrive.arr_name, shoelace_arrive.arr_quant
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;

```

Ahora se aplica la primera regla 'shoelace_ok_in' y se vuelve:

```
UPDATE shoelace SET
```

```

        sl_avail = int4pl(shoelace.sl_avail, shoelace_arrive.arr_quant)
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
     shoelace_ok *OLD*, shoelace_ok *NEW*,
     shoelace shoelace
WHERE bpchareq(shoelace.sl_name, shoelace_arrive.arr_name);

```

y lanza otra vez la INSERT original sobre shoelace_ok. Esta query reescrita se pasa al sistema de reglas de nuevo, y la aplicación de la segunda regla 'shoelace_upd' produce

```

UPDATE shoelace_data SET
    sl_name = shoelace.sl_name,
    sl_avail = int4pl(shoelace.sl_avail, shoelace_arrive.arr_quant),
    sl_color = shoelace.sl_color,
    sl_len = shoelace.sl_len,
    sl_unit = shoelace.sl_unit
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
     shoelace_ok *OLD*, shoelace_ok *NEW*,
     shoelace shoelace, shoelace *OLD*,
     shoelace *NEW*, shoelace_data shoelace_data
WHERE bpchareq(shoelace.sl_name, shoelace_arrive.arr_name)
     AND bpchareq(shoelace_data.sl_name, shoelace.sl_name);

```

Otra vez es una regla INSTEAD, y el árbol de traducción anterior se deshecha. Nótese que esta query aún utiliza la vista shoelace. Pero el sistema de reglas no ha terminado con esta vuelta, de modo que continúa y aplica la regla '_RETshoelace', produciendo

```

UPDATE shoelace_data SET
    sl_name = s.sl_name,
    sl_avail = int4pl(s.sl_avail, shoelace_arrive.arr_quant),
    sl_color = s.sl_color,
    sl_len = s.sl_len,
    sl_unit = s.sl_unit
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
     shoelace_ok *OLD*, shoelace_ok *NEW*,
     shoelace shoelace, shoelace *OLD*,
     shoelace *NEW*, shoelace_data shoelace_data,
     shoelace *OLD*, shoelace *NEW*,
     shoelace_data s, unit u
WHERE bpchareq(s.sl_name, shoelace_arrive.arr_name)
     AND bpchareq(shoelace_data.sl_name, s.sl_name);

```

De nuevo se ha aplicado una regla de update y por ello vuelve a girar la rueda, y llegamos a la ronda de reescritura número 3. Esta vez, se aplica la regla 'log_shoelace', que produce el árbol de traducción extra

```

INSERT INTO shoelace_log SELECT
    s.sl_name,
    int4pl(s.sl_avail, shoelace_arrive.arr_quant),
    getpgusername(),
    datetime('now'::text)
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
     shoelace_ok *OLD*, shoelace_ok *NEW*,
     shoelace shoelace, shoelace *OLD*,
     shoelace *NEW*, shoelace_data shoelace_data,
     shoelace *OLD*, shoelace *NEW*,
     shoelace_data s, unit u,
     shoelace_data *OLD*, shoelace_data *NEW*
     shoelace_log shoelace_log
WHERE bpchareq(s.sl_name, shoelace_arrive.arr_name)

```

```

AND bpchareq(shoelace_data.sl_name, s.sl_name);
AND int4ne(int4pl(s.sl_avail, shoelace_arrive.arr_quant),
           s.sl_avail);

```

Tras de lo cual, el sistema de reglas se desconecta y devuelve los árboles de traducción generados. De esta forma, terminamos con dos árboles de traducción finales que son iguales a las instrucciones de SQL

```

INSERT INTO shoelace_log SELECT
    s.sl_name,
    s.sl_avail + shoelace_arrive.arr_quant,
    getpgusername(),
    'now'
FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
     shoelace_data s
WHERE s.sl_name = shoelace_arrive.arr_name
     AND shoelace_data.sl_name = s.sl_name
     AND s.sl_avail + shoelace_arrive.arr_quant != s.sl_avail;

UPDATE shoelace_data SET
    sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
FROM shoelace_arrive shoelace_arrive,
     shoelace_data shoelace_data,
     shoelace_data s
WHERE s.sl_name = shoelace_arrive.sl_name
     AND shoelace_data.sl_name = s.sl_name;

```

El resultado es que los datos vienen de una relación, se insertan en otra, cambian por actualizaciones una tercera, cambian por actualizaciones una cuarta, más registran esa actualización final en una quinta: todo eso se reduce a dos queries.

Hay un pequeño detalle un tanto desagradable. Mirando en las dos queries, descubrimos que la relación `shoelace_data` aparece dos veces en la tabla de rango, lo que se debería reducir a una sola. El optimizador no manipula esto, y por ello el plan de ejecución para la salida del sistema de reglas de la INSERT será

```

Nested Loop
-> Merge Join
    -> Seq Scan
        -> Sort
            -> Seq Scan on s
    -> Seq Scan
        -> Sort
            -> Seq Scan on shoelace_arrive
-> Seq Scan on shoelace_data

```

mientras que omitiendo la entrada extra a la tabla de rango debería ser

```

Merge Join
-> Seq Scan
    -> Sort
        -> Seq Scan on s
-> Seq Scan
    -> Sort
        -> Seq Scan on shoelace_arrive

```

que produce exactamente las mismas entradas en la relación de log. Es decir, el sistema de reglas ha provocado un barrido extra de la relación `shoelace_data` absolutamente innecesario. Y el mismo barrido obsoleto se produce de nuevo en la UPDATE. Pero era un trabajo realmente duro hacer que todo sea posible.

Una demostración final del sistema de reglas de Postgres y de su poder. Hay una astuta rubia que vende cordones de zapatos. Y lo que Al nunca hubiese imaginado, ella no es sólo astuta, también es elegante, un poco demasiado elegante. Por ello, ella se empeña de tiempo en tiempo en que Al pida cordones que son absolutamente invendibles. Esta vez ha pedido 1000 pares de cordones magenta, y aunque ahora no es posible adquirir otro color, como él se comprometió a comprar algo, prepara su base de datos para cordones rosa.

```
al_bundy=> INSERT INTO shoelace VALUES
al_bundy->      ('s19', 0, 'pink', 35.0, 'inch', 0.0);
al_bundy=> INSERT INTO shoelace VALUES
al_bundy->      ('s110', 1000, 'magenta', 40.0, 'inch', 0.0);
```

Ahora quiere revisar los cordones que no casan con ningún par de zapatos. El podría realizar una complicada query cada vez, o bien le podemos preparar una vista al efecto:

```
CREATE VIEW shoelace_obsolete AS
SELECT * FROM shoelace WHERE NOT EXISTS
(SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

cuya salida es

```
al_bundy=> SELECT * FROM shoelace_obsolete;
sl_name  |sl_avail|sl_color  |sl_len|sl_unit  |sl_len_cm
-----+-----+-----+-----+-----+-----
s19      |      0|pink      |   35|inch     |    88.9
s110     |    1000|magenta    |   40|inch     |   101.6
```

Sobre los 1000 cordones magenta, deberíamos avisar a Al antes de que podamos hacerlo de nuevo, pero ese es otro problema. La entrada rosa, la borramos. Para hacerlo un poco más difícil para Postgres, no la borramos directamente. En su lugar, crearemos una nueva vista

```
CREATE VIEW shoelace_candelelete AS
SELECT * FROM shoelace_obsolete WHERE sl_avail = 0;
```

Y lo haremos de esta forma:

```
DELETE FROM shoelace WHERE EXISTS
(SELECT * FROM shoelace_candelelete
WHERE sl_name = shoelace.sl_name);
```

Voila:

```
al_bundy=> SELECT * FROM shoelace;
sl_name  |sl_avail|sl_color  |sl_len|sl_unit  |sl_len_cm
-----+-----+-----+-----+-----+-----
s11      |      5|black     |   80|cm       |    80
s12      |      6|black     |  100|cm       |   100
s17      |      6|brown     |   60|cm       |    60
s14      |      8|black     |   40|inch     |   101.6
s13      |     10|black     |   35|inch     |    88.9
s18      |     21|brown     |   40|inch     |   101.6
s110     |    1000|magenta    |   40|inch     |   101.6
s15      |      4|brown     |    1|m       |    100
s16      |     20|brown     |   0.9|m       |     90
(9 rows)
```

Una DELETE en una vista, con una subselect como cualificación, que en total utiliza 4 vistas anidadas/cruzadas, donde una de ellas mismas tiene una subselect de cualificación conteniendo una vista y donde se utilizan columnas calculadas queda reescrita en un único árbol de traducción que borra los datos requeridos de una tabla real.

Pienso que hay muy pocas ocasiones en el mundo real en las que se una construcción similar sea necesaria. Pero me tranquiliza un poco que esto funcione.

La verdad es: Haciendo esto encontré otro bug mientras escribía este documento. Pero tras fijarlo comprobé un poco avergonzado que trabajaba correctamente.

Reglas y permisos

Debido a la reescritura de las queries por el sistema de reglas de Postgre, se han accedido a otras tablas/vistas diferentes de las de la query original. Utilizando las reglas de update, esto puede incluir acceso en escritura a tablas.

Las reglas de reescritura no tienen un propietario diferenciado. El propietario de una relación (tabla o vista) es automáticamente el propietario de las reglas de reescritura definidas para ella. El sistema de reglas de Postgres cambia el comportamiento del sistema de control de acceso de defecto. Las relaciones que se utilizan debido a las reglas son comprobadas durante la reescritura contra los permisos del propietario de la relación, contra la que la regla se ha definido. Esto hace que el usuario no necesite sólo permisos para las tablas/vistas a las que él hace referencia en sus queries.

Por ejemplo: Un usuario tiene una lista de números de teléfono en la que algunos son privados y otros son de interés para la secretaria en la oficina. Él puede construir lo siguiente:

```
CREATE TABLE phone_data (person text, phone text, private bool);
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE NOT private;
GRANT SELECT ON phone_number TO secretary;
```

Nadie excepto él, y el superusuario de la base de datos, pueden acceder a la tabla phone_data. Pero debido a la GRANT, la secretaria puede SELECT a través de la vista phone_numero. El sistema de reglas reescribirá la SELECT de phone_numero en una SELECT de phone_data y añade la cualificación de que sólo se buscan las entradas cuyo "privado" sea falso. Una vez que el usuario sea el propietario de phone_numero, la lectura accede a phone_data se comprueba contra sus permisos, y la query se considera autorizada. La comprobación para acceder a phone_number se realiza entonces, de modo que nadie más que la secretaria pueda utilizarlo.

Los permisos son comprobados regla a regla. De modo que la secretaria es ahora la única que puede ver los números de teléfono públicos. Pero la secretaria puede crear otra vista y autorizar el acceso a ella al público. Entonces, cualquiera puede ver los datos de phone_numero a través de la vista de la secretaria. Lo que la secretaria no puede hacer es crear una vista que acceda directamente a phone_data (realmente si puede, pero no trabajará, puesto que cada acceso abortará la transacción durante la comprobación de los permisos). Y tan pronto como el usuario tenga noticia de que la secretaria ha abierto su vista a phone_numero, el puede REVOKE su acceso. Inmediatamente después, cualquier acceso a la vista de las secretarias fallará.

Alguien podría pensar que este chequeo regla a regla es un agujero de seguridad, pero de hecho no lo es. Si esto no trabajase, la secretaria podría generar una tabla con las mismas columnas de phone_number y copiar los datos aquí todos los días. En este caso serían ya sus propios datos, y podría autorizar el acceso a cualquiera que ella quisiera. Un GRANT quiere decir "Yo Confío en Tí". Si alguien en quien confiamos hace lo anterior, es el momento de volver sobre nuestros pasos, y hacer el REVOKE.

Este mecanismo también trabaja para reglas de update. En el ejemplo de la sección previa, el propietario de las tablas de la base de datos de AI (suponiendo que no fuera el mismo AI) podría haber autorizado (GRANT) SELECT, INSERT, UPDATE o DELETE a la vista shoelace a AI. Pero sólo SELECT en shoelace_log. La acción de la regla de escribir entradas del log deberá ser ejecutada con éxito, y AI podría ver las entradas del log, pero el no puede crear nuevas entradas, ni podría manipular ni remover las existentes.

Atención: GRANT ALL actualmente incluye permisos RULE. Esto permite al usuario autorizado borrar la regla, hacer los cambios y reinstalarla. Pienso que esto debería ser cambiado rápidamente.

Reglas frente triggers

Son muchas las cosas que se hacen utilizando triggers que pueden hacerse también utilizando el sistema de las reglas de Postgres. Lo que actualmente no se puede implementar a través de reglas son algunos tipos de restricciones (constraints). Es posible situar una regla cualificada que reescriba una query a NOTHING si el valor de la columna no aparece en otra tabla, pero entonces los datos son eliminados silenciosamente, y eso no es una buena idea. Si se necesitan comprobaciones para valores válidos, y en el caso de aparecer un valor inválido dar un mensaje de error, eso deberá hacerse por ahora con un trigger.

Por otro lado, un trigger que se dispare a partir de una INSERT en una vista puede hacer lo mismo que una regla, situar los datos en cualquier otro sitio y suprimir la inserción en una vista. Pero no puede hacer lo mismo en una UPDATE o una DELETE, porque no hay datos reales en la relación vista que puedan ser comprobados, y por ello el trigger nunca podría ser llamado. Sólo una regla podría ayudarnos.

Para los tratamientos que podrían implementarse de ambas formas, dependerá del uso de la base de datos cuál sea la mejor. Un trigger se dispara para cada fila afectada. Una regla manipula el árbol de traducción o genera uno adicional. De modo que si se manipulan muchas filas en una instrucción, una regla ordenando una query adicional usualmente daría un mejor resultado que un trigger que se llama para cada fila individual y deberá ejecutar sus operaciones muchas veces.

Por ejemplo: hay dos tablas.

```
CREATE TABLE computer (
    hostname          text          - indexed
    manufacturer      text          - indexed
);

CREATE TABLE software (
    software          text,         - indexed
    hostname          text          - indexed
);
```


Ambas tablas tienen muchos millares de filas y el índice sobre hostname es único. La columna hostname contiene el nombre de dominio cualificado completo del ordenador. La regla/trigger debería desencadenar el borrado de filas de la tabla software que se refieran a un host borrado. Toda vez que el trigger se llama para cada fila individual borrada de computer, se puede usar la instrucción

```
DELETE FROM software WHERE hostname = $1;
```

en un plan preparado y salvado, y pasar el hostname en el parámetro. La regla debería ser escrita como

```
CREATE RULE computer_del AS ON DELETE TO computer
DO DELETE FROM software WHERE hostname = OLD.hostname;
```

Veremos ahora en que se diferencian los dos tipos de delete. En el caso de una

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

La tabla computer se revisa por índice (rápido) y la query lanzada por el trigger también debería ser un barrido de índice (rápido también). La query extra para la regla sería una

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
AND software.hostname = computer.hostname;
```

Puesto que se han creado los índices apropiados, el optimizador creará un plan de

```
Nestloop
-> Index Scan using comp_hostidx on computer
-> Index Scan using soft_hostidx on software
```

De modo que no habría mucha diferencia de velocidad entre la implementación del trigger y de la regla. Con la siguiente delete, queremos mostrar borrar los 2000 ordenadores cuyo hostname empieza con 'old'. Hay dos posibles queries para hacer eso. Una es

```
DELETE FROM computer WHERE hostname >= 'old'
AND hostname < 'ole'
```

Donde el plan de ejecución para la query de la regla será

```
Hash Join
-> Seq Scan on software
-> Hash
-> Index Scan using comp_hostidx on computer
```

La otra query posible es

```
DELETE FROM computer WHERE hostname ~ '^old';
```

con un plan de ejecución

```
Nestloop
-> Index Scan using comp_hostidx on computer
-> Index Scan using soft_hostidx on software
```

Esto muestra que el optimizador no comprueba que la cualificación sobre hostname en computer también debería ser utilizado para un barrido por índice en software donde hay múltiples expresiones de cualificación combinadas con AND, que el hace en la versión regexp de la query. El trigger será invocado una vez para cada una de los

2000 viejos ordenadores que serán borrados, lo que dará como resultado un barrido por índice sobre computer y 2000 barridos por índice sobre software. La implementación de la regla lo hará con dos queries sobre índices. Y dependerá del tamaño promedio de la tabla software si la regla será más rápida en una situación de barrido secuencial. 2000 ejecuciones de queries sobre el gestor SPI toman su tiempo, incluso si todos los bloques del índice se encuentran en la memoria caché.

La última query que veremos es

```
DELETE FROM computer WHERE manufacturer = 'bim';
```

De nuevo esto debería dar como resultado muchas filas para borrar de computer. Por ello el trigger disparará de nuevo muchas queries sobre el ejecutor. Pero el plan de las reglas será de nuevo un bucle anidado sobre dos barridos de índice. Sólo usando otro índice en computer:

```
Nestloop
->  Index Scan using comp_manufidx on computer
->  Index Scan using soft_hostidx on software
```

dando como resultado de la query de las reglas

```
DELETE FROM software WHERE computer.manufacturer = 'bim'
AND software.hostname = computer.hostname;
```

En cualquiera de estos casos, las queries extra del sistema de reglas serán más o menos independientes del número de filas afectadas en la query.

Otra situación son los casos de UPDATE donde depende del cambio de un atributo si la acción debería realizarse o no. En la versión 6.4 de Postgres, la especificación de atributos para acontecimientos de reglas se ha deshabilitado (y tendrá su regreso en la 6.5, quizá antes ¡permanezcan en antena!). De modo que por ahora la única forma de crear una regla como en el ejemplo de shoelace_log es hacerlo con una cualificación de la regla. Eso da como resultado una query adicional que se realiza siempre, incluso si el atributo que nos interesa no puede ser cambiado de ninguna forma porque no aparece en la lista objetivo de la query inicial. Cuando se habilite de nuevo, será una nueva ventaja del sistema de reglas sobre los triggers. La optimización de un trigger deberá fallar por definición en este caso, porque el hecho de que su acción solo se hará cuando un atributo específico sea actualizado, está oculto a su funcionalidad. La definición de un trigger sólo permite especificar el nivel de fila, de modo que si se toca una fila, el trigger será llamado a hacer su trabajo. El sistema de reglas lo sabrá mirando la lista objetivo y suprimirá la query adicional por completo si el atributo no se ha tocado. De modo que la regla, cualificada o no, sólo hará sus barridos si tiene algo que hacer.

Las reglas sólo serán significativamente más lentas que los triggers si sus acciones dan como resultado joins grandes y mal cualificadas, una situación en la que falla el optimizador. Tenemos un gran martillo. Utilizar un gran martillo sin cuidado puede causar un gran daño, pero dar el toque correcto, puede hundir cualquier clavo hasta la cabeza.

Capítulo 38. Utilización de las Extensiones en los Índices

Los procedimientos descritos hasta aquí le permiten definir un nuevo tipo, nuevas funciones y nuevos operadores. Sin embargo, todavía no podemos definir un índice secundario (tal como un B-tree, R-tree o método de acceso hash) sobre un nuevo tipo o sus operadores.

Mírese nuevamente *El principal sistema de catalogo de Postgres*. La mitad derecha muestra los catálogos que debemos modificar para poder indicar a Postgres cómo utilizar un tipo definido por el usuario y/u operadores definidos por el usuario con un índice (es decir, `pg_am`, `pg_amop`, `pg_amproc`, `pg_operator` y `pg_opclass`). Desafortunadamente, no existe un comando simple para hacer esto. Demostraremos cómo modificar estos catálogos a través de un ejemplo ejecutable: una nueva clase de operador para el método de acceso B-tree que almacene y ordene números complejos en orden ascendente de valor absoluto.

La clase `pg_am` contiene una instancia para cada método de acceso definido por el usuario. El soporte de acceso a la memoria (heap) está integrado en Postgres, pero todos los demás métodos de acceso están descritos aquí. El esquema es

Tabla 38-1. Esquema de un Índice

Atributo	Descripción
<code>amname</code>	nombre del método de acceso
<code>amowner</code>	identificador de objeto del propietario de esta instancia en <code>pg_user</code>
<code>amstrategies</code>	número de estrategias para este método de acceso (véase más abajo)
<code>amsupport</code>	número de rutinas de soporte para este método de acceso (véase más abajo)
<code>amorderstrategy</code>	cero si el índice no ofrece secuencia de ordenamiento, sino el número de estrategia del operador de estrategia que describe la secuencia de ordenamiento
<code>amgettuple</code>	
<code>aminsert</code>	
...	indicadores de procedimiento para las rutinas de interfaz con el método de acceso. Por ejemplo, aquí aparecen identificadores <code>regproc</code> para abrir, cerrar y obtener instancias desde el método de acceso

El identificador de objeto (object ID) de la instancia en `pg_am` se utiliza como una clave foránea en multitud de otras clases. No es necesario que Ud. agregue una nueva instancia en esta clase; lo que debe interesarle es el identificador de objeto (object ID) de la instancia del método de acceso que quiere extender:

```
SELECT oid FROM pg_am WHERE amname = 'btree';
```

```
+---+  
|oid |
```

```

+---+
| 403 |
+---+

```

Utilizaremos ese comando **SELECT** en una cláusula **WHERE** posterior.

El atributo `amstrategies` tiene como finalidad estandarizar comparaciones entre tipos de datos. Por ejemplo, los B-trees imponen un ordenamiento estricto en las claves, de menor a mayor. Como Postgres permite al usuario definir operadores, no puede, a través del nombre del operador (por ej., ">" or "<"), identificar qué tipo de comparación es. De hecho, algunos métodos de acceso no imponen ningún ordenamiento. Por ejemplo, los R-trees expresan una relación de inclusión en un rectángulo, mientras que una estructura de datos de tipo hash expresa únicamente similitud de bits basada en el valor de una función hash. Postgres necesita alguna forma consistente para interpretar los requisitos en sus consultas, identificando el operador y decidiendo si se puede utilizar un índice existente. Esto implica que Postgres necesita conocer, por ejemplo, que los operadores "<=" y ">" particionan un B-tree. Postgres utiliza estrategias para expresar esas relaciones entre los operadores y las formas en que pueden utilizarse al recorrer los índices.

Definir un nuevo conjunto de estrategias está más allá del alcance de esta exposición, pero explicaremos cómo funcionan las estrategias B-tree porque necesitará conocerlas para agregar una nueva clase de operador. En la clase `pg_am`, el atributo `amstrategies` es el número de estrategias definidas para este método de acceso. Para los B-trees, este número es 5. Estas estrategias corresponden a

Tabla 38-2. Estrategias B-tree

Operación	Índice
menor que	1
menor que o igual a	2
igual	3
mayor que o igual a	4
mayor que	5

La idea es que será necesario agregar procedimientos correspondientes a las comparaciones mencionadas arriba a la tabla `pg_amop` (véase más abajo). El código de método de acceso puede utilizar estos números de estrategia, sin tener en cuenta el tipo de datos, para resolver cómo particionar el B-tree, calcular la selectividad, etcétera. No se preocupe aún acerca de los detalles para agregar procedimientos; sólo comprenda que debe existir un conjunto de procedimientos para `int2`, `int4`, `oid`, y todos los demás tipos de datos donde puede operar un B-tree.

Algunas veces, las estrategias no proporcionan la información suficiente para resolver la forma de utilizar un índice. Algunos métodos de acceso requieren otras rutinas de soporte para poder funcionar. Por ejemplo, el método de acceso B-tree debe ser capaz de comparar dos claves y determinar si una es mayor que, igual a, o menor que la otra. De manera análoga, el método de acceso R-tree debe ser capaz de calcular intersecciones, uniones, y tamaños de rectángulos. Estas operaciones no corresponden a requisitos del usuario en las consultas SQL; son rutinas administrativas utilizadas por los métodos de acceso, internamente.

Para manejar diversas rutinas de soporte consistentemente entre todos los métodos de acceso de Postgres, `pg_am` incluye un atributo llamado `amsupport`. Este atributo

almacena el número de rutinas de soporte utilizadas por un método de acceso. Para los B-trees, este número es uno – la rutina que toma dos claves y devuelve -1, 0, o +1, dependiendo si la primer clave es menor que, igual a, o mayor que la segunda.

Nota: En términos estrictos, esta rutina puede devolver un número negativo (< 0), 0, o un valor positivo distinto de cero (> 0).

La entrada `amstrategies` en `pg_am` sólo indica el número de estrategias definidas para el método de acceso en cuestión. Los procedimientos para menor que, menor que o igual a, etcétera no aparecen en `pg_am`. De manera similar, `amsupport` es solamente el número de rutinas de soporte que requiere el método de acceso. Las rutinas reales están listadas en otro lado.

Además, la entrada `amorderstrategy` indica si el método de acceso soporta o no un recorrido ordenado. Cero significa que no; si lo hace, `amorderstrategy` es el número de la rutina de estrategia que corresponde al operador de ordenamiento. Por ejemplo, `btree` tiene `amorderstrategy` = 1 que corresponde al número de estrategia de "menor que".

La próxima clase de interés es `pg_opclass`. Esta clase tiene como única finalidad asociar un nombre y tipo por defecto con un oid. En `pg_amop` cada clase de operador B-tree tiene un conjunto de procedimientos, de uno a cinco, descritos más arriba. Algunas clases de operadores (`opclasses`) son `int2_ops`, `int4_ops`, y `oid_ops`. Es necesario que Ud. agregue una instancia con su nombre de clase de operador (por ejemplo, `complex_abs_ops`) a `pg_opclass`. El oid de esta instancia es una clave foránea en otras clases.

```
INSERT INTO pg_opclass (opcname, opcdeftype)
SELECT 'complex_abs_ops', oid FROM pg_type WHERE typename = 'complex_abs';

SELECT oid, opcname, opcdeftype
FROM pg_opclass
WHERE opcname = 'complex_abs_ops';
```

```
+---+-----+-----+
|oid  | opcname          | opcdeftype |
+---+-----+-----+
|17314| complex_abs_ops  |          29058 |
+---+-----+-----+
```

¡Nótese que el oid para su instancia de `pg_opclass` será diferente! No se preocupe por esto. Obtendremos este número del sistema después igual que acabamos de hacerlo con el oid del tipo aquí.

De esta manera ahora tenemos un método de acceso y una clase de operador. Aún necesitamos un conjunto de operadores; el procedimiento para definir operadores fue discutido antes en este manual. Para la clase de operador `complex_abs_ops` en Btrees, los operadores que necesitamos son:

```
valor absoluto menor que (absolute value less-than)
valor absoluto menor que o igual a (absolute value less-than-
or-equal)
valor absoluto igual (absolute value equal)
valor absoluto mayor que o igual a (absolute value greater-than-
or-equal)
valor absoluto mayor que (absolute value greater-than)
```

Supongamos que el código que implementa las funciones definidas está almacenado en el archivo `PGROOT/src/tutorial/complex.c`

Parte del código será parecido a este: (nótese que solamente mostraremos el operador de igualdad en el resto de los ejemplos. Los otros cuatro operadores son muy similares. Refiérase a `complex.co` `complex.source` para más detalles.)

```
#define Mag(c) ((c)->x*(c)->x + (c)->y*(c)->y)

bool
complex_abs_eq(Complex *a, Complex *b)
{
    double amag = Mag(a), bmag = Mag(b);
    return (amag==bmag);
}
```

Hay un par de cosas importantes que suceden arriba.

Primero, nótese que se están definiendo operadores menor que, menor que o igual a, igual, mayor que o igual a, y mayor que para `int4`. Todos estos operadores ya están definidos para `int4` bajo los nombres `<`, `<=`, `=`, `>=`, and `>`. Los nuevos operadores, por supuesto, se comportan de manera distinta. Para garantizar que Postgres usa estos nuevos operadores en vez de los anteriores, es necesario que sean nombrados distinto que ellos. Este es un punto clave: Ud. puede sobrecargar operadores en Postgres, pero sólo si el operador no ha sido definido aún para los tipos de los argumentos. Es decir, si Ud. tiene `<` definido para (`int4`, `int4`), no puede definirlo nuevamente. Postgres no comprueba esto cuando define un nuevo operador, así es que debe ser cuidadoso. Para evitar este problema, se utilizarán nombres dispares para los operadores. Si hace esto mal, los métodos de acceso seguramente fallen cuando intente hacer recorridos.

El otro punto importante es que todas las funciones de operador devuelven valores lógicos (Boolean). Los métodos de acceso cuentan con este hecho. (Por otro lado, las funciones de soporte devuelven cualquier cosa que el método de acceso particular espera – en este caso, un entero con signo.) La rutina final en el archivo es la "rutina de soporte" mencionada cuando tratábamos el atributo `amsupport` de la clase `pg_am`. Utilizaremos esto más adelante. Por ahora, ignórelo.

```
CREATE FUNCTION complex_abs_eq(complex_abs, complex_abs)
    RETURNS bool
    AS 'PGROOT/tutorial/obj/complex.so'
    LANGUAGE 'c';
```

Ahora defina los operadores que los utilizarán. Como se hizo notar, los nombres de operadores deben ser únicos entre todos los operadores que toman dos operandos `int4`. Para ver si los nombres de operadores listados más arriba ya han sido ocupados, podemos hacer una consulta sobre `pg_operator`:

```
/*
 * esta consulta utiliza el operador de expresión regular (~)
 * para encontrar nombres de operadores de tres caracteres que terminen
 * con el carácter &
 */
SELECT *
```

```
FROM pg_operator
WHERE oprname ~ '^..&$'::text;
```

para ver si su nombre ya ha sido ocupado para los tipos que Ud. quiere. Las cosas importantes aquí son los procedimientos (que son las funciones Cdefinidas más arriba) y las funciones de restricción y de selectividad de unión. Ud. debería utilizar solamente las que se usan abajo – nótese que hay distintas funciones para los casos menor que, igual, y mayor que. Éstas deben proporcionarse, o el método de acceso fallará cuando intente utilizar el operador. Debería copiar los nombres para las funciones de restricción y de unión, pero utilice los nombres de procedimiento que definió en el último paso.

```
CREATE OPERATOR = (
    leftarg = complex_abs, rightarg = complex_abs,
    procedure = complex_abs_eq,
    restrict = eqsel, join = eqjoinsel
)
```

Téngase en cuenta que se definen cinco operadores correspondientes a menor, menor o igual, igual, mayor, y mayor o igual.

Ya casi hemos terminado. La última cosa que necesitamos hacer es actualizar la tabla `pg_amop`. Para hacer esto, necesitamos los siguientes atributos:

Tabla 38-3. Esquema de `pg_amproc`

Atributo	Descripción
amopid	el oid de la instancia de <code>pg_am</code> para B-tree (== 403, véase arriba)
amopclaid	el oid de la instancia de <code>pg_opclass</code> para <code>complex_abs_ops</code> (== lo que obtuvo en vez de 17314, véase arriba)
amopopr	los oids de los operadores para la clase de operador (opclass) (que obtendremos dentro de un minuto)

Entonces necesitamos los oids de los operadores que acabamos de definir. Buscaremos los nombres de todos los operadores que toman dos argumentos de tipo `complex`, y así sacaremos los nuestros:

```
SELECT o.oid AS opoid, o.oprname
INTO TABLE complex_ops_tmp
FROM pg_operator o, pg_type t
WHERE o.oprleft = t.oid and o.oprright = t.oid
and t.typname = 'complex_abs';
```

```
+----+-----+
|oid  | oprname |
+----+-----+
|17321| <      |
+----+-----+
```

```

|17322 | <= |
+-----+-----+
|17323 | =   |
+-----+-----+
|17324 | >= |
+-----+-----+
|17325 | >   |
+-----+-----+

```

(De nuevo, algunos de sus números de `oid` serán seguramente diferentes.) Los operadores en los que estamos interesados son los que tienen `oids` 17321 hasta 17325. Los valores que Ud. obtendrá serán probablemente distintos, y debe sustituirlos abajo por estos valores. Haremos esto con una sentencia `SELECT`.

Ahora estamos listos para actualizar `pg_amop` con nuestra nueva clase de operador. La cosa más importante en toda esta explicación es que los operadores están ordenados desde menor que hasta mayor que, en `pg_amop`. Agregamos las instancias necesarias:

```

INSERT INTO pg_amop (amopid, amopclaid, amopopr, amopstrategy)
SELECT am.oid, opcl.oid, c.opoid, 1
FROM pg_am am, pg_opclass opcl, complex_abs_ops_tmp c
WHERE amname = 'btree' AND
      opcname = 'complex_abs_ops' AND
      c.oprname = '<';

```

Ahora haga lo mismo con los otros operadores sustituyendo el "1" en la tercera línea de arriba y el "<" en la última línea. Nótese el orden: "menor que" es 1, "menor que o igual a" es 2, "igual" es 3, "mayor que o igual a" es 4, y "mayor que" es 5.

El próximo paso es registrar la "rutina de soporte" previamente descrita en la explicación de `pg_am`. El `oid` de esta rutina de soporte está almacenada en la clase `pg_amproc`, cuya clave está compuesta por el `oid` del método de acceso y el `oid` de la clase de operador. Primero, necesitamos registrar la función en Postgres (recuerde que pusimos el código C que implementa esta rutina al final del archivo en el cual implementamos las rutinas del operador):

```

CREATE FUNCTION complex_abs_cmp(complex, complex)
RETURNS int4
AS 'PGROOT/tutorial/obj/complex.so'
LANGUAGE 'c';

SELECT oid, proname FROM pg_proc
WHERE proname = 'complex_abs_cmp';

```

```

+-----+-----+
|oid    | proname          |
+-----+-----+
|17328 | complex_abs_cmp |
+-----+-----+

```

(De nuevo, su número de `oid` será probablemente distinto y debe sustituirlo abajo por el valor que vea.) Podemos agregar la nueva instancia de la siguiente manera:

```

INSERT INTO pg_amproc (amid, amopclaid, amproc, amprocnum)
SELECT a.oid, b.oid, c.oid, 1
FROM pg_am a, pg_opclass b, pg_proc c
WHERE a.amname = 'btree' AND

```



```

b.opcname = 'complex_abs_ops' AND
c.proname = 'complex_abs_cmp';

```

Ahora necesitamos agregar una estrategia de hash para permitir que el tipo sea indexado. Hacemos esto utilizando otro tipo en pg_am pero reutilizamos los mismos operadores.

```

INSERT INTO pg_amop (amopid, amopclaid, amopopr, amopstrategy)
SELECT am.oid, opcl.oid, c.opoid, 1
FROM pg_am am, pg_opclass opcl, complex_abs_ops_tmp c
WHERE amname = 'hash' AND
      opcname = 'complex_abs_ops' AND
      c.oprname = '=';

```

Para utilizar este índice en una cláusula WHERE, necesitamos modificar la clase pg_operator de la siguiente manera.

```

UPDATE pg_operator
SET oprrest = 'eqsel'::regproc, oprjoin = 'eqjoinsel'
WHERE oprname = '=' AND
      oprleft = oprright AND
      oprleft = (SELECT oid FROM pg_type WHERE typename = 'complex_abs');

UPDATE pg_operator
SET oprrest = 'neqsel'::regproc, oprjoin = 'neqjoinsel'
WHERE oprname = '' AND
      oprleft = oprright AND
      oprleft = (SELECT oid FROM pg_type WHERE typename = 'complex_abs');

UPDATE pg_operator
SET oprrest = 'neqsel'::regproc, oprjoin = 'neqjoinsel'
WHERE oprname = '' AND
      oprleft = oprright AND
      oprleft = (SELECT oid FROM pg_type WHERE typename = 'complex_abs');

UPDATE pg_operator
SET oprrest = 'scalartsel'::regproc, oprjoin = 'scalartjoinsel'
WHERE oprname = '<' AND
      oprleft = oprright AND
      oprleft = (SELECT oid FROM pg_type WHERE typename = 'complex_abs');

UPDATE pg_operator
SET oprrest = 'scalartsel'::regproc, oprjoin = 'scalartjoinsel'
WHERE oprname = '<=' AND
      oprleft = oprright AND
      oprleft = (SELECT oid FROM pg_type WHERE typename = 'complex_abs');

UPDATE pg_operator
SET oprrest = 'scalargtsel'::regproc, oprjoin = 'scalargtjoinsel'
WHERE oprname = '>' AND
      oprleft = oprright AND
      oprleft = (SELECT oid FROM pg_type WHERE typename = 'complex_abs');

UPDATE pg_operator
SET oprrest = 'scalargtsel'::regproc, oprjoin = 'scalargtjoinsel'
WHERE oprname = '>=' AND
      oprleft = oprright AND

```

```
oprleft = (SELECT oid FROM pg_type WHERE typname = 'complex_abs');
```

Y por último (¡por fin!) registramos una descripción de este tipo.

```
INSERT INTO pg_description (objoid, description)
SELECT oid, 'Two part G/L account'
FROM pg_type WHERE typname = 'complex_abs';
```

Capítulo 39. GiST Indices

La información sobre GiST está en <http://GiST.CS.Berkeley.EDU:8000/gist/>¹ con más sobre diferentes esquemas de ordenación e indexado en <http://s2k-ftp.CS.Berkeley.EDU:8000/persona>. También existe más lectura interesante en el sitio de la base de datos de Berkeley en <http://epoch.cs.berkeley.edu:8000/>³.

Autor: Esta extracción de un e-mail enviado por Eugene Selkov Jr.⁴ contiene buena información sobre GiST. Seguramente aprenderemos más en el futuro y actualizaremos esta información. - thomas 1998-03-01

Bueno, no puedo decir que entienda lo que está pasando, pero por lo menos (casi) he logrado portar los ejemplos GiST a linux. El método de acceso GiST ya está en el árbol de postfres (`src/backend/access/gist`).

Examples at Berkeley⁵ vienen con una introducción de los métodos y demuestran mecanismos de índices espaciales para cajas 2D, polígonos, intervalos enteros y texto come with an overview of the methods and demonstrate spatial index mechanisms for 2D boxes, polygons, integer intervals and text (véase también GiST at Berkeley⁶). En el ejemplo de la caja, se supone que veremos un aumento en el rendimiento al utilizar el índice GiST; a mí me funcionó, pero yo no tengo una colección razonablemente grande de cajas para comprobar. Otros ejemplos también funcionaron, excepto polígonos: obtuve un error al hacer

```
test=> create index pix on polytmp
test-> using gist (p:box gist_poly_ops) with (islossy);
ERROR:  cannot open pix
```

(PostgreSQL 6.3

Sun Feb 1 14:57:30 EST 1998)

No entiendo el sentido de este mensaje de error; parece ser algo que deberíamos preguntar a los desarrolladores (mira también la Nota 4 más abajo). Lo que sugeriría aquí es que alguien de vosotros, gurús de Linux (linux==gcc?), tomeis las fuentes originales citadas arriba y apliqueis mi parche (véase el adjunto) y nos dijeseis que pensais sobre esto. Me parece muy bien a mí, pero no me gustaría mantenerlo mientras que hay tanta gente competente disponible.

Unas pocas notas en los fuentes:

1. No fui capaz de utilizar el Makefile original (HPUX) y reordenarlo con el viejo tutorial de postgres95 para hacerlo funcionar. Intenté mantenerlo genérico, pero no soy un escritor de makefiles muy pobre –simplemente lo hizo funcionar algún mono. Lo siento, pero creo que ahora es un poco más portable que el makefile original.
2. Compilé las fuentes de ejemplo inmediatamente debajo de `pgsql/src` (simplemente extraje el archivo tar allí). El Makefile previamente mencionado supone que está un nivel por debajo de `pgsql/src` (en nuestro caso, en `pgsql/src/pggist`).
3. Los cambios que hice a los ficheros *.c fueron todos sobre #includes's, prototipos de funciones y typecasting. Fuera de eso, solamente deseché una ristra de variables no utilizadas y añadí un par de parentesis para contentar a gcc. Espero que esto no haya enredado las cosas mucho :)
4. Hay un comentario en `polyproc.sql`:

```
- - there's a memory leak in rtree poly_ops!!
- - create index pix2 on polytmp using rtree (p poly_ops);
(- - existe una fuga de memoria en el rtree poly_ops!!)
(- - crea un índice pix2 en polytmp utilizando rtree (p poly_ops)
```

Pensé que podría estar relacionado con un número de versión de Postgres anterior e intenté la consulta. Mi sistema se volvió loco y tuve que tirar el postmaster en unos diez minutos.

Voy a contunuar mirando dentro de GiST un rato, pero también agradecería más ejemplos en la utilización de los R-tree.

Notas

1. <http://GiST.CS.Berkeley.EDU:8000/gist/>
2. <http://s2k-ftp.CS.Berkeley.EDU:8000/personal/jmh/>
3. <http://epoch.cs.berkeley.edu:8000/>
4. <mailto:selkovjr@mcs.anl.gov>
5. <ftp://s2k-ftp.cs.berkeley.edu/pub/gist/pggist/pggist.tgz>
6. <http://gist.cs.berkeley.edu:8000/gist/>

Capítulo 40. Enlazando funciones de carga dinámica

Después de crear y registrar una función definida por el usuario, el trabajo está prácticamente terminado. Postgres, sin embargo, debe cargar el fichero de código objeto (e.g., a `.o`, o una biblioteca compartida) que implemente esa función. Como se ha mencionado anteriormente, Postgres carga el código en tiempo de ejecución, a medida que es necesario. A fin de permitir que el código sea cargado dinámicamente, puede tener que compilar y enlazar este código de algún modo especial. Esta sección explica brevemente cómo realizar la compilación y el enlazado necesario antes de que pueda cargar sus funciones en un servidor Postgres en ejecución. Nótese que este proceso ha cambiado respecto al de la versión 4.2.

Debe estar preparado para leer (y releer, y re-leer) las páginas de manual del compilador de C, `cc(1)`, y del enlazador, `ld(1)`, por si necesita información específica. Además, los paquetes de prueba de regresión del directorio `PGROOT/src/regress` contienen varios ejemplos de este proceso. Si comprende lo que realizan estas pruebas, no debería tener ningún problema.

La siguiente terminología se usará más adelante:

- *Carga dinámica (Dynamic loading)* es lo que Postgres hace con un fichero objeto. El fichero objeto se copia en el servidor Postgres en ejecución, y las funciones y variables del fichero quedan disponibles para las funciones de los procesos Postgres. Postgres hace esto usando el mecanismo de carga dinámica proporcionado por el sistema operativo.
- *Configuración de la carga y enlazado (Loading and link editing)* es lo que usted hace con un fichero objeto a fin de producir otro tipo de fichero objeto (por ejemplo, un programa ejecutable o una biblioteca compartida). Esto se realiza por medio del programa de configuración de enlazado, `ld(1)`.

Las siguientes restricciones generales y notas se aplican también al texto siguiente:

- Las rutas dadas a la orden para crear la función deben ser absolutas (es decir, han de empezar con `"/`), y referirse a directorios visibles para la máquina en la que se está ejecutando el servidor Postgres.

Sugerencia: Las rutas relativas también funcionan, pero hay que tener en cuenta que serían relativas al directorio donde reside la base de datos (que es generalmente invisible para las aplicaciones finales). Obviamente, no tiene sentido hacer la ruta relativa al directorio en el que el usuario inicial la aplicación final, dado que el servidor puede estar ejecutándose en una máquina distinta.

- El usuario Postgres debe ser capaz de recorrer la ruta dada a la orden de creación de la función, y ser capaz de leer el fichero objeto. Esto es así porque el servidor Postgres se ejecuta como usuario Postgres, no como el usuario que inicia el proceso final. (Hacer el fichero en el directorio de nivel superior no legible y/o no ejecutable para el usuario "postgres" es un error extremadamente común.)
- Los nombres de símbolos definidos en los ficheros objetos no deben estar en conflicto entre sí, ni con los símbolos definidos en Postgres.
- El compilador de C GNU normalmente no dispone de las opciones especiales necesarias para usar la interfase del cargador dinámico del sistema. En caso de que esto ocurra, ha de usarse el compilador de C que venga con el sistema operativo.

ULTRIX

Es muy fácil escribir ficheros objeto de carga dinámica bajo ULTRIX. ULTRIX no tiene ningún mecanismo para bibliotecas compartidas, y por lo tanto, no plantea restricciones a la interfase del cargador dinámico. Por otra parte, tendremos que (re)escribir un cargador dinámico no portable, y no podremos usar verdaderas bibliotecas compartidas. Bajo ULTRIX, la única restricción es que debe producir cada fichero objeto con la opción -G 0. (Nótese que es trata del número 0, no del literal "o"). Por ejemplo:

```
# simple ULTRIX example
% cc -G 0 -c foo.c
```

produce un fichero objeto llamado foo.o que puede ser cargado dinámicamente en Postgres. No ha de realizarse carga o enlazado adicional.

DEC OSF/1

Bajo DEC OSF/1, puede convertir cualquier fichero objeto en un objeto compartido, ejecutando el comando ld con las adecuadas opciones. La orden es del estilo de:

```
# simple DEC OSF/1 example
% cc -c foo.c
% ld -shared -expect_unresolved '*' -o foo.so foo.o
```

El objeto compartido resultante puede entonces ser cargado en Postgres. Cuando especifique el nombre del fichero objeto para la orden de creación, ha de dar el nombre del fichero objeto compartido (terminando en .so) en lugar de el del fichero objeto normal.

Sugerencia: En realidad, Postgres. no se preocupa del nombre del fichero, mientras sea un fichero objeto compartido. Si prefiere denominar el nombre del fichero compartido con la extensión .o, esto estará bien para Postgres, siempre que se asegure de que se envía el nombre correcto al comando de creación. En otras palabras, ha de ser consistente. Sin embargo, desde un punto de vista práctico, no recomendamos esta práctica, dado que puede acabar confundiéndole respecto a que ficheros han sido convertidos en objetos compartidos, y que ficheros no. Por ejmplo, es muy difícil escribir Makefiles para realizar un enlace automático, si tanto los ficheros objeto, como los objetos compartidos tienen la extensión .o

¡Si el fichero que especifica no es un objeto compartido, la aplicación final se colgará!

SunOS 4.x, Solaris 2.x y HP-UX

Bajo SunOS 4.x, Solaris 2.x y HP-UX, los ficheros objetos pueden crearse compilando el código fuente con parametros especiales del compilador, con lo que se produce una biblioteca compartida. Los pasos necesarios en HP-UX son como sigue. El parametro +z hace que el compilador de C de HP-UX produzca el denominado "Codigo independiente de la posición", ("Position Independent Code", PIC), y el parametro +u elimina algunas restricciones que normalmente son necesarias en la arquitectura

PA-RISC. El fichero objeto ha de convertirse en una biblioteca compartida usando el editor de enlazado de HP-UX, con la opción -b. Todo esto suena complicado, pero en realidad es muy simple, dado que los comandos para hacer todo esto son:

```
# simple HP-UX example
% cc +z +u -c foo.c
% ld -b -o foo.sl foo.o
```

Como los ficheros .so mencionados en la anterior subsección, hay que indicarle a la orden de creación de funciones que fichero es el que hay que cargar (por ejemplo, puede darle la localización de la biblioteca compartida, o fichero .sl). Bajo SunOS 4.x es algo así:

```
# simple SunOS 4.x example
% cc -PIC -c foo.c
% ld -dc -dp -Bdynamic -o foo.so foo.o
```

y bajo Solaris 2.x es:

```
# simple Solaris 2.x example
% cc -K PIC -c foo.c
% ld -G -Bdynamic -o foo.so foo.o
```

O

```
# simple Solaris 2.x example
% gcc -fPIC -c foo.c
% ld -G -Bdynamic -o foo.so foo.o
```

Cuando enlace bibliotecas compartidas, puede tener que especificar bibliotecas compartidas adicionales (normalmente bibliotecas de sistema, como las bibliotecas de C y matemáticas) en la línea de órdenes de ld

Capítulo 41. Triggers (disparadores)

Postgres tiene algunas interfaces cliente como Perl, Tcl, Python y C, así como dos *Lenguajes Procedurales* (PL). También es posible llamar a funciones C como acciones trigger. Notar que los eventos trigger a nivel STATEMENT no están soportados en la versión actual. Actualmente es posible especificar BEFORE o AFTER en los INSERT, DELETE o UPDATE de un registro como un evento trigger.

Creación de Triggers

Si un evento trigger ocurre, el administrador de triggers (llamado Ejecutor) inicializa la estructura global TriggerData *CurrentTriggerData (descrita más abajo) y llama a la función trigger para procesar el evento.

La función trigger debe ser creada antes que el trigger, y debe hacerse como una función sin argumentos, y códigos de retorno opacos.

La sintaxis para la creación de triggers es la siguiente:

```
CREATE TRIGGER <trigger name> <BEFORE|AFTER> <INSERT|DELETE|UPDATE>
ON <relation name> FOR EACH <ROW|STATEMENT>
EXECUTE PROCEDURE <procedure name> (<function args>);
```

El nombre del trigger se usará si se desea eliminar el trigger. Se usa como argumento del comando DROP TRIGGER.

La palabra siguiente determina si la función debe ser llamada antes (BEFORE) o después (AFTER) del evento.

El siguiente elemento del comando determina en que evento/s será llamada la función. Es posible especificar múltiples eventos utilizando el operador OR.

El nombre de la relación (relation name) determinará la tabla afectada por el evento.

La instrucción FOR EACH determina si el trigger se ejecutará para cada fila afectada o bien antes (o después) de que la secuencia se haya completado.

El nombre del procedimiento (procedure name) es la función C llamada.

Los argumentos son pasados a la función en la estructura CurrentTriggerData. El propósito de pasar los argumentos a la función es permitir a triggers diferentes con requisitos similares llamar a la misma función.

Además, la función puede ser utilizada para disparar distintas relaciones (estas funciones son llamadas "general trigger functions").

Como ejemplo de utilización de lo descrito, se puede hacer una función general que toma como argumentos dos nombres de campo e inserta el nombre del usuario y la fecha (timestamp) actuales en ellos. Esto permite, por ejemplo, utilizar los triggers en los eventos INSERT para realizar un seguimiento automático de la creación de registros en una tabla de transacciones. Se podría utilizar también para registrar actualizaciones si es utilizado en un evento UPDATE.

Las funciones trigger retornan un área de tuplas (HeapTuple) al ejecutor. Esto es ignorado para trigger lanzados tras (AFTER) una operación INSERT, DELETE o UPDATE, pero permite lo siguiente a los triggers BEFORE: - retornar NULL e ignorar la operación para la tupla actual (y de este modo la tupla no será insertada/actualizada/borrada); - devolver un puntero a otra tupla (solo en eventos INSERT y UPDATE) que serán insertados (como la nueva versión de la tupla actualizada en caso de UPDATE) en lugar de la tupla original.

Notar que no hay inicialización por parte del CREATE TRIGGER handler. Esto será cambiado en el futuro. Además, si más de un trigger es definido para el mismo evento en la misma relación, el orden de ejecución de los triggers es impredecible. Esto puede ser cambiado en el futuro.

Si una función trigger ejecuta consultas SQL (utilizando SPI) entonces estas funciones pueden disparar nuevos triggers. Esto es conocido como triggers en cascada. No hay ninguna limitación explícita en cuanto al número de niveles de cascada.

Si un trigger es lanzado por un INSERT e inserta una nueva tupla en la misma relación, el trigger será llamado de nuevo (por el nuevo INSERT). Actualmente, no se proporciona ningún mecanismo de sincronización (etc) para estos casos pero esto puede cambiar. Por el momento, existe una función llamada `funny_dup17()` en los tests de regresión que utiliza algunas técnicas para parar la recursividad (cascada) en si misma...

Interacción con el Trigger Manager

Como se ha mencionado, cuando una función es llamada por el administrador de triggers (trigger manager), la estructura `TriggerData *CurrentTriggerData` no es NULL y se inicializa. Por lo cual es mejor verificar que `CurrentTriggerData` no sea NULL al principio y asignar el valor NULL justo después de obtener la información para evitar llamadas a la función trigger que no procedan del administrador de triggers.

La estructura `TriggerData` se define en `src/include/commands/trigger.h`:

```
typedef struct TriggerData
{
    TriggerEvent tg_event;
    Relation tg_relation;
    HeapTuple tg_trigtuple;
    HeapTuple tg_newtuple;
    Trigger *tg_trigger;
} TriggerData;

tg_event
    describe los eventos para los que la función es llamada. Puede utilizar
    las siguientes macros para examinar tg_event:

    TRIGGER_FIRED_BEFORE(event) devuelve TRUE si el trigger se disparó antes;
    TRIGGER_FIRED_AFTER(event) devuelve TRUE si se disparó después;
    TRIGGER_FIRED_FOR_ROW(event) devuelve TRUE si el trigger se dispa-
    ró para un
                                evento a nivel de fila;
    TRIGGER_FIRED_FOR_STATEMENT(event) devuelve TRUE si el trigger se disparó
                                para un evento a nivel de sentencia.
    TRIGGER_FIRED_BY_INSERT(event) devuelve TRUE si fue disparado por un INSERT;
    TRIGGER_FIRED_BY_DELETE(event) devuelve TRUE si fue disparado por un DELETE;
    TRIGGER_FIRED_BY_UPDATE(event) devuelve TRUE si fue disparado por un UPDATE.

tg_relation
    es un puntero a una estructura que describe la relación disparado-
    ra. Mirar
    en src/include/utils/rel.h para ver detalles sobre esta estructura. Lo más
    interesante es tg_relation->rd_att (descriptor de las tuplas de la relación)
    y tg_relation->rd_rel->relname (nombre de la relación. No es un char*, sino
    NameData. Utilizar SPI_getrelname(tg_relation) para obtener char* si se
    necesita una copia del nombre).

tg_trigtuple
```

es un puntero a la tupla por la que es disparado el trigger, esto es, la tupla que se está insertando (en un INSERT), borrando (DELETE) o actualizando (UPDATE).

En caso de un INSERT/DELETE esto es lo que se debe devolver al Ejecutor si no se desea reemplazar la tupla con otra (INSERT) o ignorar la operación.

`tg_newtuple`
es un puntero a la nueva tupla en caso de UPDATE y NULL si es para un INSERT o un DELETE. Esto es lo que debe devolverse al Ejecutor en el caso de un UPDATE si no se desea reemplazar la tupla por otra o ignorar la operación.

`tg_trigger`
es un puntero a la estructura Trigger definida en `src/include/utils/rel.h`:

```
typedef struct Trigger
{
    Oid          tgoid;
    char         *tgname;
    Oid          tgfoid;
    FmgrInfo     tgfunc;
    int16        tgtype;
    bool         tgenabled;
    bool         tgisconstraint;
    bool         tgdeferrable;
    bool         tginitdeferred;
    int16        tgnargs;
    int16        tgattr[FUNC_MAX_ARGS];
    char         **tgargs;
} Trigger;
```

`tgname` es el nombre del trigger, `tgnargs` es el número de argumentos en `tgargs`, `tgargs` es un array de punteros a los argumentos especificados en el CREATE TRIGGER. Otros miembros son exclusivamente para uso interno.

Visibilidad de Cambios en Datos

Regla de visibilidad de cambios en Postgres: durante la ejecución de una consulta, los cambios realizados por ella misma (vía funciones SQL O SPI, o mediante triggers) le son invisibles. Por ejemplo, en la consulta

```
INSERT INTO a SELECT * FROM a
```

las tuplas insertadas son invisibles para el propio SELECT. En efecto, esto duplica la tabla dentro de sí misma (sujeto a las reglas de índice único, por supuesto) sin recursividad.

Pero hay que recordar esto sobre visibilidad en la documentación de SPI:

Los cambios hechos por la consulta Q son visibles por las consultas que empiezan tras la consulta Q, no importa si son iniciados desde Q (durante su ejecución) o una vez ha acabado.

Esto es válido también para los triggers, así mientras se inserta una tupla (`tg_trigtuple`) no es visible a las consultas en un trigger BEFORE, mientras que esta tupla (recién

insertada) es visible a las consultas de un trigger AFTER, y para las consultas en triggers BEFORE/AFTER lanzados con posterioridad!

Ejemplos

Hay ejemplos más complejos en `src/test/regress/regress.c` y en `contrig/spi`.

He aquí un ejemplo muy sencillo sobre el uso de triggers. La función `trigf` devuelve el número de tuplas en la relación `ttest` e ignora la operación si la consulta intenta insertar NULL en `x` (i.e - actúa como una restricción NOT NULL pero no aborta la transacción).

```
#include "executor/spi.h" /* Necesario para trabajar con SPI */
#include "commands/trigger.h" /* -"- y triggers */

HeapTuple trigf(void);

HeapTuple
trigf()
{
    TupleDesc tupdesc;
    HeapTuple rettupple;
    char *when;
    bool checknull = false;
    bool isnull;
    int ret, i;

    if (!CurrentTriggerData)
        elog(WARN, "trigf: triggers sin inicializar");

    /* tupla para devolver al Ejecutor */
    if (TRIGGER_FIRED_BY_UPDATE(CurrentTriggerData->tg_event))
        rettupple = CurrentTriggerData->tg_newtuple;
    else
        rettupple = CurrentTriggerData->tg_trigtuple;

    /* comprobar NULLs ? */
    if (!TRIGGER_FIRED_BY_DELETE(CurrentTriggerData->tg_event) &&
        TRIGGER_FIRED_BEFORE(CurrentTriggerData->tg_event))
        checknull = true;

    if (TRIGGER_FIRED_BEFORE(CurrentTriggerData->tg_event))
        when = "antes ";
    else
        when = "después ";

    tupdesc = CurrentTriggerData->tg_relation->rd_att;
    CurrentTriggerData = NULL;

    /* Conexión al gestor SPI */
    if ((ret = SPI_connect()) < 0)
        elog(WARN, "trigf (lanzado %s): SPI_connect devolvió %d", when, ret);

    /* Obtiene el número de tuplas en la relación */
    ret = SPI_exec("select count(*) from ttest", 0);

    if (ret < 0)
        elog(WARN, "trigf (lanzado %s): SPI_exec devolvió %d", when, ret);

    i = SPI_getbinval(SPI_tuptable->vals[0], SPI_tuptable->tupdesc, 1, &isnull);
```

```

elog (NOTICE, "trigf (lanzado %s): hay %d tuplas en ttest", when, i);

SPI_finish();

if (checknull)
{
i = SPI_getbinval(rettuple, tupdesc, 1, &isnull);
if (isnull)
rettuple = NULL;
}

return (rettuple);
}

```

Ahora, compila y create table ttest (x int4); create function trigf () returns opaque as '...path_to_so' language 'c';

```

vac=> create trigger tbefore before insert or update or delete on ttest
for each row execute procedure trigf();
CREATE
vac=> create trigger tafter after insert or update or delete on ttest
for each row execute procedure trigf();
CREATE
vac=> insert into ttest values (null);
NOTICE:trigf (fired before): there are 0 tuples in ttest
INSERT 0 0

- Insertion skipped and AFTER trigger is not fired

vac=> select * from ttest;
x
-
(0 rows)

vac=> insert into ttest values (1);
NOTICE:trigf (fired before): there are 0 tuples in ttest
NOTICE:trigf (fired after ): there are 1 tuples in ttest
^^^^^^^^^
remember what we said about visibility.
INSERT 167793 1
vac=> select * from ttest;
x
-
1
(1 row)

vac=> insert into ttest select x * 2 from ttest;
NOTICE:trigf (fired before): there are 1 tuples in ttest
NOTICE:trigf (fired after ): there are 2 tuples in ttest
^^^^^^^^^
remember what we said about visibility.
INSERT 167794 1
vac=> select * from ttest;
x
-
1
2
(2 rows)

vac=> update ttest set x = null where x = 2;
NOTICE:trigf (fired before): there are 2 tuples in ttest

```

```

UPDATE 0
vac=> update ttest set x = 4 where x = 2;
NOTICE:trigf (fired before): there are 2 tuples in ttest
NOTICE:trigf (fired after ): there are 2 tuples in ttest
UPDATE 1
vac=> select * from ttest;
x
-
1
4
(2 rows)

vac=> delete from ttest;
NOTICE:trigf (fired before): there are 2 tuples in ttest
NOTICE:trigf (fired after ): there are 1 tuples in ttest
NOTICE:trigf (fired before): there are 1 tuples in ttest
NOTICE:trigf (fired after ): there are 0 tuples in ttest
                                ^^^^^^^^
                                remember what we said about visibility.

DELETE 2
vac=> select * from ttest;
x
-
(0 rows)

```

Nota: Aportación del traductor.Manuel Martínez Valls

En la version 6.4 ya existian los triggers, lo que eran triggers para tuplos, (FOR EACH ROW) pero no para sentencias (FOR STATEMENT), por eso creo que es importante poner disparadores para sentencias, no disparadores solo.

Los trigger son parte de lo que se conoce como "elementos activos" de una BD. Asi como lo son las constraints tales como NOT NULL, FOREIGN KEY, PRIMARY KEY, CHECK. Una vez definidas ellas "se activaran" solo al ocurrir un evento que las viole, un valor nulo en un campo con NOT NULL, etc

¿Por que entonces llamar triggers a los triggers? ;Con ellos se quiso dar mas control al programador sobre los eventos que desencadenan un elemento activo, se le conoce en ingles como ECA rules o event-condition-action rule. Es por ello que los triggers tienen una clausula BEFORE, AFTER o INSTEAD (por cierto pgsq1 no tiene INSTEAD) y bajo que evento (INSERT, UPDATE, DELETE) pero de esta forma el trigger se ejecutara para tuplo (o fila) sometido al evento (clausula FOR EACH ROW) pero el standard (que pgsq1 no cubre completamente) dice que puede ser tambien FOR EACH SENTENCE.

Esto provoca que se ejecute el trigger para toda la relacion (o tabla) para la cual se define (clausula ON). La diferencia para los que lo han programado, por ejemplo en plpgsql, queda clara entonces: cuando es FOR EACH ROW en la funcion pgsq1 que implementa el trigger se tiene un objeto NEW y uno OLD que se refiere a la tupla completa, en el trigger de STATEMENT tiene un objeto NEW y OLD que son la relacion (o tabla) completa

Esta claro entonces que es un poco mas dificil implementar un trigger para statement que para fila (todavia pgsq1 no lo tiene).

Finalmente este es un buen ejemplo de que por que pgsq1 dice que "implementa un subconjunto extendido de SQL92", no hay trigger en SQL92, son del SQL3.

Capítulo 42. Server Programming Interface

The *Server Programming Interface* (SPI) gives users the ability to run SQL queries inside user-defined C functions. The available Procedural Languages (PL) give an alternate means to access these capabilities.

In fact, SPI is just a set of native interface functions to simplify access to the Parser, Planner, Optimizer and Executor. SPI also does some memory management.

To avoid misunderstanding we'll use *function* to mean SPI interface functions and *procedure* for user-defined C-functions using SPI.

SPI procedures are always called by some (upper) Executor and the SPI manager uses the Executor to run your queries. Other procedures may be called by the Executor running queries from your procedure.

Note, that if during execution of a query from a procedure the transaction is aborted then control will not be returned to your procedure. Rather, all work will be rolled back and the server will wait for the next command from the client. This will be changed in future versions.

Other restrictions are the inability to execute BEGIN, END and ABORT (transaction control statements) and cursor operations. This will also be changed in the future.

If successful, SPI functions return a non-negative result (either via a returned integer value or in SPI_result global variable, as described below). On error, a negative or NULL result will be returned.

Interface Functions

SPI_connect

Nombre

SPI_connect — Connects your procedure to the SPI manager.

Synopsis

```
int SPI_connect(void)
```

Inputs

None

Outputs

int

Return status

SPI_OK_CONNECT

if connected

`SPI_ERROR_CONNECT`
if not connected

Description

`SPI_connect` opens a connection to the Postgres backend. You should call this function if you will need to execute queries. Some utility SPI functions may be called from un-connected procedures.

You may get `SPI_ERROR_CONNECT` error if `SPI_connect` is called from an already connected procedure - e.g. if you directly call one procedure from another connected one. Actually, while the child procedure will be able to use SPI, your parent procedure will not be able to continue to use SPI after the child returns (if `SPI_finish` is called by the child). It's bad practice.

Usage

XXX thomas 1997-12-24

Algorithm

`SPI_connect` performs the following:

- Initializes the SPI internal structures for query execution and memory management.

SPI_finish

Nombre

`SPI_finish` — Disconnects your procedure from the SPI manager.

Synopsis

```
SPI_finish(void)
```

Inputs

None

Outputs

int

SPI_OK_FINISH if properly disconnected

SPI_ERROR_UNCONNECTED if called from an un-connected procedure

Description

`SPI_finish` closes an existing connection to the Postgres backend. You should call this function after completing operations through the SPI manager.

You may get the error return `SPI_ERROR_UNCONNECTED` if `SPI_finish` is called without having a current valid connection. There is no fundamental problem with this; it means that nothing was done by the SPI manager.

Usage

`SPI_finish` *must* be called as a final step by a connected procedure or you may get unpredictable results! Note that you can safely skip the call to `SPI_finish` if you abort the transaction (via `elog(ERROR)`).

Algorithm

`SPI_finish` performs the following:

- Disconnects your procedure from the SPI manager and frees all memory allocations made by your procedure via `palloc` since the `SPI_connect`. These allocations can't be used any more! See Memory management.

SPI_exec

Nombre

`SPI_exec` — Creates an execution plan (parser+planner+optimizer) and executes a query.

Synopsis

`SPI_exec(query, tcount)`

Inputs

char **query*

String containing query plan

int *tcount*

Maximum number of tuples to return

Outputs

int

SPI_OK_EXEC if properly disconnected

SPI_ERROR_UNCONNECTED if called from an un-connected procedure

SPI_ERROR_ARGUMENT if query is NULL or *tcount* < 0.

SPI_ERROR_UNCONNECTED if procedure is unconnected.

SPI_ERROR_COPY if COPY TO/FROM stdin.

SPI_ERROR_CURSOR if DECLARE/CLOSE CURSOR, FETCH.

SPI_ERROR_TRANSACTION if BEGIN/ABORT/END.

SPI_ERROR_OPUNKNOWN if type of query is unknown (this shouldn't occur).

If execution of your query was successful then one of the following (non-negative) values will be returned:

SPI_OK_UTILITY if some utility (e.g. CREATE TABLE ...) was executed

SPI_OK_SELECT if SELECT (but not SELECT ... INTO!) was executed

SPI_OK_SELINTO if SELECT ... INTO was executed

SPI_OK_INSERT if INSERT (or INSERT ... SELECT) was executed

SPI_OK_DELETE if DELETE was executed

SPI_OK_UPDATE if UPDATE was executed

Description

`SPI_exec` creates an execution plan (parser+planner+optimizer) and executes the query for *tcount* tuples.

Usage

This should only be called from a connected procedure. If *tcount* is zero then it executes the query for all tuples returned by the query scan. Using *tcount* > 0 you may restrict the number of tuples for which the query will be executed. For example,

```
SPI_exec ("insert into table select * from table", 5);
```

will allow at most 5 tuples to be inserted into table. If execution of your query was successful then a non-negative value will be returned.

Nota: You may pass many queries in one string or query string may be re-written by RULES. `SPI_exec` returns the result for the last query executed.

The actual number of tuples for which the (last) query was executed is returned in the global variable `SPI_processed` (if not `SPI_OK_UTILITY`). If `SPI_OK_SELECT` returned and `SPI_processed > 0` then you may use global pointer `SPITupleTable *SPI_tuptable` to access the selected tuples: Also NOTE, that `SPI_finish` frees and makes all `SPITupleTables` unusable! (See Memory management).

`SPI_exec` may return one of the following (negative) values:

`SPI_ERROR_ARGUMENT` if query is NULL or `tcount < 0`.
`SPI_ERROR_UNCONNECTED` if procedure is unconnected.
`SPI_ERROR_COPY` if COPY TO/FROM stdin.
`SPI_ERROR_CURSOR` if DECLARE/CLOSE CURSOR, FETCH.
`SPI_ERROR_TRANSACTION` if BEGIN/ABORT/END.
`SPI_ERROR_OPUNKNOWN` if type of query is unknown (this shouldn't occur).

Algorithm

`SPI_exec` performs the following:

- Disconnects your procedure from the SPI manager and frees all memory allocations made by your procedure via `palloc` since the `SPI_connect`. These allocations can't be used any more! See Memory management.

SPI_prepare

Nombre

`SPI_prepare` — Connects your procedure to the SPI manager.

Synopsis

```
SPI_prepare(query, nargs, argtypes)
```

Inputs

query

Query string

nargs

Number of input parameters (\$1 ... \$nargs - as in SQL-functions)

argtypes

Pointer list of type OIDs to input arguments

Outputs

void *

Pointer to an execution plan (parser+planner+optimizer)

Description

`SPI_prepare` creates and returns an execution plan (parser+planner+optimizer) but doesn't execute the query. Should only be called from a connected procedure.

Usage

`nargs` is number of parameters (\$1 ... \$`nargs` - as in SQL-functions), and `nargs` may be 0 only if there is not any \$1 in query.

Execution of prepared execution plans is sometimes much faster so this feature may be useful if the same query will be executed many times.

The plan returned by `SPI_prepare` may be used only in current invocation of the procedure since `SPI_finish` frees memory allocated for a plan. See `SPI_saveplan`.

If successful, a non-null pointer will be returned. Otherwise, you'll get a NULL plan. In both cases `SPI_result` will be set like the value returned by `SPI_exec`, except that it is set to `SPI_ERROR_ARGUMENT` if query is NULL or `nargs` < 0 or `nargs` > 0 && `argtypes` is NULL.

SPI_saveplan**Nombre**

`SPI_saveplan` — Saves a passed plan

Synopsis

```
SPI_saveplan(plan)
```

Inputsvoid **query*

Passed plan

Outputs

`void *`

Execution plan location. NULL if unsuccessful.

`SPI_result`

`SPI_ERROR_ARGUMENT` if plan is NULL

`SPI_ERROR_UNCONNECTED` if procedure is un-connected

Description

`SPI_saveplan` stores a plan prepared by `SPI_prepare` in safe memory protected from freeing by `SPI_finish` or the transaction manager.

In the current version of Postgres there is no ability to store prepared plans in the system catalog and fetch them from there for execution. This will be implemented in future versions. As an alternative, there is the ability to reuse prepared plans in the consequent invocations of your procedure in the current session. Use `SPI_execp` to execute this saved plan.

Usage

`SPI_saveplan` saves a passed plan (prepared by `SPI_prepare`) in memory protected from freeing by `SPI_finish` and by the transaction manager and returns a pointer to the saved plan. You may save the pointer returned in a local variable. Always check if this pointer is NULL or not either when preparing a plan or using an already prepared plan in `SPI_execp` (see below).

Nota: If one of the objects (a relation, function, etc.) referenced by the prepared plan is dropped during your session (by your backend or another process) then the results of `SPI_execp` for this plan will be unpredictable.

SPI_execp

Nombre

`SPI_execp` — Executes a plan from `SPI_saveplan`

Synopsis

```
SPI_execp(plan,
values,
nulls,
```

tcount)

Inputs

void **plan*

Execution plan

Datum **values*

Actual parameter values

char **nulls*

Array describing what parameters get NULLs

'n' indicates NULL allowed

'' indicates NULL not allowed

int *tcount*

Number of tuples for which plan is to be executed

Outputs

int

Returns the same value as *SPI_exec* as well as

SPI_ERROR_ARGUMENT if *plan* is NULL or *tcount* < 0

SPI_ERROR_PARAM if *values* is NULL and *plan* was prepared with some parameters.

SPI_tuptable

initialized as in *SPI_exec* if successful

SPI_processed

initialized as in *SPI_exec* if successful

Description

SPI_execp stores a plan prepared by *SPI_prepare* in safe memory protected from freeing by *SPI_finish* or the transaction manager.

In the current version of Postgres there is no ability to store prepared plans in the system catalog and fetch them from there for execution. This will be implemented in future versions. As a work around, there is the ability to reuse prepared plans in the consequent invocations of your procedure in the current session. Use *SPI_execp* to execute this saved plan.

Usage

If *nulls* is NULL then *SPI_execp* assumes that all values (if any) are NOT NULL.

Nota: If one of the objects (a relation, function, etc.) referenced by the prepared plan is dropped during your session (by your backend or another process) then the results of

`SPI_execp` for this plan will be unpredictable.

Interface Support Functions

All functions described below may be used by connected and unconnected procedures.

SPI_copytuple

Nombre

`SPI_copytuple` — Makes copy of tuple in upper Executor context

Synopsis

```
SPI_copytuple(tuple)
```

Inputs

HeapTuple *tuple*

Input tuple to be copied

Outputs

HeapTuple

Copied tuple

non-NULL if *tuple* is not NULL and the copy was successful

NULL only if *tuple* is NULL

Description

`SPI_copytuple` makes a copy of tuple in upper Executor context. See the section on Memory Management.

Usage

TBD

SPI_modifytuple

Nombre

`SPI_modifytuple` — Modifies tuple of relation

Synopsis

```
SPI_modifytuple(rel, tuple , nattrs
, attnum , Values , Nulls)
```

Inputs

Relation *rel*

HeapTuple *tuple*

Input tuple to be modified

int *nattrs*

Number of attribute numbers in *attnum*

int * *attnum*

Array of numbers of the attributes which are to be changed

Datum * *Values*

New values for the attributes specified

char * *Nulls*

Which attributes are NULL, if any

Outputs

HeapTuple

New tuple with modifications

non-NULL if *tuple* is not NULL and the modify was successful

NULL only if *tuple* is NULL

SPI_result

SPI_ERROR_ARGUMENT if *rel* is NULL or *tuple* is NULL or *natts* ≤ 0 or *attnum* is NULL or *Values* is NULL

SPI_ERROR_NOATTRIBUTE if there is an invalid attribute number in *attnum* (*attnum* ≤ 0 or $> \text{number of attributes}$)

Description

`SPI_modifytuple` Modifies a tuple in upper Executor context. See the section on

Memory Management.

Usage

If successful, a pointer to the new tuple is returned. The new tuple is allocated in upper Executor context (see Memory management). Passed tuple is not changed.

SPI_fnumber

Nombre

`SPI_fnumber` — Finds the attribute number for specified attribute

Synopsis

```
SPI_fnumber(tupdesc, fname)
```

Inputs

TupleDesc *tupdesc*
 Input tuple description
 char * *fname*
 Field name

Outputs

int
 Attribute number
 Valid one-based index number of attribute
 SPI_ERROR_NOATTRIBUTE if the named attribute is not found

Description

`SPI_fnumber` returns the attribute number for the attribute with name in *fname*.

Usage

Attribute numbers are 1 based.

SPI_fname

Nombre

`SPI_fname` — Finds the attribute name for the specified attribute

Synopsis

```
SPI_fname(tupdesc, fname)
```

Inputs

`TupleDesc tupdesc`
Input tuple description

`char * fnumber`
Attribute number

Outputs

`char *`
Attribute name
NULL if *fnumber* is out of range
SPI_result set to `SPI_ERROR_NOATTRIBUTE` on error

Description

`SPI_fname` returns the attribute name for the specified attribute.

Usage

Attribute numbers are 1 based.

Algorithm

Returns a newly-allocated copy of the attribute name.

SPI_getvalue

Nombre

`SPI_getvalue` — Returns the string value of the specified attribute

Synopsis

```
SPI_getvalue(tuple, tupdesc, fnumber)
```

Inputs

HeapTuple *tuple*

Input tuple to be examined

TupleDesc *tupdesc*

Input tuple description

int *fnumber*

Attribute number

Outputs

char *

Attribute value or NULL if
attribute is NULL

fnumber is out of range (SPI_result set to SPI_ERROR_NOATTRIBUTE)

no output function available (SPI_result set to SPI_ERROR_NOOUTFUNC)

Description

`SPI_getvalue` returns an external (string) representation of the value of the specified attribute.

Usage

Attribute numbers are 1 based.

Algorithm

Allocates memory as required by the value.

SPI_getbinval

Nombre

`SPI_getbinval` — Returns the binary value of the specified attribute

Synopsis

```
SPI_getbinval(tuple, tupdesc, fnumber, isnull)
```

Inputs

HeapTuple *tuple*

Input tuple to be examined

TupleDesc *tupdesc*

Input tuple description

int *fnumber*

Attribute number

Outputs

Datum

Attribute binary value

bool * *isnull*

flag for null value in attribute

SPI_result

SPI_ERROR_NOATTRIBUTE

Description

`SPI_getbinval` returns the binary value of the specified attribute.

Usage

Attribute numbers are 1 based.

Algorithm

Does not allocate new space for the binary value.

SPI_gettype

Nombre

`SPI_gettype` — Returns the type name of the specified attribute

Synopsis

```
SPI_gettype(tupdesc, fnumber)
```

Inputs

TupleDesc *tupdesc*

Input tuple description

int *fnumber*

Attribute number

Outputs

char *

The type name for the specified attribute number

SPI_result

SPI_ERROR_NOATTRIBUTE

Description

`SPI_gettype` returns a copy of the type name for the specified attribute.

Usage

Attribute numbers are 1 based.

Algorithm

Does not allocate new space for the binary value.

SPI_gettypeid

Nombre

`SPI_gettypeid` — Returns the type OID of the specified attribute

Synopsis

```
SPI_gettypeid(tupdesc, fnumber)
```

Inputs

TupleDesc *tupdesc*

Input tuple description

int *fnumber*

Attribute number

Outputs

OID

The type OID for the specified attribute number

SPI_result

SPI_ERROR_NOATTRIBUTE

Description

`SPI_gettypeid` returns the type OID for the specified attribute.

Usage

Attribute numbers are 1 based.

Algorithm

TBD

SPI_getrelname

Nombre

`SPI_getrelname` — Returns the name of the specified relation

Synopsis

```
SPI_getrelname(rel)
```

Inputs

Relation *rel*
Input relation

Outputs

`char *`
The name of the specified relation

Description

`SPI_getrelname` returns the name of the specified relation.

Usage

TBD

Algorithm

Copies the relation name into new storage.

SPI_palloc

Nombre

`SPI_palloc` — Allocates memory in upper Executor context

Synopsis

```
SPI_palloc(size)
```

Inputs

Size *size*

Octet size of storage to allocate

Outputs

void *

New storage space of specified size

Description

`SPI_palloc` allocates memory in upper Executor context. See section on memory management.

Usage

TBD

SPI_repalloc**Nombre**

`SPI_repalloc` — Re-allocates memory in upper Executor context

Synopsis

```
SPI_repalloc(pointer, size)
```

Inputs

void **pointer*

Pointer to existing storage

Size *size*

Octet size of storage to allocate

Outputs

`void *`

New storage space of specified size with contents copied from existing area

Description

`SPI_realloc` re-allocates memory in upper Executor context. See section on memory management.

Usage

TBD

SPI_pfree**Nombre**

`SPI_pfree` — Frees memory from upper Executor context

Synopsis

`SPI_pfree(pointer)`

Inputs

`void *pointer`

Pointer to existing storage

Outputs

None

Description

`SPI_pfree` frees memory in upper Executor context. See section on memory management.

Usage

TBD

Memory Management

Server allocates memory in memory contexts in such way that allocations made in one context may be freed by context destruction without affecting allocations made in other contexts. All allocations (via `palloc`, etc) are made in the context which are chosen as current one. You'll get unpredictable results if you'll try to free (or reallocate) memory allocated not in current context.

Creation and switching between memory contexts are subject of SPI manager memory management.

SPI procedures deal with two memory contexts: upper Executor memory context and procedure memory context (if connected).

Before a procedure is connected to the SPI manager, current memory context is upper Executor context so all allocation made by the procedure itself via `palloc`/`repalloc` or by SPI utility functions before connecting to SPI are made in this context.

After `SPI_connect` is called current context is the procedure's one. All allocations made via `palloc`/`repalloc` or by SPI utility functions (except for `SPI_copytuple`, `SPI_modifytuple`, `SPI_palloc` and `SPI_repalloc`) are made in this context.

When a procedure disconnects from the SPI manager (via `SPI_finish`) the current context is restored to the upper Executor context and all allocations made in the procedure memory context are freed and can't be used any more!

If you want to return something to the upper Executor then you have to allocate memory for this in the upper context!

SPI has no ability to automatically free allocations in the upper Executor context!

SPI automatically frees memory allocated during execution of a query when this query is done!

Visibility of Data Changes

Postgres data changes visibility rule: during a query execution, data changes made by the query itself (via SQL-function, SPI-function, triggers) are invisible to the query scan. For example, in query `INSERT INTO a SELECT * FROM a` tuples inserted are invisible for `SELECT`' scan. In effect, this duplicates the database table within itself (subject to unique index rules, of course) without recursing.

Changes made by query `Q` are visible by queries which are started after query `Q`, no matter whether they are started inside `Q` (during the execution of `Q`) or after `Q` is done.

Examples

This example of SPI usage demonstrates the visibility rule. There are more complex examples in `src/test/regress/regress.c` and in `contrib/spi`.

This is a very simple example of SPI usage. The procedure `execq` accepts an SQL-query in its first argument and `tcount` in its second, executes the query using `SPI_exec` and returns the number of tuples for which the query executed:

```
#include "executor/spi.h" /* this is what you need to work with SPI */

int execq(text *sql, int cnt);

int
execq(text *sql, int cnt)
{
    int ret;
    int proc = 0;

    SPI_connect();

    ret = SPI_exec(textout(sql), cnt);

    proc = SPI_processed;
    /*
     * If this is SELECT and some tuple(s) fetched -
     * returns tuples to the caller via elog (NOTICE).
     */
    if ( ret == SPI_OK_SELECT && SPI_processed > 0 )
    {
        TupleDesc tupdesc = SPI_tuptable->tupdesc;
        SPITupleTable *tuptable = SPI_tuptable;
        char buf[8192];
        int i;

        for (ret = 0; ret < proc; ret++)
        {
            HeapTuple tuple = tuptable->vals[ret];

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                sprintf(buf + strlen (buf), " %s%s",
                    SPI_getvalue(tuple, tupdesc, i),
                    (i == tupdesc->natts) ? " " : " |");
            elog (NOTICE, "EXECQ: %s", buf);
        }

        SPI_finish();

        return (proc);
    }
}
```

Now, compile and create the function:

```
create function execq (text, int4) returns int4 as '...path_to_so' lan-
guage 'c';

vac=> select execq('create table a (x int4)', 0);
execq
---
      0
(1 row)

vac=> insert into a values (execq('insert into a values (0)',0));
INSERT 167631 1
vac=> select execq('select * from a',0);
```

```

NOTICE:EXECQ:  0 << inserted by execq

NOTICE:EXECQ:  1 << value returned by execq and inserted by upper INSERT

execq
---
      2
(1 row)

vac=> select execq('insert into a select x + 2 from a',1);
execq
---
      1
(1 row)

vac=> select execq('select * from a', 10);
NOTICE:EXECQ:  0

NOTICE:EXECQ:  1

NOTICE:EXECQ:  2 << 0 + 2, only one tuple inserted - as specified

execq
---
      3          << 10 is max value only, 3 is real # of tuples
(1 row)

vac=> delete from a;
DELETE 3
vac=> insert into a values (execq('select * from a', 0) + 1);
INSERT 167712 1
vac=> select * from a;
x
-
1          << no tuples in a (0) + 1
(1 row)

vac=> insert into a values (execq('select * from a', 0) + 1);
NOTICE:EXECQ:  0
INSERT 167713 1
vac=> select * from a;
x
-
1
2          << there was single tuple in a + 1
(2 rows)

-   This demonstrates data changes visibility rule:

vac=> insert into a select execq('select * from a', 0) * x from a;
NOTICE:EXECQ:  1
NOTICE:EXECQ:  2
NOTICE:EXECQ:  1
NOTICE:EXECQ:  2
NOTICE:EXECQ:  2
INSERT 0 2
vac=> select * from a;
x
-
1
2
2          << 2 tuples * 1 (x in first tuple)
6          << 3 tuples (2 + 1 just inserted) * 2 (x in second tuple)

```

```
(4 rows)          ^^^^^^^^  
                  tuples visible to execq() in different invocations
```


Capítulo 43. Lenguajes Procedurales

A partir del lanzamiento de la versión 6.3, Postgres soporta la definición de lenguajes procedurales. En el caso de una función o procedimiento definido en un lenguaje procedural, la base de datos no tiene un conocimiento implícito sobre como interpretar el código fuente de las funciones. El manejador en sí es una función de un lenguaje de programación compilada en forma de objeto compartido, y cargado cuando es necesario.

Instalación de lenguajes procedurales

Instalación de lenguajes procedurales

Un lenguaje procedural se instala en la base de datos en tres pasos.

1. El objeto compartido que contienen el manejador del lenguaje ha de ser compilado e instalado. Por defecto, el manejador para PL/pgSQL está integrado e instalado en el directorio de bibliotecas de la base de datos. Si el soporte de Tcl/Tk está instalado y configurado, el manejador para PL/Tcl está integrado e instalado en el mismo sitio.

La escritura de un nuevo lenguaje procedural (Procedural language, PL) está mas allá del ámbito de este manual.

2. El manejador debe ser declarado mediante la orden

```
CREATE FUNCTION handler_function_name () RETURNS OPAQUE AS
    'path-to-shared-object' LANGUAGE 'C';
```

El calificador especial de tipo devuelto OPAQUE le dice a la base de datos que esta función no devuelve uno de los tipos definidos en la base de datos ni un tipo compuesto, y que no es directamente utilizable en una sentencia SQL.

3. El PL debe ser declarado con la orden

```
CREATE [ TRUSTED ] PROCEDURAL LANGUAGE 'language-name'
    HANDLER handler_function_name
    LANCMPILER 'description';
```

La palabra clave opcional TRUSTED indica si un usuario normal de la base de datos, sin privilegios de superusuario, puede usar este lenguaje para crear funciones y procedimientos activadores. Dado que las funciones de los PL se ejecutan dentro de la aplicación de base de datos, sólo deberían usarse para lenguajes que no puedan conseguir acceso a las aplicaciones internas de la base de datos, o al sistema de ficheros. Los lenguajes PL/pgSQL y PL/Tcl son manifiestamente fiables en este sentido

Ejemplo

1. La siguiente orden le dice a la base de datos donde encontrar el objeto compartido para el manejador de funciones que llama al lenguaje PL/pgSQL

```
CREATE FUNCTION plpgsql_call_handler () RETURNS OPAQUE AS
    '/usr/local/pgsql/lib/plpgsql.so' LANGUAGE 'C';
```

2. La orden

```
CREATE TRUSTED PROCEDURAL LANGUAGE 'plpgsql'
    HANDLER plpgsql_call_handler
    LANCMPILER 'PL/pgSQL';
```

define que la función manejadora de llamadas previamente declarada debe ser invocada por las funciones y procedimientos disparadores cuando el atributo del lenguaje es 'plpgsql'

Las funciones manejadoras de PL tienen una interfase de llamadas especial distinta del de las funciones de lenguaje C normales. Uno de los argumentos dados al manejador es el identificador del objeto en las entradas de la tabla `pg_proc` para la función que ha de ser ejecutada. El manejador examina varios catálogos de sistema para analizar los argumentos de llamada de la función y los tipos de dato que devuelve. El texto fuente del cuerpo de la función se encuentra en el atributo `prosrc` de `pg_proc`. Debido a esto, en contraste con las funciones de lenguaje C, las funciones PL pueden ser sobrecargadas, como las funciones del lenguaje SQL. Puede haber múltiples funciones PL con el mismo nombre de función, siempre que los argumentos de llamada sean distintos.

Los lenguajes procedurales definidos en la base de datos `template1` se definen automáticamente en todas las bases de datos creadas subsecuentemente. Así que el administrador de la base de datos puede decidir que lenguajes están definidos por defecto.

PL/pgSQL

PL/pgSQL es un lenguaje procedural cargable para el sistema de bases de datos Postgres.

Este paquete fue escrito originalmente por Jan Wieck.

Panorámica

Los objetivos de diseño de PL/pgSQL fueron crear un lenguaje procedural cargable que

- pueda usarse para crear funciones y procedimientos disparados por eventos,
- añada estructuras de control al lenguaje SQL,
- pueda realizar cálculos complejos,
- herede todos los tipos definidos por el usuario, las funciones y los operadores,
- pueda ser definido para ser fiable para el servidor,
- sea fácil de usar,

El gestor de llamadas PL/pgSQL analiza el texto de las funciones y produce un árbol de instrucciones binarias interno la primera vez que la función es invocada por una aplicación. El bytecode producido es identificado por el manejador de llamadas mediante el ID de la función. Esto asegura que el cambio de una función por parte de una secuencia DROP/CREATE tendrá efecto sin tener que establecer una nueva conexión con la base de datos.

Para todas y las expresiones y sentencias SQL usadas en la función, el interprete de bytecode de PL/pgSQL crea un plan de ejecución preparado usando los gestores de SPI, funciones `SPI_prepare()` y `SPI_saveplan()`. Esto se hace la primera vez que las sentencias individuales se procesan en la función PL/pgSQL. Así, una función con código condicional que contenga varias sentencias que puedan ser ejecutadas, solo

preparará y almacenará las opciones que realmente se usarán durante el ámbito de la conexión con la base de datos.

Excepto en el caso de funciones de conversión de entrada/salida y de cálculo para tipos definidos, cualquier cosa que pueda definirse en funciones de lenguaje C puede ser hecho con PL/pgSQL. Es posible crear funciones complejas de calculo y después usarlas para definir operadores o usarlas en índices funcionales.

Descripcion

Estructura de PL/pgSQL

El lenguaje PL/pgSQL no es sensible a las mayúsculas. Todas las palabras clave e identificadores pueden usarse en cualquier mezcla de mayúsculas y minúsculas.

PL/pgSQL es un lenguaje orientado a bloques. Un bloque se define como

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END;
```

Puede haber cualquier numero de subbloques en la sección de sentencia de un bloque. Los subbloques pueden usarse para ocultar variables a otros bloques de sentencias. Las variables declaradas en la sección de declaraciones se inicializan a su valor por defecto cada vez que se inicia el bloque, no cada vez que se realiza la llamada a la función.

Es importante no confundir el significado de BEGIN/END en la agrupación de sentencias de OL/pgSQL y las ordenes de la base de datos para control de transacciones. Las funciones y procedimientos disparadores no pueden iniciar o realizar transacciones y Postgres no soporta transacciones anidadas.

Comments

Hay dos tipos de comentarios en PL/pgSQL. Un par de guiones '-' comienza un comentario que se extiende hasta el fin de la línea. Los caracteres '/*' comienzan un bloque de comentarios que se extiende hasta que se encuentre un '*/'. Los bloques de comentarios no pueden anidarse pero un par de guiones pueden encerrarse en un bloque de comentario, o ocultar los limitadores de estos bloques.

Declaraciones

Todas las variables, filas y columnas que se usen en un bloque o subbloque ha de ser declarado en la sección de declaraciones del bloque, excepto las variables de control de bucle en un bucle FOR que se itere en un rango de enteros. Los parámetros dados a una función PL/pgSQL se declaran automáticamente con los identificadores usuales, \$n. Las declaraciones tienen la siguiente sintaxis:

```
name [ CONSTANT ] >typ> [ NOT NULL ] [ DEFAULT | := value ];
```

Esto declara una variable de un tipo base especificado. Si la variable es declarada como CONSTANT, su valor no podrá ser cambiado. Si se especifica NOT NULL,

la asignación de un NULL producirá un error en tiempo de ejecución. Dado que el valor por defecto de todas las variables es el valor NULL de SQL, todas las variables declaradas como NOT NULL han de tener un valor por defecto.

El valor por defecto es evaluado cada vez que se invoca la función. Así que asignar 'now' a una variable de tipo *datetime* hace que la variable tome el momento de la llamada a la función, no el momento en que la función fue compilada a bytecode.

name class%ROWTYPE;

Esto declara una fila con la estructura de la clase indicada. La clase ha de ser una tabla existente, o la vista de una base de datos. Se accede a los campos de la fila mediante la notación de punto. Los parámetros de una función pueden ser de tipos compuestos (filas de una tabla completas). En ese caso, el correspondiente identificador \$n será un tipo de fila, pero ha de ser referido usando la orden ALIAS que se describe más adelante. Solo los atributos de usuario de una fila de tabla son accesibles en la fila, no se puede acceder a Oid o a los otros atributos de sistema (dado que la fila puede ser de una vista, y las filas de una vista no tienen atributos de sistema útiles).

Los campos de un tipo de fila heredan los tipos de datos, tamaños y precisiones de las tablas.

name RECORD;

Los registros son similares a los tipos de fila, pero no tienen una estructura predefinida. Se emplean en selecciones y bucles FOR, para mantener una fila de la actual base de datos en una operación SELECT. El mismo registro puede ser usado en diferentes selecciones. El acceso a un campo de registro cuando no hay una fila seleccionada resultará en un error de ejecución.

Las filas NEW y OLD en un disparador se pasan a los procedimientos como registros. Esto es necesario porque en Postgres un mismo procedimiento desencadenado puede tener sucesos disparadores en diferentes tablas.

name ALIAS FOR \$n;

Para una mejor legibilidad del código, es posible definir un alias para un parámetro posicional de una función.

Estos alias son necesarios cuando un tipo compuesto se pasa como argumento a una función. La notación punto \$1.salary como en funciones SQL no se permiten en PL/pgSQL

RENAME *oldname* TO *newname*;

Esto cambia el nombre de una variable, registro o fila. Esto es útil si NEW o OLD ha de ser referenciado por parte de otro nombre dentro de un procedimiento desencadenado.

Tipos de datos

Los tipos de una variable pueden ser cualquiera de los tipos básicos existentes en la base de datos. *type* en la sección de declaraciones se define como:

- `Postgres-basetype`
- `variable%TYPE`
- `class.field%TYPE`

variable es el nombre de una variable, previamente declarada en la misma función, que es visible en este momento.

class es el nombre de una tabla existente o vista, donde *field* es el nombre de un atributo.

El uso de `class.field%TYPE` hace que PL/pgSQL busque las definiciones de atributos en la primera llamada a la función, durante toda la vida de la aplicación final. Supongamos que tenemos una tabla con un atributo `char(20)` y algunas funciones PL/pgSQL, que procesan el contenido por medio de variables locales. Ahora, alguien decide que `char(20)` no es suficiente, cierra la tabla, y la recrea con el atributo en cuestión definido como `char(40)`, tras lo que restaura los datos. Pero se ha olvidado de las funciones. Los cálculos internos de éstas truncarán los valores a 20 caracteres. Pero si hubieran sido definidos usando las declaraciones `class.field%TYPE` automáticamente se adaptarán al cambio de tamaño, o a si el nuevo esquema de la tabla define el atributo como de tipo texto.

Expressions

Todas las expresiones en las sentencias PL/pgSQL son procesadas usando backends de ejecución. Las expresiones que puedan contener constantes pueden de hecho requerir evaluación en tiempo de ejecución (por ejemplo, `'now'` para el tipo `'datetime'`), dado que es imposible para el analizador de PL/pgSQL identificar los valores constantes distintos de la palabra clave `NULL`. Todas las expresiones se evalúan internamente ejecutando una consulta

```
SELECT expression
```

usando el gestor SPI. En la expresión, las apariciones de los identificadores de variables son sustituidos por parámetros, y los valores reales de las variables son pasadas al ejecutor en la matriz de parámetros. Todas las expresiones usadas en una función PL/pgSQL son preparadas de una sola vez, y guardadas una única vez.

La comprobación de tipos hecha por el analizador principal de Postgres tiene algunos efectos secundarios en la interpretación de los valores constantes. En detalle, hay una diferencia entre lo que hacen estas dos funciones

```
CREATE FUNCTION logfunc1 (text) RETURNS datetime AS '
DECLARE
    logtxt ALIAS FOR $1;
BEGIN
    INSERT INTO logtable VALUES (logtxt, "now");
    RETURN "now";
END;
' LANGUAGE 'plpgsql';
```

y

```
CREATE FUNCTION logfunc2 (text) RETURNS datetime AS '
DECLARE
    logtxt ALIAS FOR $1;
    curtime datetime;
BEGIN
    curtime := "now";
    INSERT INTO logtable VALUES (logtxt, curtime);
    RETURN curtime;
END;
' LANGUAGE 'plpgsql';
```

En el caso de `logfunc1()`, el analizador principal de Postgres sabe cuando prepara la ejecución de `INSERT` que la cadena `'now'` debe ser interpretada como una fecha, dado que el campo objeto de `'logtable'` tiene ese tipo. Así, hará una constante de ese tipo, y el valor de esa constante se empleará en todas las llamadas a `logfunc1()`, durante toda la vida útil de ese proceso. No hay que decir que eso no era lo que pretendía el programador.

En el caso de `logfunc2()`, el analizador principal de Postgres no sabe cual es el tipo de `'now'`, por lo que devuelve un tipo de texto, que contiene la cadena `'now'`. Durante la asignación a la variable local `'curtime'`, el interprete PL/pgSQL asigna a esta cadena el tipo fecha, llamando a las funciones `text_out()` y `datetime_in()` para realizar la conversión.

esta comprobación de tipos realizada por el analizador principal de Postgres fue implementado antes de que PL/pgSQL estuviera totalmente terminado. Es una diferencia entre 6.3 y 6.4, y afecta a todas las funciones que usan la planificación realizada por el gestor SPI. El uso de variables locales en la manera descrita anteriormente es actualmente la única forma de que PL/pgSQL interprete esos valores correctamente.

Si los campos del registro son usados en expresiones o sentencias, los tipos de datos de campos no deben cambiarse entre llamadas de una misma expresión. Tenga esto en cuenta cuando escriba procedimientos disparadores que gestionen eventos en más de una tabla.

Sentencias

Cualquier cosa no comprendida por el analizador PL/pgSQL tal como se ha especificado será enviado al gestor de bases de datos, para su ejecución. La consulta resultante no devolverá ningún dato.

Asignación

Una asignación de un valor a una variable o campo de fila o de registro se escribe:

```
identifier := expression;
```

Si el tipo de dato resultante de la expresión no coincide con el tipo de dato de las variables, o la variable tienen un tamaño o precisión conocido (como `char(29)`), el resultado será amoldado implícitamente por el interprete de bytecode de PL/pgSQL, usando los tipos de las variables para las funciones de entrada y los tipos resultantes en las funciones de salida. Nótese que esto puede potencialmente producir errores de ejecución generados por los tipos de las funciones de entrada.

Una asignación de una selección completa en un registro o fila puede hacerse del siguiente modo:

```
SELECT expressions INTO target FROM ...;
```

target puede ser un registro, una variable de fila o una lista separada por comas de variables y campo de registros o filas.

Si una fila o una lista de variables se usa como objetivo, los valores seleccionados han de coincidir exactamente con la estructura de los objetivos o se producirá un error de ejecución. La palabra clave FROM puede preceder a cualquier calificador válido, agrupación, ordenación, etc. que pueda pasarse a una sentencia SELECT.

Existe una variable especial llamada FOUND de tipo booleano, que puede usarse inmediatamente después de SELECT INTO para comprobar si una asignación ha tenido éxito.

```
SELECT * INTO myrec FROM EMP WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION "employee % not found", myname;
END IF;
```

Si la selección devuelve múltiples filas, solo la primera se mueve a los campos objetivo. todas las demás se descartan.

Llamadas a otra función

Todas las funciones definidas en una base de datos Postgres devuelven un valor. Por lo tanto, la forma normal de llamar a una función es ejecutar una consulta SELECT o realizar una asignación (que de lugar a un SELECT interno de PL/pgSQL). Pero hay casos en que no interesa saber los resultados de las funciones.

```
PERFORM query
```

Esto ejecuta 'SELECT *query*' en el gestor SPI, y descarta el resultado. Los identificadores como variables locales son de todos modos sustituidos en los parámetros.

Volviendo de la función

```
RETURN expression
```

La función termina y el valor de *expression* se devolverá al ejecutor superior. El valor devuelto por una función no puede quedar sin definir. Si el control alcanza el fin del bloque de mayor nivel de la función sin encontrar una sentencia RETURN, ocurrirá un error de ejecución.

Las expresiones resultantes serán amoldadas automáticamente en los tipos devueltos por la función, tal como se ha descrito en el caso de las asignaciones.

Abortando la ejecución y mensajes

Como se ha indicado en los ejemplos anteriores, hay una sentencia RAISE que puede enviar mensajes al sistema de registro de Postgres. #####

ATENCION WARNING ACHTUNG ##### ¡Aquí puede haber una errata! Comparad con el original

```
    RAISE level
  for" [,
    identifier [...]];
```

Dentro del formato, "%" se usa como situación para los subsecuentes identificadores, separados por comas. Los posibles niveles son DEBUG (suprimido en las bases de datos de producción), NOTICE (escribe en el registro de la base de datos y lo envía a la aplicación del cliente) y EXCEPTION (escribe en el registro de la base de datos y aborta la transacción).

Condiciones

```
IF expression THEN
    statements
[ELSE
    statements]
END IF;
```

expression debe devolver un valor que al menos pueda ser adaptado en un tipo booleano.

Bucles

Hay varios tipos de bucles.

```
[<<label>>]
LOOP
    statements
END LOOP;
```

Se trata de un bucle no condicional que ha de ser terminado de forma explícita, mediante una sentencia EXIT. La etiqueta opcional puede ser usado por las sentencias EXIT de otros bucles anidados, para especificar el nivel del bucle que ha de terminarse.

```
[<<label>>]
WHILE expression LOOP
    statements
END LOOP;
```

Se trata de un lazo condicional que se ejecuta mientras la evaluación de *expression* sea cierta.

```
[<<label>>]
FOR name IN [ REVERSE ]
express .. expression LOOP
    statements
END LOOP;
```

Se trata de un bucle que se itera sobre un rango de valores enteros. La variable *name* se crea automáticamente con el tipo entero, y existe solo dentro del bucle. Las dos expresiones dan el límite inferior y superior del rango y son evaluados sólo cuando se entra en el bucle. El paso de la iteración es siempre 1.

```
[<<label>>]
```

```
FOR record / row IN select_clause LOOP
    statements
END LOOP;
```

EL registro o fila se asigna a todas las filas resultantes de la clausula de selección, y la sentencia se ejecuta para cada una de ellas. Si el bucle se termina con una sentencia EXIT, la ultima fila asignada es aún accesible después del bucle.

```
EXIT [ label ] [ WHEN expression ];
```

Si no se incluye *label*, se termina el lazo más interno, y se ejecuta la sentencia que sigue a END LOOP. Si se incluye *label* ha de ser la etiqueta del bucle actual u de otro de mayor nivel. EL bucle indicado se termina, y el control se pasa a la sentencia de después del END del bucle o bloque correspondiente.

Procedimientos desencadenados

PL/pgSQL puede ser usado para definir procedimientos desencadenados por eventos. Estos se crean con la orden CREATE FUNCTION, igual que una función, pero sin argumentos, y devuelven un tipo OPAQUE.

Hay algunos detalles específicos de Postgres cuando se usan funciones como procedimientos desencadenados.

En primer lugar, disponen de algunas variables especiales que se crean automáticamente en los bloques de mayor nivel de la sección de declaración. Son:

NEW

Tipo de dato RECORD; es una variable que mantienen la fila de la nueva base de datos en las operaciones INSERT/UPDATE, en los desencadenados ROW.

OLD

Tipo de dato RECORD; es una variable que mantiene la fila de la base de datos vieja en operaciones UPDATE/DELETE, en los desencadenados ROW.

TG_NAME

Nombre de tipo de dato; es una variable que contiene el nombre del procedimiento desencadenado que se ha activado.

TG_WHEN

Tipo de dato texto; es una cadena de caracteres del tipo 'BEFORE' o 'AFTER', dependiendo de la definición del procedimiento desencadenado.

TG_LEVEL

Tipo de dato texto; una cadena de 'ROW' o 'STATEMENT', dependiendo de la definición del procedimiento desencadenado.

TG_OP

Tipo de dato texto; una cadena de 'INSERT', 'UPDATE' o 'DELETE', que nos dice la operación para la que se ha disparado el procedimiento desencadenado.

TG_RELID

Tipo de dato oid; el ID del objeto de la tabla que ha provocado la invocación del procedimiento desencadenado.

TG_RELNAME

Tipo de dato nombre; el nombre de la tabla que ha provocado la activación del procedimiento desencadenado.

TG_NARGS

Tipo de dato entero; el numero de argumentos dado al procedimiento desencadenado en la sentencia CREATE TRIGGER.

TG_ARGV[]

Tipo de dato matriz de texto; los argumentos de la sentencia CREATE TRIGGER. El índice comienza por cero, y puede ser dado en forma de expresión. Índices no validos dan lugar a un valor NULL.

En segundo lugar, han de devolver o NULL o una fila o registro que contenga exactamente la estructura de la tabla que ha provocado la activación del procedimiento desencadenado. Los procedimientos desencadenados activados por AFTER deben devolver siempre un valor NULL, sin producir ningún efecto. Los procedimientos desencadenados activados por BEFORE indican al gestor de procedimientos desencadenados que no realice la operación sobre la fila actual cuando se devuelva NULL. En cualquier otro caso, la fila o registro devuelta sustituye a la fila insertada o actualizada. Es posible reemplazar valores individuales directamente en una sentencia NEW y devolverlos, o construir una nueva fila o registro y devolverla.

Excepciones

Postgres no dispone de un modelo de manejo de excepciones muy elaborado. Cuando el analizador, el optimizador o el ejecutor deciden que una sentencia no puede ser procesada, la transacción completa es abortada y el sistema vuelve al lazo principal para procesar la siguiente consulta de la aplicación cliente.

Es posible introducirse en el mecanismo de errores para detectar cuando sucede esto. Pero lo que no es posible es saber qué ha causado en realidad el aborto (un error de conversión de entrada/salida, un error de punto flotante, un error de análisis). Y es posible que la base de datos haya quedado en un estado inconsistente, por lo que volver a un nivel de ejecución superior o continuar ejecutando comandos puede corromper toda la base de datos. E incluso aunque se pudiera enviar la información a la aplicación cliente, la transacción ya se habría abortado, por lo que carecería de sentido el intentar reanudar la operación.

Por todo esto, lo único que hace PL/pgSQL cuando se produce un aborto de ejecución durante la ejecución de una función o procedimiento disparador es enviar mensajes de depuración al nivel DEBUG, indicando en qué función y donde (numero de línea y tipo de sentencia) ha sucedido el error.

Ejemplos

Se incluyen unas pocas funciones para demostrar lo fácil que es escribir funciones en PL/pgSQL. Para ejemplos más complejos, el programador debería consultar el test de regresión de PL/pgSQL.

Un detalle doloroso a la hora de escribir funciones en PL/pgSQL es el manejo de la comilla simple. El texto de las funciones en CREATE FUNCTION ha de ser una cadena de texto. Las comillas simples en el interior de una cadena literal deben duplicarse o anteponerse de una barra invertida. Aún estamos trabajando en una

alternativa más elegante. Mientras tanto, duplique las comillas sencillas como en los ejemplos siguientes. Cualquier solución a este problema en futuras versiones de Postgres mantendrán la compatibilidad con esto.

Algunas funciones sencillas en PL/pgSQL

Las dos funciones siguientes son idénticas a sus contrapartidas que se verán cuando estudiemos el lenguaje C.

```
CREATE FUNCTION add_one (int4) RETURNS int4 AS '
BEGIN
    RETURN $1 + 1;
END;
' LANGUAGE 'plpgsql';

CREATE FUNCTION concat_text (text, text) RETURNS text AS '
BEGIN
    RETURN $1 || $2;
END;
' LANGUAGE 'plpgsql';
```

Funciones PL/pgSQL para tipos compuestos

De nuevo, estas funciones PL/pgSQL tendrán su equivalente en lenguaje C.

```
CREATE FUNCTION c_overpaid (EMP, int4) RETURNS bool AS '
DECLARE
    emprec ALIAS FOR $1;
    sallim ALIAS FOR $2;
BEGIN
    IF emprec.salary ISNULL THEN
        RETURN "f";
    END IF;
    RETURN emprec.salary > sallim;
END;
' LANGUAGE 'plpgsql';
```

Procedimientos desencadenados en PL/pgSQL

Estos procedimientos desencadenados aseguran que, cada vez que se inserte o actualice un fila en la tabla, se incluya el nombre del usuario y la fecha y hora. Y asegura que se proporciona un nombre de empleado y que el salario tiene un valor positivo.

```
CREATE TABLE emp (
    empname text,
    salary int4,
    last_date datetime,
    last_user name);

CREATE FUNCTION emp_stamp () RETURNS OPAQUE AS
BEGIN
```

```

- Check that empname and salary are given
IF NEW.empname ISNULL THEN
    RAISE EXCEPTION "empname cannot be NULL value";
END IF;
IF NEW.salary ISNULL THEN
    RAISE EXCEPTION "% cannot have NULL salary", NEW.empname;
END IF;

- Who works for us when she must pay for?
IF NEW.salary < 0 THEN
    RAISE EXCEPTION "% cannot have a negative salary", NEW.empname;
END IF;

- Remember who changed the payroll when
NEW.last_date := "now";
NEW.last_user := getpgusername();
RETURN NEW;
END;
' LANGUAGE 'plpgsql';

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
FOR EACH ROW EXECUTE PROCEDURE emp_stamp();

```

PL/Tcl

PL/Tcl es un lenguaje procedural para el gestor de bases de datos Postgres que permite el uso de Tcl para la creación de funciones y procedimientos desencadenados por eventos.

Este paquete fue escrito originalmente por Jan Wieck.

Introducción

PL/Tcl ofrece la mayoría de las capacidades de que dispone el lenguaje C, excepto algunas restricciones.

Las restricciones buenas son que todo se ejecuta en un buen interprete Tcl. Además del reducido juego de ordenes de Tcl, solo se disponen de unas pocas ordenes para acceder a bases de datos a través de SPI y para enviar mensajes mediante `elog()`. No hay forma de acceder a las interioridades del proceso de gestión de la base de datos, no de obtener acceso al nivel del sistema operativo, bajo los permisos del identificador de usuario de Postgres, como es posible en C. Así, cualquier usuario de bases de datos sin privilegios puede usar este lenguaje.

La otra restricción, interna, es que los procedimientos Tcl no pueden usarse para crear funciones de entrada / salida para nuevos tipos de datos.

Los objetos compartidos para el gestor de llamada PL/Tcl se construyen automáticamente y se instalan en el directorio de bibliotecas de Postgres, si el soporte de Tcl/Tk ha sido especificado durante la configuración, en el procedimiento de instalación.

Descripción

Funciones de Postgres y nombres de procedimientos Tcl

En Postgres, un mismo nombre de función puede usarse para diferentes funciones, siempre que el número de argumentos o sus tipos sean distintos. Esto puede ocasionar conflictos con los nombres de procedimientos Tcl. Para ofrecer la misma flexibilidad en PL/Tcl, los nombres de procedimientos Tcl internos contienen el identificador de objeto de la fila de procedimientos `pg_proc` como parte de sus nombres. Así, diferentes versiones (por el número de argumentos) de una misma función de Postgres pueden ser diferentes también para Tcl.

Definiendo funciones en PL/Tcl

Para crear una función en el lenguaje PL/Tcl, se usa la sintaxis

```
CREATE FUNCTION funcname
argumen) RETURNS
    returntype AS '
    # PL/Tcl function body
    ' LANGUAGE 'pltcl';
```

Cuando se invoca esta función en una consulta, los argumentos se dan como variables `$1 ... $n` en el cuerpo del procedimiento Tcl. Así, una función de máximo que devuelva el mayor de dos valores `int4` sería creada del siguiente modo:

```
CREATE FUNCTION tcl_max (int4, int4) RETURNS int4 AS '
    if {$1 > $2} {return $1}
return $2
    ' LANGUAGE 'pltcl';
```

Argumentos de tipo compuesto se pasan al procedimiento como matrices de Tcl. Los nombres de elementos en la matriz son los nombres de los atributos del tipo compuesto. ¡Si un atributo de la fila actual tiene el valor NULL, no aparecerá en la matriz! He aquí un ejemplo que define la función `overpaid_2` (que se encuentra en la antigua documentación de Postgres), escrita en PL/Tcl

```
CREATE FUNCTION overpaid_2 (EMP) RETURNS bool AS '
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
        return "t"
    }
    return "f"
    ' LANGUAGE 'pltcl';
```

Datos Globales en PL/Tcl

A veces (especialmente cuando se usan las funciones SPI que se describirán más adelante) es útil tener algunos datos globales que se mantengan entre dos llamadas al

procedimiento. Todos los procedimientos PL/Tcl ejecutados por un backend comparten el mismo intérprete de Tcl. Para ayudar a proteger a los procedimientos PL/Tcl de efectos secundarios, una matriz queda disponible para cada uno de los procedimientos a través de la orden 'upvar'. El nombre global de esa variable es el nombre interno asignado por el procedimiento, y el nombre local es GD.

Procedimientos desencadenados en PL/Tcl

Los procedimientos desencadenados se definen en Postgres como funciones sin argumento y que devuelven un tipo opaco. Y lo mismo en el lenguaje PL/Tcl.

La información del gestor de procedimientos desencadenados se pasan al cuerpo del procedimiento en las siguientes variables:

\$TG_name

El nombre del procedimiento disparador se toma de la sentencia CREATE TRIGGER.

\$TG_relid

El ID de objeto de la tabla que provoca el desencadenamiento ha de ser invocado.

\$TG_relatts

Una lista Tcl de los nombres de campos de las tablas, precedida de un elemento de lista vacío. Esto se hace para que al buscar un nombre de elemento en la lista con la orden de Tcl 'lsearch', se devuelva el mismo número positivo, comenzando por 1, en el que los campos están numerados en el catálogo de sistema 'pg_attribute'.

\$TG_when

La cadena BEFORE o AFTER, dependiendo del suceso de la llamada desencadenante.

\$TG_level

La cadena ROW o STATEMENT, dependiendo del suceso de la llamada desencadenante.

\$TG_op

La cadena INSERT, UPDATE o DELETE, dependiendo del suceso de la llamada desencadenante.

\$NEW

Una matriz que contiene los valores de la fila de la nueva tabla para acciones INSERT/UPDATE, o vacía para DELETE.

\$OLD

Una matriz que contiene los valores de la fila de la vieja tabla para acciones UPDATE o DELETE, o vacía para INSERT.

\$GD

La matriz de datos de estado global, como se describa más adelante.

\$args

Una lista Tcl de los argumentos del procedimiento como se dan en la sentencia CREATE TRIGGER. Los argumentos son también accesibles como \$1 ... \$n en el cuerpo del procedimiento.

EL valor devuelto por un procedimiento desencadenado es una de las cadenas OK o SKIP, o una lista devuelta por la orden Tcl 'array get'. Si el valor devuelto es OK, la operación normal que ha desencadenado el procedimiento (INSERT/UPDATE/DELETE) tendrá lugar. Obviamente, SKIP le dice al gestor de procesos desencadenados que suprima silenciosamente la operación. La lista de 'array get' le dice a PL/Tcl que devuelva una fila modificada al gestor de procedimientos desencadenados que será insertada en lugar de la dada en \$NEW (solo para INSERT/UPDATE). No hay que decir que todo esto solo tiene sentido cuando el desencadenante es BEFORE y FOR EACH ROW.

Ha aquí un pequeño ejemplo de procedimiento desencadenado que fuerza a un valor entero de una tabla a seguir la pista del numero de actualizaciones que se han realizado en esa fila. Para cada nueva fila insertada, el valor es inicializado a 0, e incrementada en cada operación de actualización:

```
CREATE FUNCTION trigfunc_modcount() RETURNS OPAQUE AS '
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
        default {
            return OK
        }
    }
    return [array get NEW]
' LANGUAGE 'pltcl';

CREATE TABLE mytab (num int4, modcnt int4, desc text);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

Acceso a bases de datos desde PL/Tcl

Las siguientes ordenes permiten acceder a una base de datos desde el interior de un procedimiento PL/Tcl:

elog level msg

Lanza un mensaje de registro. Los posibles niveles son NOTICE, WARN, ERROR, FATAL, DEBUG y NOIND, como en la función 'elog()' de C.

quote string

Duplica todas las apariciones de una comilla o de la barra invertida. Debería usarse cuando las variables se usen en la cadena en la cadena de la consulta

enviada a 'spi_exec' o 'spi_prepara' (no en la lista de valores de 'spi_execp'). Consideremos una cadena de consulta como esta:

```
"SELECT '$val' AS ret"
```

Donde la variable Tcl 'val' contiene "doesn't". Esto da lugar a la cadena de consulta

```
"SELECT 'doesn't' AS ret"
```

que produce un error del analizador durante la ejecución de 'spi_exec' o 'spi_prepara'. Debería contener

```
"SELECT 'doesn"t' AS ret"
```

y ha de escribirse de la siguiente manera

```
"SELECT '[ quote $val ]' AS ret"
```

spi_exec ?-count n? ?-array nam?que ?loop-body?

Llama al analizador/planificador/optimizador/ejecutor de la consulta. El valor opcional -count la dice a 'spi_exec' el máximo número de filas que han de ser procesadas por la consulta.

Si la consulta es una sentencia SELECT y se incluye el cuerpo del lazo opcional (un cuerpo de sentencias Tcl similar a una sentencia anticipada), se evalúa para cada fila seleccionada, y se comporta como se espera, tras continua/break. Los valores de los campos seleccionados se colocan en nombres de variables, como nombres de columnas. Así,

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

pondrá en la variable cnt el número de filas en el catálogo de sistema 'pg_proc'. Si se incluye la opción -array, los valores de columna son almacenados en la matriz asociativa llamada 'name', indexada por el nombre de la columna, en lugar de en variables individuales.

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "have table $C(relname)"
}
```

imprimirá un mensaje de registro DEBUG para cada una de las filas de pg_class. El valor devuelto por spi_exec es el número de filas afectado por la consulta, y se encuentra en la variable global SPI_processed.

spi_prepare query typelist

Prepara Y GUARDA una consulta para una ejecución posterior. Es un poco distinto del caso de C, ya que en ese caso, la consulta prevista es automáticamente copiada en el contexto de memoria de mayor nivel. Por lo tanto, no actualmente ninguna forma de planificar una consulta sin guardarla.

Si la consulta hace referencia a argumentos, los nombres de los tipos han de incluirse, en forma de lista Tcl. El valor devuelto por 'spi_prepare' es el identificador de la consulta que se usará en las siguientes llamadas a 'spi_execp'. Véase 'spi_execp' para un ejemplo.

`spi_exec ?-count n? ?-array nam? ?-nullsesquvalue? ?loop-body?`

Ejecuta una consulta preparada en 'spi_prepare' con sustitución de variables. El valor opcional '-count' le dice a 'spi_execp' el máximo número de filas que se procesarán en la consulta.

El valor opcional para '-nulls' es una cadena de espacios de longitud "n", que le indica a 'spi_execp' qué valores son NULL. Si se indica, debe tener exactamente la longitud del número de valores.

El identificador de la consulta es el identificador devuelto por la llamada a 'spi_prepare'.

Si se pasa una lista de tipos a 'spi_prepare', ha de pasarse una lista Tcl de exactamente la misma longitud a 'spi_execp' después de la consulta. Si la lista de tipos de 'spi_prepare' está vacía, este argumento puede omitirse.

Si la consulta es una sentencia SELECT, lo que se ha descrito para 'spi_exec' ocurrirá para el cuerpo del bucle y las variables de los campos seleccionados.

He aquí un ejemplo de una función PL/Tcl que usa una consulta planificada:

```
CREATE FUNCTION t1_count(int4, int4) RETURNS int4 AS '
    if {[ info exists GD(plan) ]} {
        # prepare the saved plan on the first call
        set GD(plan) [ spi_prepare \
            "SELECT count(*) AS cnt FROM t1 WHERE num >= \\$1 AND num <= \\
            int4 ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
' LANGUAGE 'pltcl';
```

Nótese que cada una de las barras invertidas que Tcl debe ver ha de ser doblada en la consulta que crea la función, dado que el analizador principal procesa estas barras en CREATE FUNCTION. Dentro de la cadena de la consulta que se pasa a 'spi_prepare' debe haber un signo \$ para marcar la posición del parámetro, y evitar que \$1 sea sustituido por el valor dado en la primera llamada a la función.

Módulos y la orden 'desconocido'

PL/Tcl tiene una característica especial para cosas que suceden raramente. Reconoce dos tablas "mágicas", 'pltcl_modules' y 'pltcl_modfuncs'. Si existen, el módulo 'desconocido' es cargado por el interprete, inmediatamente tras su creación. Cada vez que se invoca un procedimiento Tcl desconocido, el procedimiento 'desconocido' es comprobado, por si el procedimiento en cuestión está definido en uno de esos módulos. Si ocurre esto, el módulo es cargado cuando sea necesario. Para habilitar este comportamiento, el gestor de llamadas de PL/Tcl ha de ser compilado con la opción -DPLTCL_UNKNOWN_SUPPORT habilitado.

Existen scripts de soporte para mantener esas tablas en el subdirectorio de módulos del código fuente de PL/Tcl, incluyendo el código fuente del módulo 'desconocido', que ha de ser instalado inicialmente.

Capítulo 44. Funciones

Información referente a funciones llamadas por usuario.

Nota: Esta sección necesita ser escrita. ¿Voluntarios?

Capítulo 45. Objetos Grandes

En Postgres, los valores de los datos se almacenan en tuplas y las tuplas individuales no pueden abarcar varias páginas de datos. Como el tamaño de una página de datos es de 8192 bytes, el límite máximo del tamaño de un valor de un dato es relativamente pequeño. Para soportar el almacenamiento de valores atómicos más grandes, Postgres proporciona una interfaz para objetos grandes. Esta interfaz proporciona un acceso orientado a archivos para aquellos datos del usuario que han sido declarados como de tipo grande. Esta sección describe la implementación y las interfaces del lenguaje de consulta y programación para los datos de objetos grandes en Postgres.

Nota Histórica

Originalmente, Postgres 4.2 soportaba tres implementaciones estándar de objetos grandes: como archivos externos a Postgres, como archivos externos controlados por Postgres, y como datos almacenados dentro de la base de datos Postgres. Esto causaba gran confusión entre los usuarios. Como resultado, sólo se soportan objetos grandes como datos almacenados dentro de la base de datos Postgres en PostgreSQL. Aún cuando es más lento el acceso, proporciona una integridad de datos más estricta. Por razones históricas, a este esquema de almacenamiento se lo denomina Objetos grandes invertidos. (Utilizaremos en esta sección los términos objetos grandes invertidos y objetos grandes en forma alternada refiriéndonos a la misma cosa.)

Características de la Implementación

La implementación de objetos grandes invertidos separa los objetos grandes en "trozos" y almacena los trozos en tuplas de la base de datos. Un índice B-tree garantiza búsquedas rápidas del número de trozo correcto cuando se realizan accesos de lectura y escritura aleatorios.

Interfaces

Las herramientas que Postgres proporciona para acceder a los objetos grandes, tanto en el backend como parte de funciones definidas por el usuario como en el frontend como parte de una aplicación que utiliza la interfaz, se describen más abajo. Para los usuarios familiarizados con Postgres 4.2, PostgreSQL tiene un nuevo conjunto de funciones que proporcionan una interfaz más coherente.

Nota: Toda manipulación de objetos grandes *debe* ocurrir dentro de una transacción SQL. Este requerimiento es obligatorio a partir de Postgres v6.5, a pesar que en versiones anteriores era un requerimiento implícito, e ignorarlo resultará en un comportamiento impredecible.

La interfaz de objetos grandes en Postgres está diseñada en forma parecida a la interfaz del sistema de archivos de Unix, con funciones análogas como `open(2)`, `read(2)`, `write(2)`, `lseek(2)`, etc. Las funciones de usuario llaman a estas rutinas para obtener sólo los datos de interés de un objeto grande. Por ejemplo, si existe un tipo de objeto grande llamado `foto_sorpresa` que almacena fotografías de caras, entonces puede definirse una función llamada `barba` sobre los datos de `foto_sorpresa`. `Barba` puede mirar el tercio inferior de una fotografía, y determinar el color de la barba que apa-

rece, si es que hubiera. El contenido total del objeto grande no necesita ser puesto en un búfer, ni siquiera examinado por la función `barba`. Los objetos grandes pueden ser accedidos desde funciones C cargadas dinámicamente o programas clientes de bases de datos enlazados con la librería. Postgres proporciona un conjunto de rutinas que soportan la apertura, lectura, escritura, cierre y posicionamiento en objetos grandes.

Creando un Objeto Grande

La rutina

```
Oid lo_creat(PGconn *conexion, int modo)
```

crea un nuevo objeto grande. *modo* es una máscara de bits que describe distintos atributos del nuevo objeto. Las constantes simbólicas listadas aquí se encuentran definidas en `$PGROOT/src/backend/libpq/libpq-fs.h`. El tipo de acceso (lectura, escritura, o ambos) se controla efectuando una operación OR entre los bits `INV_READ` (lectura) e `INV_WRITE` (escritura). Si el objeto grande debe archivarse – es decir, si versiones históricas del mismo deben moverse periódicamente a una tabla de archivo especial – entonces el bit `INV_ARCHIVE` debe utilizarse. Los dieciséis bits de orden bajo de la máscara constituyen el número de manejador de almacenamiento donde debe residir el objeto grande. Para otros sitios que no sean Berkeley, estos bits deberán estar siempre en cero. Los comandos indicados más abajo crean un objeto grande (invertido):

```
inv_oid = lo_creat(INV_READ | INV_WRITE | INV_ARCHIVE);
```

Importando un Objeto Grande

Para importar un archivo de UNIX como un objeto grande, puede llamar a la función

```
Oid lo_import(PGconn *conexion, const char *nombre_de_archivo)
```

nombre_de_archivo especifica la ruta y el nombre del archivo Unix que será importado como objeto grande.

Exportando un Objeto Grande

Para exportar un objeto grande dentro de un archivo de UNIX, puede llamar a la función

```
int lo_export(PGconn *conexion, Oid lobjId, const char *nombre_de_archivo)
```

El argumento *lobjId* especifica el Oid del objeto grande a exportar y el argumento *nombre_de_archivo* indica la ruta y nombre del archivo UNIX.

Abriendo un Objeto Grande Existente

Para abrir un objeto grande existente, llame a la función

```
int lo_open(PGconn *conexion, Oid lobjId, int modo)
```

El argumento *lobjId* especifica el Oid del objeto grande que se abrirá. Los bits de *modo* controlan si el objeto se abre para lectura (INV_READ), escritura o ambos. Un objeto grande no puede abrirse antes de crearse. *lo_open* devuelve un descriptor de objeto grande para su uso posterior en *lo_read*, *lo_write*, *lo_lseek*, *lo_tell*, y *lo_close*.

Escribiendo Datos en un Objeto Grande

La rutina

```
int lo_write(PGconn *conexion, int fd, const char *buf, size_t largo)
```

escribe *largo* bytes desde *buf* al objeto grande *fd*. El argumento *fd* debió ser previamente devuelto por una llamada a *lo_open*. Devuelve el número de bytes escritos efectivamente. En caso de error, el valor de retorno es negativo.

Leyendo Datos desde un Objeto Grande

La rutina

```
int lo_read(PGconn *conexion, int fd, char *buf, size_t largo)
```

lee *largo* bytes desde el objeto grande *fd* a *buf*. El argumento *fd* debió ser previamente devuelto por una llamada a *lo_open*. Devuelve el número de bytes leídos efectivamente. En caso de error, el valor de retorno es negativo.

Posicionándose en un Objeto Grande

Para cambiar la ubicación actual de lectura o escritura en un objeto grande, utilice la función

```
int lo_lseek(PGconn *conexion, int fd, int desplazamiento, int desde_donde)
```

Esta rutina mueve el puntero de posición actual para el objeto grande descrito por *fd* a la nueva ubicación especificada por el *desplazamiento*. Los valores válidos para *desde_donde* son SEEK_SET, SEEK_CUR, y SEEK_END.

Cerrando un Descriptor de Objeto Grande

Un objeto grande puede cerrarse llamando a

```
int lo_close(PGconn *conexion, int fd)
```

donde *fd* es un descriptor de objeto grande devuelto por *lo_open*. Si hay éxito, *lo_close* devuelve cero. Si hay un error, el valor devuelto es negativo.

Funciones registradas Incorporadas

Existen dos funciones registradas incorporadas, `lo_import` y `lo_export` que son convenientes para el uso en consultas SQL. Aquí hay un ejemplo de su uso

```
CREATE TABLE imagen (
    nombre          text,
    contenido        oid
);

INSERT INTO imagen (nombre, contenido)
VALUES ('imagen hermosa', lo_import('/etc/motd'));

SELECT lo_export(imagen.contenido, "/tmp/motd") from imagen
WHERE nombre = 'imagen hermosa';
```

Accediendo a Objetos Grandes desde LIBPQ

Debajo se encuentra un programa de ejemplo que muestra cómo puede utilizarse la interfaz de objetos grandes de LIBPQ. Partes del programa están comentadas pero se dejan en la fuente para el beneficio de los lectores. Este programa puede encontrarse en `../src/test/examples`. Las aplicaciones que utilicen la interfaz de objetos grandes en LIBPQ deben incluir el archivo de cabecera `libpq/libpq-fs.h` y enlazarse con la librería `libpq`.

Programa de Ejemplo

```
/*-----
 *
 * testlo.c-
 * prueba utilizando objetos grandes con libpq
 *
 * Copyright (c) 1994, Regents of the University of California
 *
 *
 * IDENTIFICATION
 * /usr/local/devel/pglite/cvs/src/doc/manual.me,v 1.16 1995/09/01 23:55:00
 *
 *-----
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE      1024

/*
 * importFile *      importar el archivo "filename" en la base de da-
tos como el objeto grande "lobjOid"
 *
 */
Oid importFile(PGconn *conn, char *filename)
{
    Oid lobjId;
    int lobj_fd;
```

```

char buf[BUFSIZE];
int nbytes, tmp;
int fd;

/*
 * abrir el archivo a leer
 */
fd = open(filename, O_RDONLY, 0666);
if (fd < 0) { /* error */
    fprintf(stderr, "no se pudo abrir el archivo unix %s\n", filename);
}

/*
 * crear el objeto grande
 */
lobjId = lo_creat(conn, INV_READ|INV_WRITE);
if (lobjId == 0) {
    fprintf(stderr, "no se pudo crear el objeto grande\n");
}

lobj_fd = lo_open(conn, lobjId, INV_WRITE);
/*
 * Leer desde el archivo Unix y escribir al archivo invertido
 */
while ((nbytes = read(fd, buf, BUFSIZE)) > 0) {
    tmp = lo_write(conn, lobj_fd, buf, nbytes);
    if (tmp < nbytes) {
        fprintf(stderr, "error al escribir el objeto grande\n");
    }
}

(void) close(fd);
(void) lo_close(conn, lobj_fd);

return lobjId;
}

void pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int lobj_fd;
    char* buf;
    int nbytes;
    int nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0) {
        fprintf(stderr, "no se pudo abrir el objeto grande %d\n",
            lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len+1);

    nread = 0;
    while (len - nread > 0) {
        nbytes = lo_read(conn, lobj_fd, buf, len - nread);
        buf[nbytes] = ' ';
        fprintf(stderr, ">>> %s", buf);
        nread += nbytes;
    }
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

```

```

void overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int lobj_fd;
    char* buf;
    int nbytes;
    int nwritten;
    int i;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0) {
        fprintf(stderr, "no se pudo abrir el objeto grande %d\n",
            lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len+1);

    for (i=0; i<len; i++)
        buf[i] = 'X';
    buf[i] = ' ';

    nwritten = 0;
    while (len - nwritten > 0) {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
        nwritten += nbytes;
    }
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

/*
 * exportFile *      exportar el objeto grande "lobjOid" al ar-
chivo "filename"
 */
void exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int lobj_fd;
    char buf[BUFSIZE];
    int nbytes, tmp;
    int fd;

    /*
     * create an inversion "object"
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0) {
        fprintf(stderr, "no se pudo abrir el objeto grande %d\n",
            lobjId);
    }

    /*
     * open the file to be written to
     */
    fd = open(filename, O_CREAT|O_WRONLY, 0666);
    if (fd < 0) { /* error */
        fprintf(stderr, "no se pudo abrir el archivo unix %s\n",
            filename);
    }

    /*
     * leer desde el archivo invertido y escribir al archivo Unix

```



```

    */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0) {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes) {
            fprintf(stderr, "error al escribir %s\n",
                    filename);
        }
    }

    (void) lo_close(conn, lobj_fd);
    (void) close(fd);

    return;
}

void
exit_nicely(PGconn* conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    char *in_filename, *out_filename;
    char *database;
    Oid lobjOid;
    PGconn *conn;
    PGresult *res;

    if (argc != 4) {
        fprintf(stderr, "Utilización: %s database_name in_filename out_filename\n",
                argv[0]);
        exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * set up the connection
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* check to see that the backend connection was success-
fully made */
    if (PQstatus(conn) == CONNECTION_BAD) {
        fprintf(stderr, "Falló la conexión con la base de datos '%s'.\n", database);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    res = PQexec(conn, "begin");
    PQclear(res);

    printf("importando archivo %s\n", in_filename);
    /* lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
    /*
    printf("como objeto grande %d.\n", lobjOid);

```

```

        printf("extrayendo los bytes 1000-2000 del objeto grande\n");
        pickout(conn, lobjOid, 1000, 1000);

        printf("sobreescribiendo los bytes 1000-2000 del objeto gran-
de con X's\n");
        overwrite(conn, lobjOid, 1000, 1000);
    */

    printf("exportando el objeto grande al archivo %s\n", out_filename);
    /*      exportFile(conn, lobjOid, out_filename); */
    lo_export(conn, lobjOid, out_filename);

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    exit(0);
}

```

I. CCVS API Functions

Estas funciones unen el CCVS API y permiten directamente trabajar con CCVS en sus scripts de PHP. CCVS es la solución "intermediaria" de RedHat¹ en el proceso de tarjetas de crédito. Permite realizar un proceso directo en las cámaras de compensación de tarjetas de crédito por medio de su paquete *nix y un modem. Usando el módulo de CCVS para PHP, usted puede procesar tarjetas del crédito directamente a través de CCVS vía sus scripts de PHP. Las referencias siguientes perfilarán el proceso.

Para habilitar el soporte CCVS en PHP, primeramente verifique el directorio de instalación de CCVS. Luego necesitará configurar PHP con la opción `-with-ccvs`. Si usa esta opción sin especificar la ruta (path) al su instalación de CCVS, PHP buscará en la ubicación por defecto de CCVS (`/usr/local/ccvs`). Si se encuentra en una ubicación 'no común', debe ejecutarlo con la opción: `-with-ccvs=$ccvs_path`, donde `$ccvs_path` es la ruta a su instalación de CCVS. Recuerde que CCVS requiere que las librerías `$ccvs_path/lib` y `$ccvs_path/include` existan, conteniendo `cv_api.h` en la carpeta `include` y `libccvs.a` en el directorio `lib`.

Adicionalmente puede necesitarse correr un proceso `ccvsd` para realizar la configuración dentro de sus scripts en PHP. Será necesario también que pueda verificar que los procesos en PHP estén corriendo sobre el mismo usuario que tiene instalado su CCVS (ejem. Si ha instalado CCVS como un usuario 'ccvs', su proceso PHP debe correr dentro de 'ccvs' también.)

Se puede encontrar información adicional sobre CCVS en <http://www.redhat.com/products/software/ecommerce/ccvs/>

Esta sección de la documentación está trabajándose recientemente. Hasta ahora, RedHat mantiene documentación útil ligeramente anticuada pero inmóvil a <http://www.redhat.com/apps/support/>

Notas

1. <http://www.redhat.com>
2. <http://www.redhat.com/products/software/ecommerce/ccvs/>
3. <http://www.redhat.com/apps/support/>

Nombre

—

() ;

Capítulo 46. libpq

`libpq` es la interfaz para los programadores de aplicaciones en C para PostgreSQL. `libpq` es un conjunto de rutinas de biblioteca que permiten a los programas cliente trasladar consultas al servidor de Postgres y recibir el resultado de esas consultas. `libpq` es también el mecanismo subyacente para muchas otras interfaces de aplicaciones de PostgreSQL, incluyendo `libpq++` (C++), `libpqtc1` (Tcl), Perl, y `ecpg`. Algunos aspectos del comportamiento de `libpq` le resultarán de importancia si quiere utilizar uno de estos paquetes.

Se incluyen tres programas cortos al final de esta sección para mostrarle como escribir programas que utilicen `libpq`. Hay varios ejemplos completos de aplicaciones con `libpq` en los siguientes directorios:

```
../src/test/regress
../src/test/examples
../src/bin/psql
```

Los programas cliente que utilicen `libpq` deberán incluir el fichero de cabeceras `libpq-fe.h`, y deberán enlazarse con la biblioteca `libpq`.

Funciones de Conexión a la Base de Datos

Las siguientes rutinas le permitirán realizar una conexión al servidor de Postgres. El programa de aplicación puede tener abiertas varias conexiones a servidores al mismo tiempo. (Una razón para hacer esto es acceder a más de una base de datos). Cada conexión se representa por un objeto `PGconn` que se obtiene de `PQconnectdb()` o `PQsetdbLogin()`. Nótese que estas funciones siempre devolverán un puntero a un objeto no nulo, a menos que se tenga demasiada poca memoria incluso para crear el objeto `PGconn`. Se debería llamar a la función `PQstatus` para comprobar si la conexión se ha realizado con éxito antes de enviar consultas a través del objeto de conexión.

- `PQconnectdb` Realiza una nueva conexión al servidor de base de datos.

```
PGconn *PQconnectdb(const char *conninfo)
```

Esta rutina abre una conexión a una base de datos utilizando los parámetros que se dan en la cadena `conninfo`. Contra lo que ocurre más abajo con `PQsetdbLogin()`, los parámetros fijados se pueden extender sin cambiar la firma de la función, de modo que el uso de bien esta rutina o bien las análogas sin bloqueo `PQconnectStart` / `PQconnectPoll` resulta preferible para la programación de las aplicaciones. La cadena pasada puede estar variá para utilizar así los parámetros de defecto, o puede contener uno o más parámetros separados por espacios.

Cada fijación de un parámetro tiene la forma `keyword = value`. (Para escribir un valor nulo o un valor que contiene espacios, se emplearán comillas simples, por ejemplo `keyword = 'a value'`. Las comillas simples dentro de un valor se escribirán como `\'`. Los espacios alrededor del signo igual son opcionales). Los parámetros reconocidos actualmente son:

`host`

Nombre del ordenador al que conectarse. Si se da una cadena de longitud distinta de cero, se utiliza comunicación TCP/IP. El uso de este parámetro supone una búsqueda del nombre del ordenador. Ver `hostaddr`.

`hostaddr`

Dirección IP del ordenador al que se debe conectar. Debería estar en el formato estandar de números y puntos, como se usan en las funciones de BSD `inet_aton` y otras. Si se especifica una cadena de longitud distinta de cero, se emplea una comunicación TCP/IP.

El uso de `hostaddr` en lugar de `host` permite a la aplicación evitar la búsqueda del nombre de ordenador, lo que puede ser importante en aplicaciones que tienen una limitación de tiempo. Sin embargo la autenticación Kerberos necesita el nombre del ordenador. En este caso se aplica la siguiente secuencia. Si se especifica `host` sin `hostaddr`, se fuerza la búsqueda del nombre del ordenador. Si se especifica `hostaddr` sin `host`, el valor de `hostaddr` dará la dirección remota; si se emplea Kerberos, se buscará de modo inverso el nombre del ordenador. Si se dan tanto `host` como `hostaddr`, el valor de `hostaddr` dará la dirección remota; el valor de `host` se ignorará, a menos que se emplee Kerberos, en cuyo caso ese valor se utilizará para la autenticación Kerberos. Nótese que `libpq` fallará si se pasa un nombre de ordenador que no sea el nombre de la máquina en `hostaddr`.

Cuando no se empleen ni uno ni otro, `libpq` conectará utilizando un socket de dominio local.

`port`

Número del puerto para la conexión en el ordenador servidor, o extensión del nombre de fichero del socket para conexión de dominio Unix.

`dbname`

Nombre de la base de datos.

`user`

Nombre del usuario que se debe conectar.

`password`

Password que se deberá utilizar si el servidor solicita una autenticación con `password`.

`options`

Se pueden enviar las opciones Trace/debug al servidor.

`tty`

Un fichero o `tty` para la salida de la depuración opcional desde el servidor.

Si no se especifica ningún parámetro, se comprobarán las correspondiente variables de entorno. Si no se encuentran fijadas, se emplean los valores de defecto codificadas en el programa. El valor devuelto es un puntero a una estructura abstracta que representa la conexión al servidor.

Esta función no salva hebra.

- `PQsetdbLogin` Realiza una nueva conexión al servidor de base de datos.

```
PGconn *PQsetdbLogin(const char *pghost,
                    const char *pgport,
                    const char *pgoptions,
                    const char *pgtty,
                    const char *dbName,
                    const char *login,
                    const char *pwd)
```

Esta función es la predecesora de `PQconnectdb`, con un número fijado de parámetros, pero con la misma funcionalidad.

Esta función no salva hebra.

- `PQsetdb` Realiza una nueva conexión al servidor de base de datos.

```
PGconn *PQsetdb(char *pghost,
               char *pgport,
               char *pgoptions,
               char *pgtty,
               char *dbName)
```

Esta es una función que llama a `PQsetdbLogin()` con punteros nulos para los parámetros `login` y `pwd`. Se proporciona inicialmente para mantener compatibilidad con programas antiguos.

- `PQconnectStart` `PQconnectPoll` Realizan una conexión al servidor de base de datos de forma no bloqueante.

```
PGconn *PQconnectStart(const char *conninfo)
PostgresPollingStatusType *PQconnectPoll(PGconn *conn)
```

Estas dos rutinas se utilizan para abrir una conexión al servidor de base de datos tal que la hebra de ejecución de la aplicación no queda bloqueada en el I/O remoto mientras lo hace.

La conexión a la base de datos se realiza utilizando los parámetros dados en la cadena `conninfo`, que se pasa a `PQconnectStart`. Esta cadena está en el mismo formato que se describió antes para `PQconnectdb`.

Ni `PQconnectStart` ni `PQconnectPoll` bloquearán, aunque se exigen un cierto número de restricciones:

- Los parámetros `hostaddr` y `host` se utilizan apropiadamente para asegurar que no se realizan consultas de nombre ni de nombre inverso. Vea la documentación de estos parámetros bajo `PQconnectdb` antes para obtener más detalles.
- Si llama a `PQtrace`, asegúrese de que el objeto de la secuencia en la cual realiza usted un rastreo no bloquea.
- Asegúrese usted mismo de que el socket se encuentra en el estado apropiado antes de llamar a `PQconnectPoll`, como se describe más abajo.

Para empezar, llame `conn=PQconnectStart("<connection_info_string>")`. Si `conn` es `NULL`, `libpq` habrá sido incapaz de crear una nueva estructura `PGconn`. De otro modo, se devolverá un puntero `PGconn` válido (aunque todavía no represente una conexión válida a la base de datos). Al regreso de `PQconnectStart`, llame a `status=PQstatus(conn)`. Si `status` es igual a `CONNECTION_BAD`, `PQconnectStart` habrá fallado.

Si PQconnectStart funciona con éxito, el siguiente paso es comprobar libpq de forma que pueda proceder con la secuencia de conexión. Realice un bucle como sigue: Considere que por defecto una conexión se encuentra 'inactiva'. Si el último PQconnectPoll devolvió PGRES_POLLING_ACTIVE, considere ahora que la conexión está 'activa'. Si el último PQconnectPoll(conn) devolvió PGRES_POLLING_READING, realice una select para leer en PQsocket(conn). Si devolvió PGRES_POLLING_WRITING, realice una select para escribir en PQsocket(conn). Si todavía tiene que llamar a PQconnectPoll, es decir, tras llamar a PQconnectStart, comportese como si hubiera devuelto PGRES_POLLING_WRITING. Si la select muestra que el socket está preparado (ready), consíderelo 'activo'. Si ya ha decidido que esta conexión está 'activa', llame de nuevo a PQconnectPoll(conn). Si esta llamada devuelve PGRES_POLLING_OK, la conexión se habrá establecido con éxito.

Nótese que el uso de select() para asegurar que el socket se encuentra listo es realmente un ejemplo; aquellos que dispongan de otras facilidades disponibles, como una llamada poll(), por supuesto pueden utilizarla en su lugar.

En cualquier momento durante la conexión, se puede comprobar la situación de esta conexión, llamando a PQstatus. Si el resultado es CONNECTION_BAD, el procedimiento de conexión habrá fallado; si es CONNECTION_OK, la conexión está funcionando correctamente. Cualquiera de estas situaciones se puede detectar del mismo modo a partir del valor de retorno de PQconnectPoll, como antes. Otras situaciones se pueden mostrar durante (y sólo durante) un procedimiento de conexión asíncrona. Estos indican la situación actual del procedimiento de conexión, y se pueden utilizar para proporcionar información de retorno al usuario, por ejemplo. Estas situaciones pueden incluir:

- CONNECTION_STARTED: Esperando que se realice una conexión.
- CONNECTION_MADE: Conexión OK; esperando para enviar.
- CONNECTION_AWAITING_RESPONSE: Esperando una respuesta del postmaster.
- CONNECTION_AUTH_OK: Recibida autenticación, espera que arranque del servidor.
- CONNECTION_SETENV: Negociando el entorno.

Téngase en cuenta que, aunque estas constantes se conservarán (para mantener la compatibilidad), una aplicación nunca debería basarse en la aparición de las mismas en un orden particular, o en todas, o en que las situaciones siempre tengan un valor de estos documentados. Una aplicación podría hacer algo así:

```
switch(PQstatus(conn))
{
    case CONNECTION_STARTED:
        feedback = "Connecting...";
        break;

    case CONNECTION_MADE:
        feedback = "Connected to server...";
        break;

    .
    .
    .

    default:
        feedback = "Connecting...";
}
```

Nótese que si PQconnectStart devuelve un puntero no nulo, deberá usted llamar a PQfinish cuando haya terminado con él, para disponer de la estructura y de cualquier bloque de memoria asociado. Se debe hacer esto incluso si ha fallado una llamada a PQconnectStart o a PQconnectPoll.

PQconnectPoll actualmente bloqueará si libpq se compila con USE_SSL definido. Esta restricción se eliminará en el futuro.

PQconnectPoll actualmente bloqueará bajo Windows, a menos que libpq se compile con WIN32_NON_BLOCKING_CONNECTIONS definida. Este código no se ha probado aún bajo Windows, de forma que actualmente se encuentra desactivado por defecto. Esto podría cambiar en el futuro.

Estas funciones dejarán el socket en un estado de no-bloqueo como si se hubiese llamado a PQsetnonblocking.

Estas funciones no aseguran la hebra.

- PQconnndefaults Devuelve la opciones de conexión de defecto.

```
PQconninfoOption *PQconnndefaults(void)

struct PQconninfoOption
{
    char    *keyword;    /* Palabra clave de la opción */
    char    *envvar;     /* Nombre de la variable de entorno que recoge su valor
                        si no se da expresamente */
    char    *compiled;   /* Valor de defecto en el código fuente si tampoco se asigna
                        variable de entorno */
    char    *val;        /* Valor de la opción */
    char    *label;     /* Etiqueta para el campo en el diálogo de conexión */
    char    *dispchar;   /* Carácter a mostrar para este campo en un diálogo de conexión.

                        Los valores son:
                        ""      Muestra el valor entrado tal cual es
                        "*"    Campo de Password - ocultar el valor
                        "D"    Opciones de depuración - No crea un campo por defecto */
    int     dispsize;    /* Tamaño del campo en caracteres para diálogo */
}
```

Devuelve la dirección de la estructura de opciones de conexión. Esta se puede utilizar para determinar todas las opciones posibles de PQconnectdb y sus valores de defecto actuales. El valor de retorno apunta a una matriz de estructuras PQconninfoOption, que termina con una entrada que tiene un puntero a NULL. Note que los valores de defecto (los campos "val") dependerán de las variables de entorno y del resto del contexto. Cuando se le llame, se deben tratar los datos de las opciones de conexión como de sólo lectura.

Esta función no salva hebra.

- PQfinish Cierra la conexión con el servidor. También libera la memoria utilizada por el objeto PGconn.

```
void PQfinish(PGconn *conn)
```

Téngase en cuenta que incluso si falló el intento de conexión con el servidor (como se indicaba en PQstatus), la aplicación deberá llamar a PQfinish para liberar la

memoria utilizada por el objeto PGconn. No se debería utilizar el puntero PGconn una vez que se ha llamado a PQfinish.

- **PQreset** Inicializa el puerto de comunicación con el servidor.

```
void PQreset(PGconn *conn)
```

Esta función cerrará la conexión con el servidor e intentará establecer una nueva conexión al mismo postmaster, utilizando todos los mismos parámetros anteriores. Se puede utilizar para recuperar un error si una conexión que estaba trabajando se pierde.

- **PQresetStart** **PQresetPoll** Limpian el puerto de comunicación con el servidor de forma no bloqueante.

```
int PQresetStart(PGconn *conn);
```

```
PostgresPollingStatusType PQresetPoll(PGconn *conn);
```

Estas funciones cerrarán la conexión al servidor e intentarán reestablecer una nueva conexión con el mismo postmaster, utilizando los mismos parámetros previamente utilizados. Esto puede ser utilizable para recuperaciones de errores si se pierde una conexión que estaba trabajando. Difieren de del anterior PQreset en que lo hacen de una forma no bloqueante. Estas funciones sufren las mismas restricciones que PQconnectStart y PQconnectPoll.

Ejecute PQresetStart. Si devuelve 0, la limpieza ha fallado. Si devuelve 1, pruebe la limpieza utilizando PQresetPoll exactamente en la misma forma en que habría creado la conexión utilizando PQconnectPoll.

Los programadores de aplicaciones con libpq deberían ser cuidadosos de mantener la abstracción de PGconn. Utilice las funciones siguientes para tomar el contenido de PGconn. Prohíba las referencias directas a los campos de la estructura PGconn, ya que están sujetas a cambios en el futuro. (A partir de PostgreSQL 6.4, la definición de la estructura PGconn incluso ya no se proporciona en libpq-fe.h. Si tiene usted viejas aplicaciones que acceden a campos de PGconn directamente, puede usted conservarlas utilizando para incluirla libpq-int.h también, pero le recomendamos encarecidamente que fije pronto el código).

- **PQdb** Devuelve el nombre de la base de datos de la conexión.

```
char *PQdb(const PGconn *conn)
```

PQdb y las siguientes funciones devuelven los valores establecidos en la conexión. Estos valores se fijan para toda la vida de PGconn. object.

- **PQuser** Devuelve el nombre de usuario de la conexión.

```
char *PQuser(const PGconn *conn)
```

- **PQpass** Devuelve la palabra de paso de la conexión.

```
char *PQpass(const PGconn *conn)
```

- **PQhost** Devuelve el nombre del ordenador de servidor de la conexión.

```
char *PQhost(const PGconn *conn)
```

- **PQport** Devuelve el puerto de la conexión.

```
char *PQport(const PGconn *conn)
```

- **PQtty** Devuelve el terminal tty de depuración de la conexión.

```
char *PQtty(const PGconn *conn)
```

- `PQoptions` Devuelve las opciones de servidor utilizadas en la conexión.

```
char *PQoptions(const PGconn *conn)
```

- `PQstatus` Devuelve la situación (status) de la conexión.

```
ConnStatusType PQstatus(const PGconn *conn)
```

La situación puede tomar varios valores diferentes. Sin embargo, sólo dos de ellos tienen significado fuera de un procedimiento de conexión asíncrona: `CONNECTION_OK` o `CONNECTION_BAD`. Una buena conexión a la base de datos tiene es status `CONNECTION_OK`. Una conexión fallida se señala con la situación `CONNECTION_BAD`. Normalmente, una situación de OK se mantendrá hasta `PQfinish`, pero un fallo de las comunicaciones puede provocar un cambio prematuro de la situación a `CONNECTION_BAD`. En ese caso, la aplicación podría intentar recuperar la comunicación llamando a `PQreset`.

Para averiguar otras posibles situaciones que podrían comprobarse, revise las entradas de `PQconnectStart` y `PQconnectPoll`.

- `PQerrorMessage` Devuelve el mensaje de error más reciente que haya generado alguna operación en la conexión.

```
char *PQerrorMessage(const PGconn* conn);
```

Casi todas las funciones de libpq fijarán el valor de `PQerrorMessage` si fallan. Tenga en cuenta que por convención de libpq, un `PQerrorMessage` no vacío incluirá un carácter "nueva línea" final.

- `PQbackendPID` Devuelve el identificador (ID) del proceso del servidor que está controland esta conexión.

```
int PQbackendPID(const PGconn *conn);
```

El PID del servidor es utilizable si se quiere hacer depuración de errores y para comparar los mensajes de NOTIFY (que incluyen el PID del servidor que está realizando la notificación). Tenga en cuenta que el PID corresponde a un proceso que se está ejecutando en el ordenador servidor de la base de datos, ¡no en el ordenador local!

- `PQsetenvStart` `PQsetenvPoll` `PQsetenvAbort` Realizan una negociación del ambiente.

```
PGsetenvHandle *PQsetenvStart(PGconn *conn)
```

```
PostgresPollingStatusType *PQsetenvPoll(PGsetenvHandle handle)
```

```
void PQsetenvAbort(PGsetenvHandle handle)
```

Estas dos rutinas se pueden utilizar para re-ejecutar la negociación del entorno que ocurre durante la apertura de una conexión al servidor de la base de datos. No tengo idea de para qué se puede aprovechar esto (¿la tiene alguien?), pero quizá resulte interesante para los usuarios poder reconfigurar su codificación de caracteres en caliente, por ejemplo.

Estas funciones no bloquean, sujeto a las restricciones aplicadas a PQconnectStart y PQconnectPoll.

Para empezar, llame a `handle=PQsetenvStart(conn)`, donde `conn` es una conexión abierta con el servidor de la base de datos. Si `handle` es NULL, libpq habrá sido incapaz de situar una nueva estructura `PQsetenvHandle`. En otro caso, se devuelve una estructura `handle` válida. (N. del T. Dejo la palabra `handle` como identificador de una estructura de datos la aplicación, aunque evidentemente el usuario podrá utilizar el nombre que desee. Conociendo los programas que yo programo, normalmente usaría un nombre como `con_servidor`, por ejemplo). Este `handle` se piensa que sea opaco: sólo debe utilizarlo para llamar a otras funciones de libpq (`PQsetenvPoll`, por ejemplo).

Elija el procesamiento utilizando `PQsetenvPoll`, exactamente del mismo modo en que hubiese creado la conexión utilizando `PQconnectPoll`.

El procedimiento se puede abortar en cualquier momento llamando a `PQsetenvAbort(handle)`.

Estas funciones no aseguran la hebra.

- `PQsetenv` Realiza una negociación del entorno.

```
int PQsetenv(PGconn *conn)
```

Esta función realiza las mismas tareas que `PQsetenvStart` y `PQsetenvPoll`, pero bloquea para hacerlo. Devuelve 1 en caso de éxito, y 0 en caso de fracaso.

Funciones de Ejecución de Consultas

Una vez que se ha establecido correctamente una conexión con un servidor de base de datos, se utilizan las funciones que se muestran a continuación para realizar consultas y comandos de SQL.

- `PQexec` Emite una consulta a Postgres y espera el resultado.

```
PGresult *PQexec(PGconn *conn,
                 const char *query);
```

Devuelve un puntero `PGresult` o, posiblemente, un puntero NULL. Generalmente devolverá un puntero no nulo, excepto en condiciones de "fuera de memoria" (out-of-memory) o errores serios tales como la incapacidad de enviar la consulta al servidor. Si se devuelve un puntero nulo, se debería tratar de la misma forma que un resultado `PGRES_FATAL_ERROR`. Para conseguir más información sobre el error, utilice `PQerrorMessage`.

La estructura `PGresult` encapsula el resultado devuelto por el servidor a la consulta. Los programadores de aplicaciones con libpq deberían mostrarse cuidadosos de mantener la abstracción de `PGresult`. Prohiban la referencia directa a los campos de la estructura `PGresult`, porque están sujetos a cambios en el futuro. (Incluso a partir de la versión 6.4 de Postgres, ha dejado de proporcionarse la definición de `PGresult` en `libpq-fe.h`. Si tiene usted código antiguo que accede directamente a los campos de `PGresult`, puede mantenerlo utilizando `libpq-int.h` también, pero le recomendamos que ajuste pronto el código).

- `PQresultStatus` Devuelve la situación (status) resultante de una consulta.

```
ExecStatusType PQresultStatus(const PGresult *res)
```

`PQresultStatus` puede devolver uno de los siguientes valores:

- `PGRES_EMPTY_QUERY` – La cadena enviada al servidor estaba vacía.
- `PGRES_COMMAND_OK` – El comando se ha ejecutado con éxito sin devolver datos.
- `PGRES_TUPLES_OK` – La consulta se ha ejecutado con éxito.
- `PGRES_COPY_OUT` – Se ha arrancado la transmisión de datos desde el servidor (Copy Out)
- `PGRES_COPY_IN` – Se ha arrancado la transmisión de datos hacia el servidor (Copy In)
- `PGRES_BAD_RESPONSE` – El servidor ha dado una respuesta desconocida.
- `PGRES_NONFATAL_ERROR`
- `PGRES_FATAL_ERROR`

Si la situación del resultado es `PGRES_TUPLES_OK`, las rutinas descritas más abajo se pueden utilizar para recuperar las tuplas devueltas por la consulta. Tengase en cuenta que una `SELECT` que intente recuperar 0 (cero) tuplas también mostrará `PGRES_TUPLES_OK`. `PGRES_COMMAND_OK` es para comandos que nunca devuelven tuplas (`INSERT`, `UPDATE`, etc). Una respuesta `PGRES_EMPTY_QUERY` indica que hay un error en el programa cliente.

- `PQresStatus` Convierte los tipos enumerados devueltos por `PQresultStatus` en una cadena constante que describe el código de la situación.

```
char *PQresStatus(ExecStatusType status);
```

- `PQresultErrorMessage` Devuelve el mensaje de error asociado con la consulta, o una cadena vacía si no hay error.

```
char *PQresultErrorMessage(const PGresult *res);
```

Siguiendo inmediatamente a una llamada a `PQexec` o `PQgetResult`, `PQerrorMessage` (sobre la conexión) devolverá la misma cadena que `PQresultErrorMessage` (sobre el resultado). Sin embargo, un `PGresult` mantendrá su mensaje de error hasta que sea destruido, mientras que el mensaje de error de la conexión cambiará cuando se realicen subsiguientes operaciones. Utilice `PQresultErrorMessage` cuando quiera conocer la situación asociada a un `PGresult` particular; utilice `PQerrorMessage` cuando quiera conocer la situación de la última operación en la conexión.

- `PQntuples` Devuelve el número de tuplas (instancias) del resultado de la consulta.

```
int PQntuples(const PGresult *res);
```

- `PQnfields` Devuelve el número de campos (atributos) de cada tupla del resultado de la consulta.

```
int PQnfields(const PGresult *res);
```

- `PQbinaryTuples` Devuelve 1 si `PGresult` contiene datos binarios en las tuplas, y 0 si contiene datos ASCII.

```
int PQbinaryTuples(const PGresult *res);
```

Actualmente, los datos binarios de las tuplas solo los puede recuperar una consulta que extraiga datos de un cursor `BINARY`.

- `PQfname` Devuelve el nombre del campo (atributo) asociado con el índice de campo dado. Los índices de campo empiezan con 0.

```
char *PQfname(const PGresult *res,
              int field_index);
```

- `PQfnumber` Devuelve el índice del campo (atributo) asociado con el nombre del campo dado.

```
int PQfnumber(const PGresult *res,
              const char *field_name);
```

Se devuelve -1 si el nombre de campo dado no se corresponde con ningún campo.

- `PQftype` Devuelve el tipo de campo asociado con el índice del campo dado. El entero devuelto es una codificación interna del tipo. Los índices de campo empiezan con 0.

```
Oid PQftype(const PGresult *res,
            int field_num);
```

Puede usted consultar la tabla de sistema `pg_type` para obtener el nombre y propiedades de los diferentes tipos de datos. Los OID,s de los tipos de datos incluidos por defecto están definidos en `src/include/catalog/pg_type.h`, en el árbol de fuentes del producto.

- `PQfsize` Devuelve el tamaño en bytes del campo asociado con el índice de campo dado. Los índices de campo empiezan con 0.

```
int PQfsize(const PGresult *res,
            int field_index);
```

`Qfsize` devuelve el espacio reservado para este campo en una tupla de base de datos, en otras palabras, el tamaño de la representación binaria del tipo de datos en el servidor. Se devuelve -1 si el campo es de tamaño variable.

- `PQfmod` Devuelve los datos de la modificación específica del tipo del campo asociado con el índice del campo dado. Los índices de campo empiezan en 0.

```
int PQfmod(const PGresult *res,
           int field_index);
```

- `PQgetvalue` Devuelve un valor de un único campo (atributo) de una tupla de `PGresult`. Los índices de tuplas y de campos empiezan con 0.

```
char* PQgetvalue(const PGresult *res,
                 int tup_num,
                 int field_num);
```

Para la mayoría de las consultas, el valor devuelto por `PQgetvalue` es una cadena ASCII terminada en un valor NULL que representa el valor del atributo. Pero si el valor de `PQbinaryTuples()` es 1, es valor que devuelve `PQgetvalue` es la representación binaria del tipo en el formato interno del servidor (pero no incluye la palabra del tamaño, si el campo es de longitud variable). Es entonces responsabilidad del programador interpretar y convertir los datos en el tipo C correcto. El puntero devuelto por `PQgetvalue` apunta a una zona de almacenaje que forma parte de la estructura `PGresult`. No se la debería modificar, sino que se debería copiar explícitamente el valor a otra estructura de almacenamiento si se pretende utilizar una vez pasado el tiempo de vida de la estructura `PGresult` misma.

- `PQgetlength` Devuelve la longitud de un campo (atributo) en bytes. Los índices de tupla y de campo empiezan en 0.

```
int PQgetlength(const PGresult *res,
```



```
int tup_num,
int field_num);
```

Esta es la longitud de los datos actuales para el valor de datos particular, es decir, el tamaño del objeto apuntado por PQgetvalue. Notese que para valores representados en ASCII, este tamaño tiene poco que ver con el tamaño binario indicado por PQfsize.

- PQgetisnull Prueba un campo por si tiene un valor NULL. Los índices de tupla y de campo empiezan con 0.

```
int PQgetisnull(const PGresult *res,
int tup_num,
int field_num);
```

Esta función devuelve 1 si el campo contiene un NULL, o 0 si contiene un valor no nulo. (Tenga en cuenta que PQgetvalue devolverá una cadena vacía, no un puntero nulo, para un campo NULL).

- PQcmdStatus Devuelve la cadena de la situación del comando para el comando SQL que generó el PGresult.

```
char * PQcmdStatus(const PGresult *res);
```

- PQcmdTuples Devuelve el número de filas afectadas por el comando SQL.

```
char * PQcmdTuples(const PGresult *res);
```

Si el comando SQL que generó el PGresult era INSERT, UPDATE o DELETE, devolverá una cadena que contiene el número de filas afectadas. Si el comando era cualquier otro, devolverá una cadena vacía.

- PQoidValue Devuelve el identificador de objeto (oid) de la tupla insertada, si el comando SQL era una INSERT. En caso contrario, devuelve InvalidOid.

```
Oid PQoidValue(const PGresult *res);
```

Tanto el tipo Oid como la constante Invalid se definirán cuando incluya usted el fichero de cabeceras libpq. Ambos serán de tipo entero (integer).

- PQoidStatus Devuelve una cadena con el identificador de objeto de la tupla insertada si el comando SQL era una INSERT. En otro caso devuelve una cadena vacía.

```
char * PQoidStatus(const PGresult *res);
```

Esta función se ha despreciado en favor de PQoidValue y no asegura la hebra.

- PQprint Imprime todas las tuplas y, opcionalmente, los nombres de los atributos en la salida especificada.

```
void PQprint(FILE* fout, /* output stream */
const PGresult *res,
const PQprintOpt *po);
```

```
struct {
pqbool header; /* Imprime las cabeceras de los campos de salida
y el contador de filas. */
pqbool align; /* Fija la alineación de los campos. */
pqbool standard; /* old brain dead format */
pqbool html3; /* tabula la salida en html */
pqbool expanded; /* expande las tablas */
pqbool pager; /* Usa el paginador para la salida si se ne-
cesita. */
char *fieldSep; /* separador de campos */
char *tableOpt; /* lo inserta en <tabla ...> de HTML */
char *caption; /* HTML <caption> */
char **fieldName; /* cadena terminada en null de nombres de cam-
po alternativos. */
```

```
} PQprintOpt;
```

psql utilizaba anteriormente esta función para imprimir los resultados de las consultas, pero este ya no es el caso, y esta función ya no se soporta activamente.

- **PQclear** Libera la zona de almacenamiento asociada con PGresult. Todos los resultados de las consultas deberían liberarse con PQclear cuando ya no son necesarios.

```
void PQclear(PGresult *res);
```

Puede usted conserar un objeto PGresult tanto tiempo como lo necesite; no se conservará si realiza una nueva consulta, e incluso si se pierde la conexión. Para evitar esto, debe usted llamar a PQclear. No hacerlo, repercute en pérdidas de memoria en la aplicación cliente.

- **PQmakeEmptyPGresult** Construye un objeto PGresult vacío con la situación que se propone.

```
PGresult* PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
```

Esta es una rutina interna de libpq para reservar e inicializar un objeto PGresult vacío. Está exportada porque algunas aplicaciones consideran interesante generar objetos resultado (particularmente objetos con situaciones de error) por si mismas. Si conn no es NULL y status indica un error, el mensaje de error (errorMessage) de la conexión en curso se copia en el PGresult. Recuerde que debería llamar a PQclear para este objeto también, del mismo modo que para un PGresult devuelto por la libpq misma.

Procesamiento Asíncrono de Consultas

La función PQexec es adecuada para emitir consultas en aplicaciones síncronas sencillas. Sin embargo, tiene una porción de deficiencias importantes:

- **PQexec** espera hasta que se completa la consulta. La aplicación puede tener otro trabajo para hacer (como por ejemplo mantener una interfaz de usuario), en cuyo caso no se querrá bloquear esperando la respuesta.
- Una vez que el control se pasa a PQexec, la aplicación cliente tiene muy difícil intentar cancelar la consulta en curso. (Se puede hacer con un manipulador de señales, pero no de otra forma).
- **PQexec** sólo puede devolver una estructura PGresult. Si la cadena de la consulta emitida contiene múltiples comandos SQL, se perderán todos excepto el último.

Las aplicaciones que no se quieren encontrar con estas limitaciones, pueden utilizar en su lugar las funciones que subyacen bajo PQexec: PQsendQuery y PQgetResult.

Para los programas antiguos que utilizaban esta funcionalidad utilizando PQputline y PQputnbytes y esperaban bloqueados el envío de datos del servidor, se añadió la función PQsetnonblocking.

Las aplicaciones antiguas pueden rechazar el uso de PQsetnonblocking y mantener el comportamiento anterior potencialmente bloqueante. Los programas más nuevos pueden utilizar PQsetnonblocking para conseguir una conexión con el servidor completamente no bloqueante.

- **PQsetnonblocking** fija el estado de la conexión a no bloqueante.

```
int PQsetnonblocking(PGconn *conn)
```

Esta función asegura que las llamadas a `PQputline`, `PQputnbytes`, `PQsendQuery` y `PQendcopy` se ejecutarán sin bloquo, devolviendo en su lugar un error si necesitan ser llamadas de nuevo.

Cuando una conexión a una base de datos se ha fijado como no bloqueante, y se llama a `PQexec`, se cambiará el estado temporalmente a bloqueante, hasta que se completa la ejecución de `PQexec`.

Se espera que en el próximo futuro, la mayoría de libp se haga segura para la funcionalida de `PQsetnonblocking`.

- `PQisnonblocking` Devuelve la situación de bloqueante o no de la conexión a la base de datos.

```
int PQisnonblocking(const PGconn *conn)
```

Devuelve `TRUE` si la conexión está fijada a modo no bloqueante, y `FALSE` si está fijada a bloqueante.

- `PQsendQuery` Envía una consulta a Postgres sin esperar los resultados. Devuelve `TRUE` si la consulta se despachó correctamente, y `FALSE` si no fue así (en cuyo caso, utilice `PQerrorMessage` para obtener más información sobre el fallo).

```
int PQsendQuery(PGconn *conn,
               const char *query);
```

Tras llamar correctamente a `PQsendQuery`, llame a `PQgetResult` una o más veces para obtener el resultado de la consulta. No se debe volver a llamar a `PQsendQuery` en la misma conexión hasta que `PQgetResult` devuelva `NULL`, indicando que la consulta se ha realizado.

- `PQgetResult` Espera el siguiente resultado de una ejecución previa de `PQsendQuery`, y lo devuelve. Se devuelve `NULL` cuando la consulta está completa y ya no habrá más resultados.

```
PGresult *PQgetResult(PGconn *conn);
```

Se debe llamar a `PQgetResult` repetidamente hasta que devuelva `NULL`, indicando que la consulta se ha realizado. (Si se la llama cuando no hay ninguna consulta activa, simplemente devolverá `NULL` desde el principio). Cada uno de los resultados no nulos de `PQgetResult` debería procesarse utilizando las mismas funciones de acceso a `PGresult` previamente descritas. No olvide liberar cada objeto resultado con `PQclear` cuando lo haya hecho. Nótese que `PQgetResult` sólo bloqueará si hay una consulta activa y `PQconsumeInput` aún no a leído los datos de respuesta necesarios.

Utilizando `PQsendQuery` y `PQgetResult` se resuelve uno de los problemas de `PQexec`: Si una cadena de consulta contiene múltiples comandos SQL, los resultados de esos comandos se pueden obtener individualmente. (Esto permite una forma sencilla de procesamiento paralelo: la aplicación cliente puede estar manipulando los resultados de una consulta mientras el servidor sigue trabajando sobre consultas posteriores de la misma cadena de consulta). Sin embargo, la llamada a `PQgetResult` seguirá provocando que el cliente quede bloqueado hasta que el servidor complete el siguiente comando SQL de la cadena. Esto se puede impedir con el uso adecuado de tres funciones más:

- `PQconsumeInput` Si hay una entrada disponible desde el servidor, la recoge.

```
int PQconsumeInput(PGconn *conn);
```

`PQconsumeInput` normalmente devuelve 1 indicando "no hay error", pero devuelve 0 si hay algún tipo de problema (en cuyo caso se fija `PQerrorMessage`). Tenga en cuenta que el resultado no dice si se ha recogido algún dato de entrada. Tras llamar a `PQconsumeInput`, la aplicación deberá revisar `PQisBusy` y/o `PQnotifies` para ver si sus estados han cambiado.

`PQconsumeInput` se puede llamar incluso si la aplicación aún no está preparada para recibir un resultado o una notificación. La rutina leerá los datos disponibles y los situará en un almacenamiento intermedio, provocando así una indicación de preparado para leer a la función `select(2)` para que continúe. La aplicación puede por ello utilizar `PQconsumeInput` para limpiar la condición `select` inmediatamente, y examinar después los resultados tranquilamente.

- `PQisBusy` Devuelve 1 si una consulta está ocupada, es decir, si `PQgetResult` se quedaría bloqueada esperando una entrada. Un 0 indica que se puede llamar a `PQgetResult` con la seguridad de no bloquear.

```
int PQisBusy(PGconn *conn);
```

`PQisBusy` no intentará por sí mismo leer los datos del servidor; por ello, se debe llamar primero a `PQconsumeInput`, o el estado ocupado no terminará nunca.

- `PQflush` Intenta lanzar cualquier dato encolado al servidor, y devuelve 0 si lo consigue (o si la cola de envío está vacía) y EOF si ha fallado por algún motivo.

```
int PQflush(PGconn *conn);
```

Es necesario llamar a `PQflush` en una conexión no bloqueante antes de llamar a `select` para determinar si ha llegado una respuesta. Una respuesta de 0 asegura que no hay datos encolados al servidor que no se hayan enviado todavía. Solo las aplicaciones que han usado `PQsetnonblocking` necesitan esto.

- `PQsocket` Obtiene el número descriptor de fichero para el socket de conexión con el servidor. Un descriptor válido será ≥ 0 ; un resultado de `-1` indica que no hay actualmente ninguna conexión con el servidor abierta.

```
int PQsocket(const PGconn *conn);
```

Se debería utilizar `PQsocket` para obtener el descriptor del socket del servidor para preparar la ejecución de `select(2)`. Esto permite a una aplicación que utiliza conexión bloqueante esperar las respuestas u otras condiciones del servidor. Si el resultado de `select(2)` indica que los datos se pueden leer desde el socket del servidor, debería llamarse a `PQconsumeInput` para leer los datos; tras ello, se pueden utilizar `PQisBusy`, `PQgetResult`, y/o `PQnotifies` para procesar la respuesta.

Las conexiones no bloqueantes (que han utilizado `PQsetnonblocking`) no deberían utilizar `select` hasta que `PQflush` haya devuelto 0 indicando que no quedan datos almacenados esperando ser enviados al servidor.

Una aplicación cliente típica que utilice estas funciones tendrá un bucle principal que utiliza `select(2)` para esperar todas las condiciones a las que debe responder. Una de estas condiciones será la entrada disponible desde el servidor, lo que en términos de `select` son datos legibles en el descriptor de fichero identificado por `PQsocket`. Cuando el bucle principal detecta que hay preparada una entrada, debería llamar a `PQconsumeInput` para leer la entrada. Puede después llamar a `PQisBusy`, seguido de `PQgetResult` si `PQisBusy` devuelve falso (0). Puede llamar también a `PQnotifies` para detectar mensajes NOTIFY (ver "Notificación Asíncrona", más abajo).

Una aplicación cliente que utilice `PQsendQuery/PQgetResult` también puede intentar cancelar una consulta que aún se esté procesando en el servidor.

- `PQrequestCancel` Requiere de Postgres que abandone el procesado de la consulta actual.

```
int PQrequestCancel(PGconn *conn);
```

Devuelve un valor 1 si la cancelación se ha despachado correctamente, y 0 si no (y si no, `PQerrorMessage` dirá porqué). Que se despache correctamente no garantiza que el requerimiento vaya a tener ningún efecto, sin embargo. Sin mirar el valor de retorno de `PQrequestCancel`, la aplicación debe continuar con la secuencia de lectura de resultados normal, utilizando `PQgetResult`. Si la cancelación ha sido efectiva, la consulta actual terminará rápidamente y devolverá un resultado de error. Si falló la cancelación (digamos que porque el servidor ya había procesado la consulta), no se verá ningún resultado.

Nótese que si la consulta forma parte de una transacción, la cancelación abortará la transacción completa.

`PQrequestCancel` se puede invocar de modo seguro desde un manipulador de señales. De esta forma, se puede utilizar en conjunción con `PQexec` plano, si la decisión de cancelar se puede tomar en un manipulador de señales. Por ejemplo, `psql` invoca a `PQrequestCancel` desde un manipulador de la señal `SIGINT`, permitiendo de este modo la cancelación interactiva de consultas que él gestiona a través de `PQexec`. Obsérvese que `PQrequestCancel` no tendrá efecto si la conexión no está abierta en ese momento, o si el servidor no está procesando una consulta.

Ruta Rápida

PostgreSQL proporciona un interfaz rápido para enviar llamadas de función al servidor. Esta es una puerta falsa en la interioridades del sistema, y puede suponer un agujero de seguridad. La mayoría de los usuarios no necesitarán esta característica.

- `PQfn` Requiere la ejecución de una función de servidor a través del interfaz de ruta rápida.

```
PGresult* PQfn(PGconn* conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               const PQArgBlock *args,
               int nargs);
```

El argumento `fnid` es el identificador del objeto de la función que se debe ejecutar. `result_buf` es la zona de almacenamiento en la cual se debe situar el valor devuelto. El programa que hace la llamada deberá haber reservado suficiente espacio para almacenar el valor devuelto (¡no se comprueba!). La longitud del resultado real se devolverá en el entero apuntado por `result_len`. Si se espera un resultado entero de 4 bytes, fije `result_is_int` a 1; de otra forma, fíjelo a 0. (Fijando `result_is_int` a 1 se indica a `libpq` que administre el valor balanceando los bytes si es necesario, de forma que se envíe un valor `int` adecuado a la máquina cliente. Cuando `result_is_int` es 0, la cadena de bytes enviada por el servidor se devuelve sin modificar). `args` y `nargs` especifican los argumentos a pasar a la función.

```
typedef struct {
```

```

    int len;
    int isint;
    union {
        int *ptr;
        int integer;
    } u;
} PQArgBlock;

```

`PQfn` siempre devuelve un `PGresult*` válido. Se debería comprobar el valor de `resultStatus` antes de utilizar el resultado. El programa que hace la llamada es responsable de liberar el `PGresult` con `PQclear` cuando ya no lo necesite.

Notificación Asíncrona

PostgreSQL soporta notificación asíncrona a través de los comandos `LISTEN` y `NOTIFY`. Un servidor registra su interés en una condición de notificación particular con el comando `LISTEN` (y puede dejar de escuchar con el comando `UNLISTEN`). Todos los servidores que están escuchando una condición de notificación particular recibirán la notificación asíncronamente cuando cualquier servidor ejecute un `NOTIFY` de ese nombre de condición. El servidor que realiza la notificación no pasará ninguna otra información particular a los servidores que están escuchando. Por ello, cualquier dato que deba ser comunicado se transfiere habitualmente a través de una relación de base de datos. También habitualmente el nombre de la condición es el mismo de la relación asociada, pero no sólo no es necesario, sino que tampoco lo es que exista ninguna relación asociada.

Las aplicaciones `libpq` emiten los comandos `LISTEN` y `UNLISTEN` como consultas SQL ordinarias. Subsiguientemente, la llegada de mensajes `NOTIFY` se puede detectar llamando a `PQnotifies()`.

- `PQnotifies` Devuelve la siguiente notificación de una lista de mensajes de notificación aún no manipulados recibidos desde el servidor. Devuelve `NULL` si no hay notificaciones pendientes. Una vez se devuelve una notificación con `PQnotifies`, esta se considera manipulada y se borrará de la lista de notificaciones.

```
PGnotify* PQnotifies(PGconn *conn);
```

```

typedef struct PGnotify {
    char relname[NAMEDATALEN];    /* nombre de la relación */
                                   /* que contiene los datos */
    int  be_pid;                  /* identificador del proceso ser-
vidor */
} PGnotify;

```

Tras procesar un objeto `PGnotify` devuelto por `PQnotifies`, asegúrese de liberarlo con `free()` para impedir pérdidas de memoria.

Nota: En PostgreSQL 6.4 y posteriores, el `be_pid` corresponde al servidor que realiza la notificación, mientras que en versiones anteriores era siempre el PID del propio servidor.

La segunda muestra de programa da un ejemplo del uso de la notificación asíncrona.

`PQnotifies()` actualmente no lee datos del servidor; únicamente devuelve mensajes previamente absorbidos por otra función `libpq`. En versiones previas de `libpq`, la única forma de asegurar la recepción a tiempo de mensajes NOTIFY era emitir constantemente consultas, incluso vacías, y comprobar entonces `PQnotifies()` tras cada `PQexec()`. Aunque esto funcionaba, se menospreciaba como una forma de malgastar poder de proceso.

Una forma mejor de comprobar los mensajes NOTIFY cuando no se dispone de consultas utilizables es hacer una llamada `PQconsumeInput()`, y comprobar entonces `PQnotifies()`. Se puede usar `select(2)` para esperar la llegada de datos del servidor, no utilizando en este caso potencia de CPU a menos que se tenga algo que hacer. Nótese que esta forma funcionará correctamente mientras utilice usted `PQsendQuery/PQgetResult` o simplemente `PQexec` para las consultas. Debería usted, sin embargo, recordar comprobar `PQnotifies()` tras cada `PQgetResult` o `PQexec` para comprobar si ha llegado alguna notificación durante el procesado de la consulta.

Funciones Asociadas con el Comando COPY

El comando COPY en PostgreSQL tiene opciones para leer o escribir en la conexión de red utilizada para `libpq`. Por ello, se necesitan funciones para acceder a su conexión de red directamente, de forma que las aplicaciones puedan obtener ventajas de esta capacidad.

Estas funciones sólo se deberían utilizar tras obtener un objeto resultado `PGRES_COPY_OUT` o `PGRES_COPY_IN` a partir de `PQexec` o `PQgetResult`.

- `PQgetline` Lee una línea de caracteres terminada con un carácter "newline" (transmitida por el servidor) en una cadena de almacenamiento de tamaño "length".

```
int PQgetline(PGconn *conn,
              char *string,
              int length)
```

De modo similar a `fgets(3)`, esta rutina copia longitud-1 caracteres en una cadena. Es como `gets(3)`, sin embargo, en que el carácter "newline" de terminación en un carácter nulo. `PQgetline` devuelve EOF en el EOF, 0 si se ha leído la línea entera, y 1 si se ha llenado la zona de almacenamiento, pero aún no se ha leído el fin de línea.

Observe que la aplicación deberá comprobar si la nueva línea consiste en los dos caracteres "\.", lo que indicaría que el servidor ha terminado de enviar los resultados del comando copy. Si la aplicación debería recibir líneas que son más largas de longitud-1, deberá tener cuidado de reconocer la línea "\." correctamente (y no confunde, por ejemplo, el final de una larga línea de datos con la línea de terminación). El código de `src/bin/psql/copy.c` contiene rutinas de ejemplo que manipulan correctamente el protocolo copy.

- `PQgetlineAsync` Lee una línea de caracteres terminada con "newline" (transmitida por el servidor) en una zona de almacenamiento sin bloquear.

```
int PQgetlineAsync(PGconn *conn,
                  char *buffer,
                  int bufsize)
```

Esta rutina es similar a `PQgetline`, pero la pueden utilizar aplicaciones que leen datos de COPY asíncronamente, ya que es sin bloqueo. Una vez realizado el comando COPY y obtenido una respuesta `PGRES_COPY_OUT`, la aplicación debería

llamar a `PQconsumeInput` y `PQgetlineAsync` hasta que se detecte la señal end-of-data. Contra `PQgetline`, esta rutina toma la responsabilidad de detectar el EOF. En cada llamada, `PQgetlineAsync` devolverá datos, siempre que tenga disponible una línea de datos completa terminada en "newline" en el almacenamiento de entrada de libpq, o si la línea de datos de entrada es demasiado larga para colocarla en el almacenamiento ofrecido por la aplicación de llamada. En otro caso, no se devuelve ningún dato hasta que llega el resto de la línea.

La rutina devuelve -1 si reconoce el marcador end-of-copy-data, 0 si no tiene datos disponibles, o un número positivo que la el número de bytes de datos devueltos. Si se devuelve -1, la aplicación que realiza la llamada deberá llamar a `PQendcopy`, y volver después al procesado normal. Los datos devueltos no se extenderán más allá de un carácter "newline". Si es posible, se devolverá una línea completa cada vez. Pero si el almacenamiento ofrecido por la aplicación que realiza la llamada es demasiado pequeño para recibir una línea enviada por el servidor, se devolverán datos parciales. Se puede detectar esto comprobando si el último byte devuelto es "\n" o no. La cadena devuelta no se termina con un carácter nulo. (Si quiere usted añadir un NULL de terminación, asegúrese de pasar una longitud del almacenamiento más pequeña que el tamaño del almacenamiento de que realmente dispone).

- `PQputline` Envía una cadena terminada en carácter nulo al servidor. Devuelve 0 si todo funciona bien, y EOF si es incapaz de enviar la cadena.

```
int PQputline(PGconn *conn,
              const char *string);
```

Tenga en cuenta que la aplicación debe enviar explícitamente los dos caracteres "\." en una línea de final para indicar al servidor que ha terminado de enviarle datos.

- `PQputnbytes` Envía una cadena terminada en un carácter no nulo al servidor. Devuelve 0 si todo va bien, y EOF si es incapaz de enviar la cadena.

```
int PQputnbytes(PGconn *conn,
                const char *buffer,
                int nbytes);
```

Esta función es exactamente igual que `PQputline`, excepto en que el almacenamiento de datos no necesita estar terminado en un carácter nulo, una vez que el número de bytes que se envían se especifica directamente.

- `PQendcopy` Sincroniza con el servidor. Esta función espera hasta que el servidor ha terminado la copia. Debería utilizarse bien cuando se ha enviado la última cadena al servidor utilizando `PQputline` o cuando se ha recibido la última cadena desde el servidor utilizando `PQgetline`. Debe utilizarse, o el servidor puede recibir "out of sync (fuera de sincronía)" con el cliente. Una vez vuelve de esta función, el servidor está preparado para recibir la siguiente consulta. El valor devuelto es 0 si se completa con éxito, o diferente de cero en otro caso.

```
int PQendcopy(PGconn *conn);
```

Como un ejemplo:

```
PQexec(conn, "create table foo (a int4, b char(16), d float8)");
PQexec(conn, "copy foo from stdin");
PQputline(conn, "3\tthello world\t4.5\n");
PQputline(conn, "4\tgoodbye world\t7.11\n");
...
PQputline(conn, "\\.\n");
PQendcopy(conn);
```


Cuando se está utilizando `PQgetResult`, la aplicación debería responder a un resultado `PGRES_COPY_OUT` ejecutando repetidamente `PQgetline`, seguido de `PQendcopy` una vez se detecta la línea de terminación. Debería entonces volver al bucle `PQgetResult` loop until hasta que `PQgetResult` devuelva `NULL`. Similarmente, un resultado `PGRES_COPY_IN` se procesa por una serie de llamadas a `PQputline` seguidas por `PQendcopy`, y volviendo entonces al bucle `PQgetResult`. Esta organización asegurará que un comando de copia de entrada o de salida embebido en una serie de comandos SQL se ejecutará correctamente.

Las aplicaciones antiguas habitualmente emiten una copia de entrada o de salida a través de `PQexec` y asumen que la transacción ha terminado tras el `PQendcopy`. Este mecanismo trabajará adecuadamente sólo si la copia de entrada/salida es el único comando SQL de la cadena de consulta.

Funciones de Trazado de libpq

- `PQtrace` Activa la traza de la comunicación cliente/servidor para depurar la corriente de ficheros.

```
void PQtrace(PGconn *conn
             FILE *debug_port)
```

- `PQuntrace` Desactiva la traza arrancada por `PQtrace`.

```
void PQuntrace(PGconn *conn)
```

Funciones de control de libpq

- `PQsetNoticeProcessor` Controla le informe de mensajes de aviso y alarma generados por libpq.

```
typedef void (*PQnoticeProcessor) (void *arg, const char *message);
```

```
PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                    PQnoticeProcessor proc,
                    void *arg);
```

Por defecto, libpq imprime los mensajes de aviso del servidor así como unos pocos mensajes de error que genera por sí mismo en `stderr`. Este comportamiento se puede sobrescribir suministrando una función de llamada de alarma que haga alguna otra cosa con los mensajes. La función de llamada de alarma utiliza como argumentos el texto del mensaje de error (que incluye un caracter final de "newline"), y un puntero vacío que es el mismo pasado a `PQsetNoticeProcessor`. (Este puntero se puede utilizar para acceder a estados específicos de la aplicación si se necesita). El procesador de avisos de defecto es simplemente:

```
static void
defaultNoticeProcessor(void * arg, const char * message)
{
    fprintf(stderr, "%s", message);
}
```

Para utilizar un procesador de avisos especial, llame a `PQsetNoticeProcessor` inmediatamente tras la creación de un nuevo objeto `PGconn`.

El valor devuelto es el puntero al procesador de avisos previo. Si proporciona usted un puntero de función de llamada a `NUL`, no se toma ninguna acción, sino que se devuelve el puntero activo.

Variables de Entorno

Se pueden utilizar las siguientes variables de entorno para seleccionar valores de parámetros de conexión de defecto, que serán utilizadas por `PQconnectdb` o `PQsetdbLogin` si no se especifica ningún otro valor directamente en el código que realiza la llamada. Son utilizables para impedir codificar nombres de bases de datos en simples programas de aplicación.

- `PGHOST` fija el nombre del del ordenador servidor. Si se especifica una cadena de longitud distinta de 0, se utiliza comunicación TCP/IP. Sin un nombre de ordenador, libpq conectará utilizando un socket del dominio Unix local.
- `PGPORT` fija el puerto de defecto o la extensión de fichero del socket de dominio Unix local para la comunicación con el servidor PostgreSQL.
- `PGDATABASE` fija el nombre de la base de datos PostgreSQL.
- `PGUSER` fija el nombre de usuario utilizado para conectarse a la base de datos y para autenticación.
- `PGPASSWORD` fija la palabra de paso utilizada si el servidor solicita autenticación por palabra clave.
- `PGREALM` sets the Kerberos realm to use with PostgreSQL, if it is different from the local realm. If `PGREALM` is set, PostgreSQL applications will attempt authentication with servers for this realm and use separate ticket files to avoid conflicts with local ticket files. This environment variable is only used if Kerberos authentication is selected by the backend.
- `PGOPTIONS` fija opciones de ejecución adicionales para el servidor PostgreSQL.
- `PGTTY` fija el fichero o tty en el que se mostrarán los mensajes de depuración del servidor.

Las siguientes variables de entorno se pueden usar para especificar el comportamiento de defecto de los usuario para cada sesión de PostgreSQL:

- `PGDATESTYLE` Fija el estilo de la representación de fechas y horas.
- `PGTZ` Fija el valor de defecto para la zona horaria.

Las siguientes variables de entorno se pueden utilizar para especificar el comportamiento interno de defecto de cada sesión de PostgreSQL:

- PGGEQO fija el modo de defecto para el optimizador genético.

Refierase al comando SQL **SET** para obtener información sobre los valores correctos de estas variables de entorno.

Programas de Ejemplo

Programa de Ejemplo 1

```

/*
 * testlibpq.c Test the C version of Libpq, the Postgres frontend
 * library.
 * testlibpq.c Probar la versión C de libpq, la librería para aplicaciones
 * cliente de PostgreSQL
 *
 */
#include <stdio.h>
#include "libpq-fe.h"

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char        *pghost,
                *pgport,
                *pgoptions,
                *pgtty;
    char        *dbName;
    int          nFields;
    int          i,
                j;

    /* FILE *debug; */

    PGconn      *conn;
    PGresult     *res;

    /*
     * begin, by setting the parameters for a backend connection if the
     * parameters are null, then the system will try to use reasonable
     * defaults by looking up environment variables or, failing that,
     * using hardwired constants
     */
    /* Se empieza fijando los parámetros de una conexión al servidor. Si los
     * parámetros son nulos, el sistema probará a utilizar valores de defecto
     * razonables para buscar en las variables de entorno, y, si esto falla,
     * utilizará constantes incluidas directamente en el código.
     */
    pghost = NULL;                /* host name of the backend server */
    pgport = NULL;                /* nombre del ordenador servidor. */
    /* port of the backend server */

```

```

                                /* puerto asignado al servidor */
pgoptions = NULL;              /* special options to start up the backend
                                * server */
                                /* opciones especiales para arrancar el ser-
vidor */
pgtty = NULL;                  /* debugging tty for the backend ser-
ver */
                                /* tty (terminal) para depuración del ser-
vidor */
dbName = "template1";

/* make a connection to the database */
/* conectar con el servidor */
conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

/*
 * check to see that the backend connection was successfully made
 * se comprueba si la conexión se ha realizado con éxito.
 */
if (PQstatus(conn) == CONNECTION_BAD)
{
    fprintf(stderr, "Connection to database '%s' failed.\n", dbName);
    fprintf(stderr, "%s", PQerrorMessage(conn));
    exit_nicely(conn);
}

/* debug = fopen("/tmp/trace.out", "w"); */
/* PQtrace(conn, debug); */

/* start a transaction block */
/* comienza un bloque de transacción */
res = PQexec(conn, "BEGIN");
if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "BEGIN command failed\n");
    PQclear(res);
    exit_nicely(conn);
}

/*
 * should PQclear PGresult whenever it is no longer needed to avoid
 * memory leaks
 * se debería PQclear PGresult una vez que ya no es necesario, pa-
ra impedir
 * pérdidas de memoria.
 */
PQclear(res);

/*
 * fetch instances from the pg_database, the system catalog of
 * databases
 * se recogen las instancias a partir de pg_database, el catálogo de sis-
tema de
 * bases de datos
 */
res = PQexec(conn, "DECLARE mycursor CURSOR FOR select * from pg_database");
if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "DECLARE CURSOR command failed\n");
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

```

```

res = PQexec(conn, "FETCH ALL in mycursor");
if (!res || PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "FETCH ALL command didn't return tuples properly\n");
    /* no se han recogido tuplas de bases de datos */
    PQclear(res);
    exit_nicely(conn);
}

/* first, print out the attribute names */
/* primero, se imprimen los nombres de los atributos */
nFields = PQnfields(res);
for (i = 0; i < nFields; i++)
    printf("%-15s", PQfname(res, i));
printf("\n\n");

/* next, print out the instances */
/* a continuación, se imprimen las instancias */
for (i = 0; i < PQntuples(res); i++)
{
    for (j = 0; j < nFields; j++)
        printf("%-15s", PQgetvalue(res, i, j));
    printf("\n");
}
PQclear(res);

/* close the cursor */
/* se cierra el cursor */
res = PQexec(conn, "CLOSE mycursor");
PQclear(res);

/* commit the transaction */
/* se asegura la transacción */
res = PQexec(conn, "COMMIT");
PQclear(res);

/* close the connection to the database and cleanup */
/* se cierra la conexión a la base de datos y se limpia */
PQfinish(conn);

/* fclose(debug); */
}

```

Programa de Ejemplo 2

```

/*
 * testlibpq2.c
 * Test of the asynchronous notification interface
 * Se comprueba el interfaz de notificaciones asíncronas.
 *
 * Start this program, then from psql in another window do
 * NOTIFY TBL2;
 * Arranque este programa, y luego, desde psql en otra ventana ejecute
 * NOTIFY TBL2;
 *
 * Or, if you want to get fancy, try this:
 * Populate a database with the following:
 * O, si quiere hacer algo más elegante, intente esto:

```

```

*      alimente una base de datos con lo siguiente:
*
*      CREATE TABLE TBL1 (i int4);
*
*      CREATE TABLE TBL2 (i int4);
*
*      CREATE RULE r1 AS ON INSERT TO TBL1 DO
*          (INSERT INTO TBL2 values (new.i); NOTIFY TBL2);
*
* and do
*     y haga
*
*     INSERT INTO TBL1 values (10);
*
*/
#include <stdio.h>
#include "libpq-fe.h"

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char        *pghost,
                *pgport,
                *pgoptions,
                *pgtty;
    char        *dbName;
    int         nFields;
    int         i,
                j;

    PGconn      *conn;
    PGresult    *res;
    PGnotify    *notify;

    /*
     * begin, by setting the parameters for a backend connection if the
     * parameters are null, then the system will try to use reasonable
     * defaults by looking up environment variables or, failing that,
     * using hardwired constants
     */
    /* Se empieza fijando los parámetros de una conexión al servidor. Si los
     * parámetros son nulos, el sistema probará a utilizar valores de defecto
     * razonables para buscar en las variables de entorno, y, si esto falla,
     * utilizará constantes incluidas directamente en el código.
     */
    pghost = NULL;                /* host name of the backend server */
    pgport = NULL;                /* nombre del ordenador del servidor */
    pgoptions = NULL;            /* port of the backend server */
    pgtty = NULL;                /* puerto asignado al servidor */

    /* special options to start up the backend
     * server */
    /* opciones especiales para arrancar el ser-
    vidor */

    /* debugging tty for the backend ser-
    ver */
    /* tty (terminal) de depuración del ser-
    vidor */

```

```

        dbName = getenv("USER");    /* change this to the name of your test
                                     * database */
                                     /* cambie esto para asignarlo al nom-
bre de su
                                     * base de datos de prueba */

/* make a connection to the database */
/* Se hace a conexión a la base de datos */
conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

/*
 * check to see that the backend connection was successfully made
 * Se comprueba si la conexión ha funcionado correctamente.
 */
if (PQstatus(conn) == CONNECTION_BAD)
{
    fprintf(stderr, "Connection to database '%s' failed.\n", dbName);
    fprintf(stderr, "%s", PQerrorMessage(conn));
    exit_nicely(conn);
}

/* Se declara el interés en TBL2 */
res = PQexec(conn, "LISTEN TBL2");
if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "LISTEN command failed\n");
    PQclear(res);
    exit_nicely(conn);
}

/*
 * should PQclear PGresult whenever it is no longer needed to avoid
 * memory leaks
 * Se debería PQclear PGresult una vez que ya no es necesario, para
 * impedir pérdidas de memoria.
 */
PQclear(res);

while (1)
{
    /*
     * wait a little bit between checks; waiting with select()
     * would be more efficient.
     * esperamos un poquito entre comprobaciones; esperar con select()
     * sería más eficiente.
     */
    sleep(1);
    /* collect any asynchronous backend messages */
    /* Se recoge asíncronamente cualquier mensaje del servi-
dor */
    PQconsumeInput(conn);
    /* check for asynchronous notify messages */
    /* Se comprueban los mensajes de notificación asíncrona */
    while ((notify = PQnotifies(conn)) != NULL)
    {
        fprintf(stderr,
            "ASYNC NOTIFY of '%s' from backend pid '%d' received\n",
            notify->relname, notify->be_pid);
        free(notify);
    }
}

```

```

        /* close the connection to the database and cleanup */
        /*      Se cierra la conexión con la base de datos y se limpia */
        PQfinish(conn);
    }

```

Programa de Ejemplo 3

```

/*
 * testlibpq3.c Test the C version of Libpq, the Postgres frontend
 * library. tests the binary cursor interface
 *      Se comprueba el interfaz de cursores binarios.
 *
 *
 *
 * populate a database by doing the following:
 *      Alimente una base de datos con lo siguiente:
 *
 * CREATE TABLE test1 (i int4, d float4, p polygon);
 *
 * INSERT INTO test1 values (1, 3.567, '(3.0, 4.0, 1.0,
 * 2.0)'::polygon);
 *
 * INSERT INTO test1 values (2, 89.05, '(4.0, 3.0, 2.0,
 * 1.0)'::polygon);
 *
 * the expected output is:
 *      La salida esperada es:
 *
 * tuple 0: got i = (4 bytes) 1, d = (4 bytes) 3.567000, p = (4
 * bytes) 2 points   bbox = (hi=3.000000/4.000000, lo =
 * 1.000000,2.000000) tuple 1: got i = (4 bytes) 2, d = (4 bytes)
 * 89.050003, p = (4 bytes) 2 points   bbox =
 * (hi=4.000000/3.000000, lo = 2.000000,1.000000)
 *
 *
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "utils/geo-decls.h"      /* for the POLYGON type */
                                  /* para el tipo POLYGON */

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char        *pghost,
                *pgport,
                *pgoptions,
                *pgtty;
    char        *dbName;
    int          nFields;

```



```

int          i,
             j;
int          i_fnum,
             d_fnum,
             p_fnum;
PGconn       *conn;
PGresult     *res;

/*
 * begin, by setting the parameters for a backend connection if the
 * parameters are null, then the system will try to use reasonable
 * defaults by looking up environment variables or, failing that,
 * using hardwired constants
 */
/* Se empieza fijando los parámetros de una conexión al servidor. Si los
 * parámetros son nulos, el sistema probará a utilizar valores de defecto
 * razonables para buscar en las variables de entorno, y, si esto falla,
 * utilizará constantes incluidas directamente en el código.
 */
pghost = NULL;          /* host name of the backend server */
                        /* nombre de ordenador del servidor */
pgport = NULL;          /* port of the backend server */
                        /* puerto asignado al servidor. */
pgoptions = NULL;       /* special options to start up the backend
                        * server */
                        /* opciones especiales para arrancar el ser-
vidor */
pgtty = NULL;          /* debugging tty for the backend ser-
ver */
                        /* tty (terminal) para depurar el ser-
vidor */

dbName = getenv("USER"); /* change this to the name of your test
                        * database */
                        /* cambie esto al nombre de su base de da-
tos de
                        * prueba */

/* make a connection to the database */
/* Se hace la conexión a la base de datos */
conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

/*
 * check to see that the backend connection was successfully made
 * Se comprueba que la conexión se ha realizado correctamente
 */
if (PQstatus(conn) == CONNECTION_BAD)
{
    fprintf(stderr, "Connection to database '%s' failed.\n", dbName);
    fprintf(stderr, "%s", PQerrorMessage(conn));
    exit_nicely(conn);
}

/* start a transaction block */
res = PQexec(conn, "BEGIN");
if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "BEGIN command failed\n");
    PQclear(res);
    exit_nicely(conn);
}
/*

```

```

    * should PQclear PGresult whenever it is no longer needed to avoid
    * memory leaks
    * Se debería PQclear PGresult una vez que ya no es necesari-
rio, para
    * evitar pérdidas de memoria.
    */
    PQclear(res);

    /*
    * fetch instances from the pg_database, the system catalog of
    * databases
    * se recogen las instancias de pg_database, el catálogo de sis-
tema de
    * bases de datos.
    */
    res = PQexec(conn, "DECLARE mycursor BINARY CURSOR FOR select * from test1");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "FETCH ALL in mycursor");
    if (!res || PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL command didn't return tuples properly\n");
        /* no se ha recogido ninguna base de datos */
        PQclear(res);
        exit_nicely(conn);
    }

    i_fnum = PQfnumber(res, "i");
    d_fnum = PQfnumber(res, "d");
    p_fnum = PQfnumber(res, "p");

    for (i = 0; i < 3; i++)
    {
        printf("type[%d] = %d, size[%d] = %d\n",
            i, PQftype(res, i),
            i, PQfsize(res, i));
    }
    for (i = 0; i < PQntuples(res); i++)
    {
        int      *ival;
        float     *dval;
        int       plen;
        POLYGON   *pval;

        /* we hard-wire this to the 3 fields we know about */
        /* codificamos lo que sigue para los tres campos de los que
        * algo */
        ival = (int *) PQgetvalue(res, i, i_fnum);
        dval = (float *) PQgetvalue(res, i, d_fnum);
        plen = PQgetlength(res, i, p_fnum);

        /*
        * plen doesn't include the length field so need to
        * increment by VARHDSZ
        * plen no incluye el campo de longitud, por lo que necesitamos
        * incrementar con VARHDSZ
        */
    }

```


Capítulo 47. libpq C++ Binding

`libpq++` es la API C++ API para Postgres. `libpq++` es un conjunto de clases que permiten a los programas cliente conectarse al servidor de Postgres. Estas conexiones vienen de dos formas: una Clase de Base de Datos, y una clase de Objetos Grandes.

La Clase de Base de datos está pensada para manipular una base de datos. Puede usted enviar toda suerte de consultas SQL al servidor Postgres, y recibir las repuestas del servidor.

La Clase de Objetos Grandes está pensada para manipular los objetos grandes en la base de datos. Aunque una instancia de Objetos Grandes puede enviar consultas normales al servidor de Postgres, sólo está pensado para consultas sencillas que no devuelven ningún dato. Un objeto grande se debería ver como una cadena de un fichero. En el futuro, debería comportarse de forma muy próxima a las cadenas de fichero de C++ `cin`, `cout` y `cerr`.

Este capítulo está basado en la documentación para la librería C `libpq`. Al final de esta sección se listan tres programas cortos como ejemplo de programación con `libpq++` (aunque no necesariamente de una buena programación). Hay muchos tipos de aplicaciones `libpq++` en `src/libpq++/examples`, incluyendo el código fuente de los tres ejemplos expuestos en este capítulo.

Control e Inicialización

Variables de Entorno.

Las siguientes variables de entorno se pueden utilizar para fijar variables de defecto para un entorno, y para evitar codificar nombres de la base de datos en un programa de aplicación:

Nota: Diríjase a `libpq` para conseguir una lista completa de opciones de conexión.

Las siguientes variables de entorno se pueden utilizar para seleccionar valores de parámetros de conexión de defecto, que serán utilizados por `PQconnectdb` o `PQsetdbLogin` si no se ha especificado directamente ningún otro valor por parte del código que realiza la llamada. Son utilizables para impedir la codificación de nombres de base de datos en programas de aplicación sencillos.

Nota: `libpq++` utiliza sólo variables de entorno o cadenas del tipo `conninfo` de `PQconnectdb`.

- `PGHOST` fija el nombre del ordenador servidor de defecto. Si se especifica una cadena de longitud distinta de 0, se utiliza comunicación TCP/IP. Sin un nombre de host, `libpq` conectará utilizando una conexión (un socket) del dominio Unix local.
- `PGPORT` fija el puerto de defecto o la extensión del fichero de conexión del dominio Unix local para la comunicación con el servidor Postgres.
- `PGDATABASE` fija el nombre de la base de datos Postgres de defecto.

- PGUSER fija el nombre de usuario utilizado para conectarse a la base de datos y para la autenticación.
- PGPASSWORD fija la palabra de paso utilizada si el servidor solicita autenticación de la palabra de paso.
- PGREALM fija el reino Kerberos a utilizar con Postgres, si es diferente del reino local. Si se fija PGREALM, las aplicaciones Postgres intentarán la autenticación con los servidores de este reino, y utilizarán ficheros de ticket separados, para impedir conflictos con los ficheros de ticket locales. Esta variable de entorno sólo se utiliza si el servidor selecciona la autenticación Kerberos.
- PGOPTIONS fija opciones de tiempo de ejecución adicionales para el servidor de Postgres.
- PGTTY fija el fichero o tty al cual el servidor enviará los mensajes de seguimiento de la ejecución.

Las siguientes variables de entorno se pueden utilizar para especificar el comportamiento de defecto para los usuarios para cada sesión de Postgres:

- PGDATESTYLE fija el estilo de defecto de la representación de fecha/hora.
- PGTZ fija la zona horaria de defecto.

Las siguientes variables de entorno se pueden utilizar para especificar el comportamiento interno de defecto para cada sesión de Postgres:

- PGGEQO fija el modo de defecto para el optimizador genérico.

Encontrará información sobre los valores correctos de estas variables de entorno en el comando **SET** de SQL.

Clases de libpq++

Clase de Conexión: `PgConnection`

La clase de conexión realiza la conexión en vigor sobre la base de datos, y se hereda por todas las clases de acceso.

Clase Base de Datos: `PgDatabase`

La Clase Base de Datos proporciona los objetos C++ que tienen una conexión con el servidor. Para crear tal objeto, primero se necesita el entorno adecuado para acceder al servidor. Los constructores siguientes se relacionan con la realización de conexiones a un servidor desde programas C++.

Funciones de Conexión a la Base de Datos

- `PgConnection` realiza una nueva conexión a un servidor de base de datos.

```
PgConnection::PgConnection(const char *conninfo)
```

Aunque habitualmente se le llama desde una de las clases de acceso, también es posible conectarse a un servidor creando un objeto `PgConnection`.

- `ConnectionBad` Devuelve si la conexión con el servidor de datos se consiguió o no.

```
int PgConnection::ConnectionBad()
```

Devuelve `VERDADERO` si la conexión falló.

- `Status` devuelve el status de la conexión con el servidor.

```
ConnStatusType PgConnection::Status()
```

Devuelve `CONNECTION_OK` o `CONNECTION_BAD` dependiendo del estado de la conexión.

- `PgDatabase` realiza una nueva conexión a un servidor de base de datos.

```
PgDatabase(const char *conninfo)
```

Tras la creación de `PgDatabase`, se debería comprobar para asegurarse de que la conexión se ha realizado con éxito antes de enviar consultas al objeto. Se puede hacer fácilmente recogiendo el status actual del objeto `PgDatabase` con los métodos `Status` o `ConnectionBad`.

- `DBName` Devuelve el nombre de la base de datos actual.

```
const char *PgConnection::DBName()
```

- `Notifies` Devuelve la siguiente notificación de una lista de mensajes de notificación sin manipular recibidos desde el servidor.

```
PGnotify* PgConnection::Notifies()
```

Vea `PQnotifies()` para conseguir más detalles.

Funciones de Ejecución de las Consultas

- `Exec` Envía una consulta al servidor. Probablemente sea más deseable utilizar una de las dos siguientes funciones.

```
ExecStatusType PgConnection::Exec(const char* query)
```

Devuelve el resultado de la consulta. Se pueden esperar los siguientes resultados.

`PGRES_EMPTY_QUERY`

`PGRES_COMMAND_OK`, si la consulta era un comando

PGRES_TUPLES_OK, si la consulta ha devuelto tuplas
 PGRES_COPY_OUT
 PGRES_COPY_IN
 PGRES_BAD_RESPONSE, si se ha recibido una respuesta inesperada
 PGRES_NONFATAL_ERROR
 PGRES_FATAL_ERROR

- ExecCommandOk Envía una consulta de comando sobre el servidor.

```
int PgConnection::ExecCommandOk(const char *query)
```

Devuelve VERDADERO si la consulta de comando se realizó con éxito.

- ExecTuplesOk Envía una consulta de tuplas al servidor.

```
int PgConnection::ExecTuplesOk(const char *query)
```

Devuelve VERDADERO si la consulta se realizó con éxito.

- ErrorMessage Devuelve el texto del último mensaje de error.

```
const char *PgConnection::ErrorMessage()
```

- Tuples Devuelve el número de tuplas (instancias) presentes en el resultado de la consulta.

```
int PgDatabase::Tuples()
```

- Fields Devuelve el número de campos (atributos) de cada tupla de las que componen el resultado de la consulta.

```
int PgDatabase::Fields()
```

- FieldName Devuelve el nombre del campo (atributo) asociado al índice de campo dado. Los índices de campo empiezan en 0.

```
const char *PgDatabase::FieldName(int field_num)
```

- FieldNum PQfnumber Devuelve el índice del campo (atributo) asociado con el nombre del campo dado.

```
int PgDatabase::FieldNum(const char* field_name)
```

Si el nombre de campo no se corresponde con ninguno de los de la consulta se devuelve un valor de -1.

- FieldType Devuelve el tipo de campo asociado con el índice de campo dado. El entero devuelto es una codificación interna del tipo. Los índices de campo empiezan en 0.

```
Oid PgDatabase::FieldType(int field_num)
```

- FieldType Devuelve el tipo de campo asociado con el nombre de campo dado. El entero devuelto es una codificación interna del tipo. Los índices de campo empiezan en 0.

```
Oid PgDatabase::FieldType(const char* field_name)
```


- `FieldSize` Devuelve el tamaño en bytes del campo asociado con el índice de campo dado. Los índices de campo empiezan en 0.

```
short PgDatabase::FieldSize(int field_num)
```

Devuelve el lugar ocupado por este campo en la tupla de base de datos, dando el número de campo. En otras palabras, el tamaño de la representación binaria en el servidor del tipo de datos. Devolverá -1 si se trata de un campo de tamaño variable.

- `FieldSize` Devuelve el tamaño en bytes del campo asociado con el índice de campo dado. Los índices de campo empiezan en 0.

```
short PgDatabase::FieldSize(const char *field_name)
```

Devuelve el espacio ocupado por este campo en la tupla de base de datos dando el nombre del campo. En otras palabras, el tamaño de la representación binaria del tipo de datos en el servidor. Se devolverá -1 si el campo es de tamaño variable.

- `GetValue` Devuelve un valor único de campo (atributo) en una tupla de `PGresult`. Los índices de tupla y de campo empiezan en 0.

```
const char *PgDatabase::GetValue(int tup_num, int field_num)
```

Para la mayoría de las consultas, el valor devuelto por `GetValue` es una representación en ASCII terminada con un null del valor del atributo. Pero si `BinaryTuples()` es VERDADERO, el valor que devuelve `GetValue` es la representación binaria del tipo en el formato interno del servidor (pero sin incluir el tamaño, en el caso de campos de longitud variable). Es responsabilidad del programador traducir los datos al tipo C correcto. El puntero de devuelve `GetValue` apunta al almacenamiento que es parte de la estructura `PGresult`. No se debería modificar, y se debería copiar explícitamente el valor a otro almacenamiento si se debe utilizar pasado el tiempo de vida de la estructura `PGresult` misma. `BinaryTuples()` no se ha implementado aún.

- `GetValue` Devuelve el valor de un único campo (atributo) en una tupla de `PGresult`. Los índices de tupla y campo empiezan en 0.

```
const char *PgDatabase::GetValue(int tup_num, const char *field_name)
```

Para la mayoría de las consultas, el valor devuelto por `GetValue` es una representación en ASCII terminada con un null del valor del atributo. Pero si `BinaryTuples()` es VERDADERO, el valor que devuelve `GetValue` es la representación binaria del tipo en el formato interno del servidor (pero sin incluir el tamaño, en el caso de campos de longitud variable). Es responsabilidad del programador traducir los datos al tipo C correcto. El puntero de devuelve `GetValue` apunta al almacenamiento que es parte de la estructura `PGresult`. No se debería modificar, y se debería copiar explícitamente el valor a otro almacenamiento si se debe utilizar pasado el tiempo de vida de la estructura `PGresult` misma. `BinaryTuples()` no se ha implementado aún.

- `GetLength` Devuelve la longitud de un campo (atributo) en bytes. Los índices de tupla y campo empiezan en 0.

```
int PgDatabase::GetLength(int tup_num, int field_num)
```

Esta es la longitud actual del valor particular del dato, que es el tamaño del objeto apuntado por `GetValue`. Tenga en cuenta que para valores representados en ASCII, este tamaño tiene poco que ver con el tamaño binario indicado por `PQfsize`.

- `GetLength` Devuelve la longitud de un campo (atributo) en bytes. Los índices de tupla y campo empiezan en 0.

```
int PgDatabase::GetLength(int tup_num, const char* field_name)
```

Esta es la longitud actual del valor particular del dato, que es el tamaño del objeto apuntado por GetValue. Tenga en cuenta que para valores representados en ASCII, este tamaño tiene poco que ver con el tamaño binario indicado por PQfsize.

- DisplayTuples Imprime todas las tuplas y, opcionalmente, los nombres de atributo de la corriente de salida especificada.

```
void PgDatabase::DisplayTuples(FILE *out = 0, int fillAlign = 1,
const char* fieldSep = "|",int printHeader = 1, int quiet = 0)
```

- PrintTuples Imprime todas las tuplas y, opcionalmente, los nombres de los atributos en la corriente de salida especificada.

```
void PgDatabase::PrintTuples(FILE *out = 0, int printAttName = 1,
int terseOutput = 0, int width = 0)
```

- GetLine

```
int PgDatabase::GetLine(char* string, int length)
```

- PutLine

```
void PgDatabase::PutLine(const char* string)
```

- OidStatus

```
const char *PgDatabase::OidStatus()
```

- EndCopy

```
int PgDatabase::EndCopy()
```

Notificación Asíncrona

Postgres soporta notificación asíncrona a través de los comandos **LISTEN** y **NOTIFY**. Un servidor registra su interés en un semáforo particular con el comando **LISTEN**. Todos los servidores que están escuchando un semáforo particular identificado por su nombre recibirán una notificación asíncrona cuando otro servidor ejecute un **NOTIFY** para ese nombre. No se pasa ninguna otra información desde el servidor que notifica al servidor que escucha. Por ello, típicamente, cualquier dato actual que se necesite comunicar se transfiere a través de la relación.

Nota: En el pasado, la documentación tenía asociados los nombres utilizados para las notificaciones asíncronas con relaciones o clases. Sin embargo, no hay de hecho unión directa de los dos conceptos en la implementación, y los semáforos identificados con un nombre de hecho no necesitan tener una relación correspondiente previamente definida.

Las aplicaciones con `libpq++` son notificadas cada vez que un servidor al que están conectadas recibe una notificación asíncrona. Sin embargo la comunicación entre el servidor y la aplicación cliente no es asíncrona. La aplicación con `libpq++` debe llamar al servidor para ver si hay información de alguna notificación pendiente. Tras la ejecución de una consulta, una aplicación cliente puede llamar a `PgDatabase::Notifies` para ver si en ese momento se encuentra pendiente algún dato de notificación desde el servidor. `PgDatabase::Notifies` devuelve la notificación de una lista de notificaciones pendientes de manipular desde el servidor. La función devuelve `NULL` si no hay notificaciones pendientes en el servidor. `PgDatabase::Notifies` se comporta como el reparto de una pila. Una vez que `PgDatabase::Notifies` ha devuelto la notificación, esta se considera manipulada y se elimina de la lista de

- `PgDatabase::Notifies` recupera notificaciones pendientes del servidor.

```
PGnotify* PgDatabase::Notifies()
```

El segundo programa de muestra da un ejemplo del uso de notificaciones asíncronas.

Funciones Asociadas con el Comando COPY.

El comando `copy` de Postgres tiene opciones para leer y escribir en la conexión de red utilizada por `libpq++`. Por esta razón, se necesitan funciones para acceder a esta conexión de red directamente, de forma que las aplicaciones puedan tomar ventajas completas de esta capacidad.

- `PgDatabase::GetLine` lee una línea de caracteres terminada con "nueva línea" (transmitida por el servidor) en una zona de almacenamiento (un buffer) *string* de tamaño *length*.

```
int PgDatabase::GetLine(char* string, int length)
```

Como la rutina de sistema de Unix `fgets (3)`, esta rutina copia *length-1* caracteres en *string*. Es como `gets (3)`, sin embargo, en que convierte la terminación "nueva línea" en un carácter null.

`PgDatabase::GetLine` Devuelve EOF al final de un fichero, 0 si se ha leído la línea entera, y 1 si la zona de almacenamiento de ha llenado pero no se ha leído aún el carácter "nueva línea" de terminación.

Nótese que la aplicación debe comprobar si la nueva línea consiste simplemente en único punto ("."), lo que indicaría que el servidor ha terminado de enviar el resultado de `copy`. Por ello, si la aplicación siempre espera recibir líneas que tienen más de *length-1* caracteres de longitud, la aplicación deberá asegurarse de comprobar el valor de retorno de `PgDatabase::GetLine` muy cuidadosamente.

- `PgDatabase::PutLine` Envía un *string* terminado en null al servidor.

```
void PgDatabase::PutLine(char* string)
```

La aplicación debe enviar explícitamente un único punto (".") para indicar al servidor que ha terminado de enviar sus datos.

- `PgDatabase::EndCopy` sincroniza con el servidor.

```
int PgDatabase::EndCopy()
```

Esta función espera hasta que el servidor ha terminado de procesar el comando **copy**. Debería utilizarse bien cuando se ha enviado la última cadena al servidor utilizando `PgDatabase::PutLine`, bien cuando se ha recibido la última cadena desde el servidor utilizando `PgDatabase::GetLine`. Debe utilizarse, o el servidor puede detectar “fuera de sincronía” (out of sync) con la aplicación cliente. Una vez vuelve de esta función, el servidor está preparado para recibir la siguiente consulta.

El valor devuelto es 0 cuando se completa con éxito, y distinto de cero en otro caso.

As an example:

```
PgDatabase data;
data.Exec("create table foo (a int4, b char(16), d float8)");
data.Exec("copy foo from stdin");
data.putline("3\etHello World\et4.5\en");
data.putline("4\etGoodbye World\et7.11\en");
&...
data.putline(".\en");
data.endcopy();
```

Capítulo 48. pg_tcl

`pg_tcl` es un paquete tcl para programas que interactúen con backends de Postgres. Hace que la mayoría de las funciones de `libpq` estén disponibles para scripts de tcl. Este paquete fue originalmente escrito por Jolly Chen.

Comandos

Tabla 48-1. Comandos `pg_tcl`

Comando	Descripción
<code>pg_connect</code>	abre una conexión al servidor backend
<code>pg_disconnect</code>	cierra una conexión
<code>pg_conndefaults</code>	obtiene las opciones de conexión y sus valores por defecto
<code>pg_exec</code>	envía una consulta al backend
<code>pg_result</code>	manipula los resultados de una consulta
<code>pg_select</code>	hace un bucle sobre el resultado de una declaración SELECT
<code>pg_listen</code>	establece una rellamada mensajes NOTIFY
<code>pg_lo_creat</code>	crea un objeto grande
<code>pg_lo_open</code>	abre un objeto grande
<code>pg_lo_close</code>	cierra un objeto grande
<code>pg_lo_read</code>	lee un objeto grande
<code>pg_lo_write</code>	escribe un objeto grande
<code>pg_lo_lseek</code>	busca y se coloca sobre una posición en un objeto grande
<code>pg_lo_tell</code>	devuelve la posición de un objeto grande sobre la que se está
<code>pg_lo_unlink</code>	borra un objeto grande
<code>pg_lo_import</code>	importa un fichero Unix a un objeto grande
<code>pg_lo_export</code>	exporta un objeto grande a un fichero Unix

Estos comandos se describen en otras páginas más adelante.

Las rutinas `pg_lo*` son interfaces a las características de objetos grandes de Postgres. Las funciones han sido diseñadas para imitar a las funciones del sistema análogas en el sistema de ficheros de Unix. Las rutinas `pg_lo*` deberían usarse dentro de un bloque transaccional BEGIN/END porque el descriptor de fichero devuelto por `pg_lo_open` sólo es válido para la transacción en curso. `pg_lo_import` y `pg_lo_export` DEBEN ser usados en un bloque de transacción BEGIN/END.

Ejemplos

He aquí un pequeño ejemplo de cómo usar las rutinas:

```
# getDBs :
#   get the names of all the databases at a given host and port number
#   with the defaults being the localhost and port 5432
#   return them in alphabetical order
proc getDBs { {host "localhost"} {port "5432"} } {
    # datnames is the list to be result
    set conn [pg_connect template1 -host $host -port $port]
    set res [pg_exec $conn "SELECT datname FROM pg_database ORDER BY datname"]
    set ntups [pg_result $res -numTuples]
    for {set i 0} {$i < $ntups} {incr i} {
        lappend datnames [pg_result $res -getTuple $i]
    }
    pg_result $res -clear
    pg_disconnect $conn
    return $datnames
}
```

Información de referencia de comandos *pgtcl*

pg_connect

Nombre

pg_connect — abre una conexión al servidor backend

Synopsis

```
pg_connect -conninfo opcionesConexion
pg_connect nombreDb [-host nombreHost]
    [-port numeroPuerto] [-tty pqtty]
    [-options argumentosOpcionalesBackend]
```

Inputs (estilo nuevo)

opcionesConexion

Un string de opciones de conexión, cada una escrita de la forma *palabraClave* = *valor*.

Inputs (estilo viejo)

nombreBD

Especifica un nombre válido de base de datos.

`[-host nombreHost]`

Especifica el nombre de dominio del servidor backend para *nombreBD*.

`[-port numeroPuerto]`

Especifica el número de puerto IP del servidor backend para *nombreBD*.

`[-tty pgtty]`

Especifica el fichero o el tty (terminal) para mostrar los resultados de depuración provenientes del backend.

`[-options argumentosOpcionalesBackend]`

Especifica opciones para el servidor de backend para *nombreBD*.

Outputs

dbHandle

Si toda ha ido bien, se devuelve un handle para una conexión de base de datos. Los handles comienzan con el prefijo "pgsql".

Descripción

`pg_connect` abre una conexión al backend de Postgres.

Existen dos versiones de la sintaxis. En la vieja, cada posible opción tiene un switch en la declaración de `pg_connect`. En la nueva forma, se utiliza un string como opción que contiene múltiples valores. Vea `pg_conndefaults` para encontrar información sobre las opciones disponibles en la nueva sintaxis.

Uso

XXX thomas 1997-12-24

pg_disconnect

Nombre

`pg_disconnect` — cierra una conexión al servidor backend

Synopsis

`pg_disconnect dbHandle`

Inputs*dbHandle*

Especifica un handle de base de datos válido.

Outputs

Ninguno

Descripción`pg_disconnect` cierra una conexión al backend de Postgres.**pg_conndefaults****Nombre**`pg_conndefaults` — obtiene información sobre los parámetros de la conexión por defecto**Synopsis**`pg_conndefaults`**Inputs**

Ninguno

Outputs*option list*

El resultado es una lista que describe las opciones posibles de conexión y sus valores por defecto. Cada entrada en la lista es una sub-lista del formato siguiente:

{optname label dispchar dispsize value}

donde optname se utiliza como opción en `pg_connect -conninfo`.

Descripción

`pg_conndefaults` devuelve información sobre las opciones de conexión disponibles en `pg_connect -conninfo` y el valor por defecto actual de cada opción.

Uso

`pg_conndefaults`

pg_exec

Nombre

`pg_exec` — envía un string con una consulta al backend

Synopsis

`pg_exec dbHandle stringConsulta`

Inputs

dbHandle

Especifica un handle válido para una base de datos.

stringConsulta

Especifica una consulta SQL válida.

Outputs

handleResultado

Se devolverá un error Tcl si Pgsql no pudo obtener respuesta del backend. De otro modo, se crea un objeto consulta como respuesta y se devuelve un handle para él. Este handle puede ser pasado a `pg_result` para obtener los resultados de la consulta.

Description

`pg_exec` presenta una consulta al backend de Postgres y devuelve un resultado. El resultado de la consulta se encarga de manejar el comienzo con el handle de la conexión y añade un punto un número como resultado.

Tenga en cuenta que la falta de un error Tcl no es una prueba de que la consulta ha tenido éxito! Un mensaje de error devuelto por el backend será procesado como resultado de una consulta con un aviso de fallo, no generándose un error Tcl en `pg_exec`.

pg_result

Nombre

`pg_result` — obtiene información sobre el resultado de una consulta

Synopsis

`pg_result` *handleResult* *opcionResult*

Inputs

handleResult

Es el handle para el resultado de una consulta.

opcionResult

Especifica una de las varias posibles opciones.

Opciones

-status

el estado del resultado.

-error

el mensaje de error, si el estado indica error; de otro modo, un string vacío.

-conn

la conexión que produjo el resultado.

-oid

si el comando fue un INSERT, el tuplo del OID insertado; de otro modo un string vacío.

-numTuples

el número de tuplos devueltos por la consulta.

-numAttrs

el número de atributos en cada tuplo.

-assign nombreArray

asigna el resultado a un array, usando la forma (numTuplo,nombreAtributo).

-assignbyidx nombreArray ?appendstr?

asigna los resultado a un array usando el primer atributo del valor y el resto de nombres de atributos como claves. Si appendstr es pasado, entonces es añadido a cada clave. Brevemente, todos excepto el primer campo de cada tuplo son almacenados en un array, usando una nomenclatura del tipo (valorPrimerCampo,nombreCampoAppendStr).

-getTuple numeroTuplo

devuelve los campos del tuplo indicado en una lista. Los números de tuplo empiezan desde cero.

-tupleArray numeroTuplo nombreArray

almacena los campos del tuplo en el array nombreArray, indexados por nombres de campo. Los número de tuplo empiezan desde cero.

-attributes

devuelve una lista con los nombre de los atributos del tuplo.

-lAttributes

devuelve una lista de sub-listas {nombre tipo tamaño} por cada atributo del tuplo.

-clear

elimina el objeto consulta resultante.

Outputs

el resultado depende de la opción elegida, como se describió más arriba.

Descripción

`pg_result` devuelve información acerca del resultado de una consulta creada por un `pg_exec` anterior.

Puede mantener el resultado de una consulta en tanto en cuanto lo necesite, pero cuando haya terminado con él asegúrese de liberarlo ejecutando `pg_result -clear`. clear. De otro modo, tendrá un "agujero" en la memoria y Pgctl mostrará mensajes indicando que ha creado demasiados objetos consulta.

pg_select

Nombre

`pg_select` — hace un bucle sobre el resultado de una declaración SELECT

Synopsis

```
pg_select handleBD stringConsulta
      varArray procConsulta
```

Inputs

handleBD

Especifica un handle válido para una base de datos.

stringConsulta

Especifica una consulta SQL select válida.

varArray

Un array de variables para los tuplos devueltos.

procConsulta

Procedimiento que se ha ejecutado sobre cada tuplo encontrado.

Outputs

handleResult

el resultado devuelto es un mensaje de error o un handle para un resultado de consulta.

Description

`pg_select` `pg_select` envía una consulta SELECT al backend de Postgres , y ejecuta una porción de código que se le ha pasado por cada tuplo en el resultado de la consulta. El *stringConsulta* debe ser una declaración SELECT. Cualquier otra cosa devuelve un error. La variable *varArray* es un nombre de array usado en el bucle. Por cada tuplo, *varArray* `arrayVar` se rellena con los valores del campo tuplo usando los nombres de campo como índices del array. A partir de aquí *procConsulta* se ejecuta.

Uso

Esto funcionaría si la tabla "table" tiene los campos "control" y "name" (y tal vez otros campos):

```
pg_select $pgconn "SELECT * from table" array {
puts [format "%5d %s" array(control) array(name)]
}
```

pg_listen

Nombre

`pg_listen` — fija o cambia una rellamada para los mensajes NOTIFY asíncronos

Synopsis

```
pg_listen dbHandle notifyName comandoRellamada
```

Inputs

dbHandle

Especifica un handle de base de datos válido.

notifyName

Especifica el nombre de la notificación para empezar o parar de escuchar.

comandoRellamada

Si este parámetro se pasa con un valor no vacío, proporciona el comando a ejecutar cuando una notificación válida llegue.

Outputs

Ninguno

Description

`pg_listen` `pg_listen` crea, cambia o cancela una petición para escuchar mensajes NOTIFY asíncronos desde el backend de Postgres. Con un parámetro `comandoRellamada`, la petición se establecerá o el string de comando de una petición existente será reemplazada. Sin ningún parámetro `comandoRellamada`, se cancelará una petición anterior.

Después de que se establezca una petición `pg_listen`, el string de comando especificado se ejecutará cuando un mensaje NOTIFY que lleve el nombre dado llegue desde el backend. Esto ocurre cuando cualquier aplicación cliente de Postgres muestra un comando NOTIFY haciendo referencia a ese nombre. (Nótese que puede ser, aunque no obligatoriamente, el de una relación existente en la base de datos). El string de comando se ejecuta desde el loop de espera de Tcl. Este es el estado de espera normal de una aplicación escrita con Tk. En shells que no son Tk Tcl, puede ejecutar `update` o `vwait` para provocar que se introduzca el loop de espera.

No debería invocar las declaraciones SQL LISTEN o UNLISTEN directamente cuando esté usando `pg_listen`. Pgtcl se encarga de poner en marcha esas declaraciones por usted. Pero si usted quiere enviar un mensaje NOTIFY, invoque la declaración SQL NOTIFY usando `pg_exec`.

pg_lo_creat

Nombre

`pg_lo_creat` — crea un objeto grande

Synopsis

```
pg_lo_creat conn modo
```

Inputs

conn

Especifica una conexión válida a una base de datos.

modo

Especifica el modo de acceso para el objeto grande.

Outputs

idObjeto

Es el oid (id del objeto) del objeto grande creado.

Descripción

`pg_lo_creat` crea un objeto grande de inversión (Inversion Large Object).

Uso

`modo` puede ser cualquier agrupación con OR de `INV_READ`, `INV_WRITE` e `INV_ARCHIVE`. El carácter delimitador de OR es "|".

```
[pg_lo_creat $conn "INV_READ|INV_WRITE"]
```

pg_lo_open

Nombre

`pg_lo_open` — abre un objeto grande

Synopsis

```
pg_lo_open conn idObjeto modo
```

Inputs

conn

Especifica una conexión válida a una base de datos.

idObjeto

Especifica un oid válido para un objeto grande.

modo

Especifica el modo de acceso para el objeto grande.

Outputs

fd

Un descriptor de fichero para usar posteriormente en rutinas `pg_lo*`.

Descripción

`pg_lo_open` abre un objeto grande de inversión (Inversion Large Object).

Uso

`modo` puede ser "r", "w" o "rw".

pg_lo_close

Nombre

`pg_lo_close` — cierra un objeto grande

Synopsis

```
pg_lo_close conn fd
```

Inputs

conn

Especifica una conexión válida a una base de datos.

fd

Un descriptor de fichero para ser usado posteriormente en rutinas `pg_lo*`.

Outputs

Ninguno

Descripción

`pg_lo_close` cierra un objeto grande de inversión (Inversion Large Object).

Uso

`pg_lo_read`

Nombre

`pg_lo_read` — lee un objeto grande

Synopsis

```
pg_lo_read conn fd bufVar len
```

Inputs

conn

Especifica una conexión válida a una base de datos.

fd

Descriptor de fichero para el objeto grande tomado de `pg_lo_open`.

bufVar

Especifica una variable de buffer válida para contener el segmento del objeto grande.

len

Especifica el tamaño máximo permitido para el segmento del objeto grande.

Outputs

Ninguno

Descripción

`pg_lo_read` lee la mayor parte de los bytes de *len* y lo copia a la variable *bufVar*.

Uso

bufVar debe ser un nombre de variable válido.

pg_lo_write

Nombre

`pg_lo_write` — escribe un objeto grande

Synopsis

```
pg_lo_write conn fd buf len
```

Inputs

conn

Especifica una conexión válida a una base de datos.

fd

Descriptor de fichero para el objeto grande tomado de `pg_lo_open`.

buf

Especifica una variable string válida para escribir en el objeto grande.

len

Especifica el tamaño máximo del string a escribir.

Outputs

Ninguno

Descripción

`pg_lo_write` escribe la mayor parte de *len* bytes a un objeto desde una variable *buf*.

Usage

buf deber ser el string a escribir, no el nombre de la variable.

pg_lo_lseek**Nombre**

`pg_lo_lseek` — busca una posición en un objeto grande

Synopsis

```
pg_lo_lseek conn fd offset whence
```

Inputs

conn

Especifica una conexión válida a una base de datos.

fd

Descriptor de fichero para el objeto tomado de `pg_lo_open`.

offset

Especifica un offset en bytes en base cero.

whence

whence puede ser "SEEK_CUR", "SEEK_END" o "SEEK_SET"

Outputs

Ninguno

Descripción

`pg_lo_lseek` se posiciona en *offset* bytes desde el comienzo, fin o actual posición de un objeto grande.

Uso

whence puede ser "SEEK_CUR", "SEEK_END", o "SEEK_SET".

`pg_lo_tell`

Nombre

`pg_lo_tell` — devuelve la posición actual de búsqueda de un objeto grande

Synopsis

```
pg_lo_tell conn fd
```

Inputs

conn

Especifica una conexión válida a una base de datos.

fd

Descriptor de fichero del objeto tomado de `pg_lo_open`.

Outputs

offset

Un offset en bytes en base zero adecuado para `pg_lo_lseek`.

Description

`pg_lo_tell` devuelve la posición de búsqueda actual a *offset* en bytes desde el comienzo del objeto grande.

Uso

pg_lo_unlink

Nombre

`pg_lo_unlink` — borra un objeto grande

Synopsis

```
pg_lo_unlink conn idObjeto
```

Inputs

conn

Especifica una conexión válida a una base de datos.

idObjeto

Es el identificador para un objeto grande. XXX Es esto igual que idObjetos en otras llamadas?? - thomas 1998-01-11

Outputs

Ninguno

Descripción

`pg_lo_unlink` borra el objeto grande especificado.

Uso

pg_lo_import

Nombre

`pg_lo_import` — importa un objeto grande desde un fichero Unix

Synopsis

```
pg_lo_import conn nombreFichero
```

Inputs

conn

Especifica una conexión válida a una base de datos.

nombreFichero

El nombre del fichero Unix.

Outputs

Ninguno XXX Devuelve esto un idObjeto? Es lo mismo que idObjeto en otras llamadas? thomas - 1998-01-11

Descripción

`pg_lo_import` lee el fichero especificado y pone el contenido del mismo en un objeto grande.

Uso

`pg_lo_import` debe ser llamado dentro de un bloque de transacción BEGIN/END.

pg_lo_export

Nombre

`pg_lo_export` — exporta un objeto grande a un fichero Unix

Synopsis

`pg_lo_export conn idObjeto nombreFichero`

Inputs

conn

Especifica una conexión válida a una base de datos.

idObjeto

Identificador del objeto grande. XXX Es igual a idObjeto en otras llamadas??
thomas - 1998-01-11

nombreFichero

Nombre del fichero Unix.

Outputs

Ninguno. XXX Devuelve un idObjeto? Es esto igual a idObjeto en otras llamadas?
thomas - 1998-01-11

Descripción

pg_lo_export escribe el objeto grande especificado en un fichero Unix.

Uso

pg_lo_export debe ser llamado dentro de un bloque de transacción BEGIN/END.

Capítulo 49. Interfaz ODBC

Nota: Información de fondo realizada originalmente por Tim Goeke¹

ODBC (Open Database Connectivity / Conectividad Abierta para Bases de Datos) es un API abstracto que permite escribir aplicaciones que pueden interoperar con varios servidores RDBMS. ODBC facilita un interfaz de producto neutral entre aplicaciones de usuario final y servidores de bases de datos, permitiendo ya sea a un usuario o desarrollador escribir aplicaciones transportables entre servidores de diferentes fabricantes.

Trasfondo

El API ODBC se conecta en la capa inferior de la aplicación con una fuente de datos compatible con ODBC. Ésta (la fuente de datos) podría ser desde un fichero de texto a una RDBMS Oracle o Postgres.

El acceso en la capa inferior de aplicación se produce gracias a drivers ODBC, o drivers específicos del fabricante que permiten el acceso a los datos. `psqlODBC` es un driver, junto con otros que están disponibles, como los drivers ODBC Openlink.

Cuando se escribe una aplicación ODBC usted, *debería* ser capaz de conectar con *cualquier* base de datos, independientemente del fabricante, siempre y cuando el esquema de la base de datos sea el mismo.

Por ejemplo. Usted podría tener servidores MS SQL Server y Postgres que contuvieran exactamente los mismos datos. Usando ODBC, su aplicación Windows podría hacer exactamente las mismas llamadas y la fuente de datos a nivel interno sería la misma (para la aplicación cliente en Windows).

Insight Distributors² dan soporte continuo y actual a la distribución `psqlODBC`. Suministran un FAQ³, sobre el desarrollo actual del código base, y participan activamente en la lista de correo de interfaces⁴.

Aplicaciones Windows

En la actualidad, las diferencias en los drivers y el nivel de apoyo a ODBC reducen el potencial de ODBC:

- Access, Delphi, y Visual Basic soportan directamente ODBC.
- Bajo C++, y en Visual C++, puede usar el API ODBC de C++.
- En Visual C++, puede usar la clase `CRecordSet`, la cual encapsula el API ODBC dentro de una clase MFC 4.2. Es el camino más fácil si está desarrollando en C++ para Windows bajo Windows NT.

Escritura de Aplicaciones

“¿Si escribo una aplicación para Postgres puedo hacerlo realizando llamadas ODBC al servidor Postgres, o es solo cuando otro programa de bases de datos como MS SQL Server o Access necesita acceder a los datos?”

El camino es el API ODBC. Para código en Visual C++ puede informarse más en el sitio web de Microsoft's o en sus documentos VC++.

Visual Basic y las otras herramientas RAD tienen objetos Recordset que usan directamente ODBC para acceder a los datos. Usando los controles de acceso a datos, puede enlazar rápidamente con la capa ODBC de la base de datos (*muy* rápido).

Jugar con MS Access le ayudará en este cometido. Inténtelo usando `Fichero->Obtener Datos Externos`.

Sugerencia: Primero tendrá que establecer una DNS.

Instalación Unix

ApplixWare tiene un interface de base de datos ODBC soportado en al menos varias plataformas. ApplixWare v4.4.1 ha sido probado bajo Linux con Postgres v6.4 usndo el driver `psqlODBC` contenido en la distribución Postgres.

Construyendo el Driver

Lo primero que debe saberse acerca del driver `psqlODBC` (o cualquier otro driver ODBC) es que debe existir un gestor de driver en el sistema donde va a usarse el driver ODBC. Existe un driver ODBCfreeware para Unix llamado `iodbc` que puede obtenerse en varios puntos de Internet, además de en AS200⁵. Las instrucciones para instalar `iodbc` van más allá del objeto de este documento, pero hay un fichero `README` que puede encontrarse dentro del paquete `iodbc .shar` comprimido que debería explicar cómo realizar la instalación y puesta en marcha.

Una vez dicho esto, cualquier gestor de driver que encuentre para su plataforma debería poder manejar el driver `psqlODBC` o cualquier driver ODBC.

Los ficheros de configuración Unix para `psqlODBC` han sido remozados de forma intensiva recientemente para permitir una fácil construcción en las plataformas soportadas y para permitir el soporte de otras plataformas Unix en el futuro. Los nuevos ficheros de configuración y construcción para el driver deberían convertir el proceso de construcción en algo simple para las plataformas soportadas. Actualmente estas incluyen Linux y FreeBSD but we esperamos que otros usuarios contribuyan con la información necesaria para un rápido crecimiento del número de plataformas para las que puede ser construido el driver.

En la actualidad existen dos métodos distintos para la construcción del driver en función de cómo se haya recibido y sus diferencias se reducen a dónde y cómo ejecutar `configure` y `make`. El driver puede ser construido en modo de equipo aislado, instalación de sólo cliente, o como parte de la distribución Postgres. La instalación aislada es conveniente si usted tiene aplicaciones clientes de ODBC en plataformas múltiples y heterogéneas. La instalación integrada es conveniente cuando las plataformas cliente y servidora son las mismas, o cuando cliente y servidor tienen configuraciones de ejecución similares.

Específicamente si ha recibido el driver `psqlODBC` como parte de la distribución Postgres (a partir de ahora se referenciará como "instalación integrada") entonces podrá configurar el driver ODBC desde el directorio principal de fuentes de la distribución Postgres junto con el resto de las librerías. Si lo recibió como un paquete aislado, entonces podrá ejecutar "configure" y "make" desde el directorio en el que desempaquetó los fuentes.

Instalación integrada

Este procedimiento es apropiado para la instalación integrada.

1. Especificar el argumento `-with-odbc` en la línea de comandos para `src/configure`:

```
% ./configure -with-odbc
% make
```

2. Reconstruir la distribución Postgres:

```
% make install
```

Una vez configurado, el driver ODBC será construido e instalado dentro de las áreas definidas para otros componentes del sistema Postgres. El fichero de configuración de instalación ODBC será colocado en el directorio principal del árbol de destino Postgres (`POSTGRES DIR`). Esto puede ser cambiado en la línea de comandos de `make` como

```
% make ODBCINST=filename install
```

Instalación Integrada Pre-v6.4

Si usted tiene una instalación Postgres más antigua que la v6.4, tiene disponible el árbol de fuentes original, y desea usar la versión más actualizada del driver ODBC, entonces deseará esta modalidad de instalación.

1. Copie el fichero tar de salida a su sistema y desempaquetelo en un directorio vacío.
2. Desde el directorio donde se encuentran los fuentes, teclee:

```
% ./configure
% make
% make POSTGRES DIR=PostgresTopDir install
```

3. Si desea instalar los componentes en diferentes árboles, entonces puede especificar varios destinos explícitamente:

```
% make BINDIR=bindir LIBDIR=libdir HEADERDIR=headerdir ODBCINST=instfile install
```

Instalación Aislada

Una instalación aislada no está configurada en la distribución Postgres habitual. Debe realizarse un ajuste mejor para la construcción del driver ODBC para clientes múltiples y y heterogeneos que no tienen instalado un árbol de fuentes Postgres de forma local.

La ubicación por defecto para las librerías y ficheros de cabecera y para la instalación aislada es `/usr/local/lib` y `/usr/local/include/iodbc`, respectivamente. Existe otro fichero de configuración de sistema que se instala como `/share/odbcinst.ini` (si `/share` existe) o como `/etc/odbcinst.ini` (si `/share` no existe).

Nota: La instalación de ficheros en `/share` o `/etc` requiere privilegios de root. Muchas etapas de la instalación de Postgres no necesitan de este requerimiento, y usted puede elegir otra ubicación en que su cuenta de superusuario Postgres tenga permisos de escritura.

1. La instalación de la distribución aislada puede realizarse desde la distribución Postgres o puede ser obtenida a través de Insight Distributors⁶, los mantenedores actuales para distribuciones no Unix.

Copie el fichero zip o el fichero tar comprimido en un directorio vacío. Si usa el paquete zip, descomprímalo con el comando

```
% unzip -a packagename
```

La opción `-a` es necesaria para deshacerse de los pares CR/LF de DOS en los ficheros fuente

Si tiene el paquete tar comprimido, simplemente ejecute

```
tar -xzf packagename
```

- a. Para crear un fichero tar para una instalación aislada completa desde el árbol principal de fuentes de Postgres:
2. Configure la distribución principal Postgres.
 3. Cree el fichero tar:

```
% cd interfaces/odbc
% make standalone
```

4. Copie el fichero tar de salida al sistema de destino. Asegúrese de transferirlo como un fichero binario usando ftp.
5. Desempaque el fichero tar en un directorio vacío.
6. Configure la instalación aislada:

```
% ./configure
```

La configuración puede realizarse con las opciones:

```
% ./configure -prefix=rootdir -with-odbc=inidir
```

donde `-prefix` instala las bibliotecas y ficheros de cabecera en los directorios `rootdir/lib` y `rootdir/include/iolib`, y `-with-odbc` instala `odbcinst.ini` en el directorio especificado.

Nótese que ambas opciones se pueden usar desde la construcción integrada pero tenga en cuenta *que cuando se usan en la construcción integrada* `-prefix` también se aplicará al resto de su instalación Postgres. `-with-odbc` se aplica sólo al fichero de configuración `odbcinst.ini`.

7. Compile and link the source code:

```
% make ODBCINST=instdir
```

También puede obviar la ubicación por defecto en la instalación en la línea de comandos de 'make'. Esto sólo se aplica a la instalación de las librerías y los ficheros de cabecera. Desde que el driver necesita saber la ubicación del fichero `odbcinst.ini` el intento de sustituir la variable de que especifica el directorio de

instalación probablemente le causará quebraderos de cabeza. Es más seguro y simple permitir al driver que instale el fichero `odbcinst.ini` en el directorio por defecto o el directorio especificado por usted en la línea de comandos de la orden `'./configure'` con `-with-odbc`.

8. Instala el código fuente:

```
% make PGRESDIR=targettree install
```

Para sustituir la librería y los directorios principales de instalación por separado necesita pasar las variables de instalación correctas en la línea de argumentos `make install`. Estas variables son `LIBDIR`, `HEADERDIR` and `ODBCINST`. Sustituyendo `PGRESDIR` en la línea de argumentos de `make` se originará que `LIBDIR` y `HEADERDIR` puedan ser ubicados en el nuevo directorio que usted especifique. `ODBCINST` es independiente de `PGRESDIR`.

Aquí es donde usted podrían especificar varios destinos explícitamente:

```
% make BINDIR=bindir LIBDIR=libdir HEADERDIR=headerdir install
```

Por ejemplo, tecleando

```
% make PGRESDIR=/opt/psqlodbc install
```

(después de haber usado `./configure` y `make`) tendrá como consecuencia que las bibliotecas y ficheros de cabecera sean instalados en los directorios `/opt/psqlodbc/lib` y `/opt/psqlodbc/include/iodbc` respectivamente.

El comando

```
% make PGRESDIR=/opt/psqlodbc HEADERDIR=/usr/local install
```

ocasionará que las bibliotecas sean instaladas en `/opt/psqlodbc/lib` y los ficheros de cabecera en `/usr/local/include/iodbc`. Si esto no funciona como se espera por favor contacte con los mantenedores.

Ficheros de Configuración

`~/.odbc.ini` contiene información de acceso específica de usuario para el driver `psqlODBC`. El fichero usa convenciones típicas de los archivos de Registro de Windows, pero a pesar de esta restricción puede hacerse funcionar.

El fichero `.odbc.ini` tiene tres secciones requeridas. La primera es `[ODBC Data Sources]` la cual es una lista de nombres arbitrarios y descripciones para cada base de datos a la que desee acceder. La segunda sección es la denominada `Data Source Specification` y existirá una de estas secciones por cada base de datos. Cada sección debe ser etiquetada con el nombre dado en `[ODBC Data Sources]` y debe contener las siguientes entradas:

```
Driver = PGRESDIR/lib/libpsqlodbc.so
Database=DatabaseName
Servername=localhost
Port=5432
```

Sugerencia: Recuerde que el nombre de bases de datos Postgres es por lo general una palabra sencilla, sin nombres de trayectoria ni otras cosas. El servidor Postgres gestiona el acceso actual a la base de datos, y sólo necesita especificar el nombre desde el cliente.

Se pueden insertar otras entradas para controlar el formato de visualización. La tercera sección necesaria es [ODBC] la cual debe contener la palabra clave `InstallDir` además de otras opciones.

He aquí un fichero `.odbc.ini` de ejemplo, que muestra la información de acceso para tres bases de datos:

```
[ODBC Data Sources]
DataEntry = Read/Write Database
QueryOnly = Read-only Database
Test = Debugging Database
Default = Postgres Stripped

[DataEntry]
ReadOnly = 0
Servername = localhost
Database = Sales

[QueryOnly]
ReadOnly = 1
Servername = localhost
Database = Sales

[Test]
Debug = 1
CommLog = 1
ReadOnly = 0
Servername = localhost
Username = tgl
Password = "no$way"
Port = 5432
Database = test

[Default]
Servername = localhost
Database = tgl
Driver = /opt/postgres/current/lib/libpsqlodbc.so

[ODBC]
InstallDir = /opt/applix/axdata/axshlib
```

ApplixWare

Configuration

ApplixWare debe ser configurado correctamente para que pueda realizarse con él el acceso a los drivers ODBC de Postgres.

Habilitando el acceso a bases de datos con ApplixWare

Estas instrucciones son para la versión 4.4.1 de ApplixWare en Linux. Véase el libro on-line *Administración de Sistemas Linux* para información más detallada.

1. Debe modificar el fichero `axnet.cnf` para que `elfodbc` pueda encontrar la biblioteca compartida `libodbc.so` (el administrador de dispositivos ODBC). Esta biblioteca viene incluida con la distribución de Applixware, pero el fichero `axnet.cnf` necesita modificarse para que apunte a la ubicación correcta.

Como root, edite el fichero `applixroot/applix/axdata/axnet.cnf`.

- a. Al final del fichero `axnet.cnf`, y busque la línea que comienza con
`#libFor elfodbc /ax/...`
 - b. Cambie la línea para que se lea
`libFor elfodbc applixroot/applix/axdata/axshlib/lib`
 lo cual le dirá a `elfodbc` que busque en este directorio para la librería de soporte ODBC. Si ha instalado Applix en cualquier otro sitio, cambie la trayectoria en consecuencia.
2. Cree el fichero `.odbc.ini` como se describió anteriormente. También puede querer añadir el indicador
`TextAsLongVarchar=0`
 a la porción específica de bases de datos de `.odbc.ini` para que los campos de texto no sean mostrados como `**BLOB**`.

Probando las conexiones ODBC de ApplixWare

1. Ejecute `Applix Data`
2. Seleccione la base de datos Postgres de su interés.
 - a. Seleccione **Query->Choose Server**.
 - b. Seleccione ODBC, y haga click en **Browse**. La base de datos que configuró en el fichero `.odbc.ini` debería mostrarse. Asegúrese de que `Host:field` está vacío (si no es así, `axnet` intentará contactar con `axnet` en otra máquina para buscar la base de datos).
 - c. Seleccione la base de datos en la caja de diálogo que ha sido lanzada por **Browse**, entonces haga click en **OK**.
 - d. Introduzca el nombre de usuario y contraseña en la caja de diálogo de identificación, y haga click en **OK**.

Debería ver "Starting elfodbc server" en la esquina inferior izquierda de la ventana de datos. Si aparece una ventana de error, vea la sección de depuración de abajo.
3. El mensaje 'Ready' aparecerá en la esquina inferior izquierda de la ventana de datos. Esto indica que a partir de ahora se pueden realizar consultas.
4. Seleccione una tabla desde **Query->Choose tables**, y entonces seleccione **Query->Query** para acceder a la base de datos. Las primeras 50 filas de la tabla más o menos deberían aparecer.

Problemas Comunes

Los siguientes mensajes pueden aparecer a la hora de intentar una conexión ODBC a través de Applix Data:

No puedo lanzar pasarela en el servidor

`elfodbc` no puede encontrar `libodbc.so`. Chequee su fichero `axnet.cnf`.

Error de pasarela ODBC: IM003::[iODBC][Driver Manager] El driver especificado no pudo cargarse

`libodbc.so` no puedo encontrar el driver especificado en `.odbc.ini`. Verifique los ajustes.

Servidor: Tubería rota

El proceso del driver ha terminado debido a algún problem. Puede que no tenga una versión actualizada del paquete ODBC de Postgres .

setuid a 256: fallo al lanzar la pasarela

La versión de septiembre de ApplixWare v4.4.1 (la primera versión con soporte oficial de ODBC bajo Linux) presenta problemas con nombres de usuario que exceden los ocho (8) caracteres de longitud. Descripción del problema contribuida por Steve Campbell⁷.

Author: Contribuido por Steve Campbell⁸ on 1998-10-20.

El sistema de seguridad del programa `axnet` parece un poco sospechoso. `axnet` hace cosas en nombre del usuario y en un sistema de múltiples usuarios de verdad debería ejecutarse con seguridad de root (de este modo puede leer/escribir en cada directorio de usuario). Debería dudar al recomendar esto, sin embargo, ya que no tengo idea de qué tipo de hoyos de seguridad provoca esto.

Depurando las conexiones ODBC ApplixWare

Una buena herramienta para la depuración de problemas de conexión usa el la aplicación del sistema Unix `strace`.

Depurando con `strace`

1. Start applixware.
2. Inicie un comando `strace` en el proceso `axnet`. Por ejemplo, si

```
ps -auxx | grep ax
```

shows

```
cary  10432  0.0  2.6  1740   392  ?  S   Oct  9  0:00 axnet
cary  27883  0.9 31.0 12692  4596  ?  S   10:24  0:04 axmain
```

Entonces ejecute

```
strace -f -s 1024 -p 10432
```

3. Compruebe la salida de strace.

Nota de Cary: Muchos de los mensajes de error de ApplixWare van hacia `stderr`, pero no estoy seguro de a dónde está dirigido `stderr`, así que `strace` es la manera de encontrarlo.

Por ejemplo, después de obtener el mensaje “Cannot launch gateway on server”, ejecuto `strace` en `axnet` y obtengo

```
[pid 27947] open("/usr/lib/libodbc.so", O_RDONLY) = -1 ENOENT
          (No existe el fichero o directorio)
[pid 27947] open("/lib/libodbc.so", O_RDONLY) = -1 ENOENT
          (No existe el fichero o directorio)
[pid 27947] write(2, "/usr2/applix/axdata/elfodbc:
          no puedo cargar la biblioteca 'libodbc.so'\n", 61) = -1 EIO (I/O error)
```

Así que lo que ocurre es que `elfodbc` de `applix` está buscando `libodbc.so`, pero no puede encontrarlo. Por eso es por lo que `axnet.cnf` necesita cambiarse.

Ejecutando la demo ApplixWare

Para poder ir a través de *ApplixWare Data Tutorial*, necesita crear las tablas de ejemplo a las que se refiere el Tutorial. La macro ELF Macro usada para crear las tablas intenta crear una condición NULL de algunas de las columnas de la base de datos y Postgres no permite esta opción por lo general.

Para bordear este problema, puede hacer lo siguiente:

Modificando la Demo ApplixWare

1. Copie `/opt/applix/axdata/eng/Demos/sqldemo.am` a un directorio local.
2. Edite esta copia local de `sqldemo.am`:
 - a. Busque `'null_clause = "NULL"`
 - b. Cámbielo a `null_clause = ""`
3. Inicie `Applix Macro Editor`.
4. Abra el fichero `sqldemo.am` desde el `Macro Editor`.
5. Seleccione **File->Compile and Save**.
6. Salga del `Macro Editor`.
7. Inicie `Applix Data`.
8. Seleccione ***->Run Macro**
9. Introduzca el valor “`sqldemo`”, entonces haga click en **OK**.
Debería ver el progreso en la línea de estado en la ventana de datos (en la esquina inferior izquierda).
10. Ahora debería ser capaz de acceder a las tablas demo.

Useful Macros

Puede añadir información sobre el usuario y contraseña para la base de datos en el fichero de macro de inicio estándar de Applix. Este es un fichero `~/axhome/macros/login.am` de ejemplo:

```
macro login
    set_set_system_var@("sql_username@", "tgl")
    set_system_var@("sql_passwd@", "no$way")
endmacro
```

Atención

Deberá tener cuidado con las protecciones de fichero en cualquier fichero que contenga información de nombres de usuario y contraseñas.

Plataformas soportadas

psqlODBC ha sido compilado y probado en Linux. Han sido reportados éxitos con FreeBSD y Solaris. No se conocen restricciones para otras plataformas que soporten Postgres.

Notas

1. <mailto:tgoeke@xpressway.com>
2. <http://www.insightdist.com/>
3. <http://www.insightdist.com/psqlodbc/>
4. <mailto:interfaces@postgresql.org>
5. <http://www.as220.org/FreeODBC/iodbc-2.12.shar.Z>
6. <http://www.insightdist.com/psqlodbc>
7. <mailto:scampbell@lear.com>
8. <mailto:scampbell@lear.com>

Capítulo 50. JDBC Interface

Author: Written by Peter T. Mount¹, the author of the JDBC driver.

JDBC is a core API of Java 1.1 and later. It provides a standard set of interfaces to SQL-compliant databases.

Postgres provides a *type 4* JDBC Driver. Type 4 indicates that the driver is written in Pure Java, and communicates in the database's own network protocol. Because of this, the driver is platform independent. Once compiled, the driver can be used on any platform.

Building the JDBC Interface

Compiling the Driver

The driver's source is located in the `src/interfaces/jdbc` directory of the source tree. To compile simply change directory to that directory, and type:

```
% make
```

Upon completion, you will find the archive `postgresql.jar` in the current directory. This is the JDBC driver.

Nota: You must use `make`, not `javac`, as the driver uses some dynamic loading techniques for performance reasons, and `javac` cannot cope. The `Makefile` will generate the jar archive.

Installing the Driver

To use the driver, the jar archive `postgresql.jar` needs to be included in the CLASSPATH.

Example

I have an application that uses the JDBC driver to access a large database containing astronomical objects. I have the application and the jdbc driver installed in the `/usr/local/lib` directory, and the java jdk installed in `/usr/local/jdk1.1.6`.

To run the application, I would use:

```
export CLASSPATH = /usr/local/lib/finder.jar:/usr/local/lib/postgresql.jar:.
java uk.org.retep.finder.Main
```

Loading the driver is covered later on in this chapter.

Preparing the Database for JDBC

Because Java can only use TCP/IP connections, the `Postgres` postmaster must be running with the `-i` flag.

Also, the `pg_hba.conf` file must be configured. It's located in the `PGDATA` directory. In a default installation, this file permits access only by Unix domain sockets. For the JDBC driver to connect to the same localhost, you need to add something like:

```
host          all          127.0.0.1          255.255.255.255    password
```

Here access to all databases are possible from the local machine with JDBC.

The JDBC Driver supports trust, ident, password and crypt authentication methods.

Using the Driver

This section is not intended as a complete guide to JDBC programming, but should help to get you started. For more information refer to the standard JDBC API documentation. Also, take a look at the examples included with the source. The basic example is used here.

Importing JDBC

Any source that uses JDBC needs to import the `java.sql` package, using:

```
import java.sql.*;
```

Importante: Do not import the `postgresql` package. If you do, your source will not compile, as `javac` will get confused.

Loading the Driver

Before you can connect to a database, you need to load the driver. There are two methods available, and it depends on your code to the best one to use.

In the first method, your code implicitly loads the driver using the `Class.forName()` method. For `Postgres`, you would use:

```
Class.forName("postgresql.Driver");
```

This will load the driver, and while loading, the driver will automatically register itself with JDBC.

Note: The `forName()` method can throw a `ClassNotFoundException`, so you will need to catch it if the driver is not available.

This is the most common method to use, but restricts your code to use just `Postgres`. If your code may access another database in the future, and you don't use our extensions, then the second method is advisable.

The second method passes the driver as a parameter to the JVM as it starts, using the `-D` argument. Example:

```
% java -Djdbc.drivers=postgresql.Driver example.ImageViewer
```

In this example, the JVM will attempt to load the driver as part of its initialisation. Once done, the `ImageViewer` is started.

Now, this method is the better one to use because it allows your code to be used with other databases, without recompiling the code. The only thing that would also change is the URL, which is covered next.

One last thing. When your code then tries to open a `Connection`, and you get a `No driver available SQLException` being thrown, this is probably caused by the driver not being in the classpath, or the value in the parameter not being correct.

Connecting to the Database

With JDBC, a database is represented by a URL (Uniform Resource Locator). With `Postgres`, this takes one of the following forms:

- `jdbc:postgresql:database`
- `jdbc:postgresql://>hos>/database`
- `jdbc:postgresql://>hos>">poe>/database`

where:

host

The hostname of the server. Defaults to "localhost".

port

The port number the server is listening on. Defaults to the `Postgres` standard port number (5432).

database

The database name.

To connect, you need to get a `Connection` instance from JDBC. To do this, you would use the `DriverManager.getConnection()` method:

```
Connection db = DriverManager.getConnection(url,user,pwd);
```

Issuing a Query and Processing the Result

Any time you want to issue SQL statements to the database, you require a `Statement` instance. Once you have a `Statement`, you can use the `executeQuery()` method to issue a query. This will return a `ResultSet` instance, which contains the entire result.

Using the Statement Interface

The following must be considered when using the `Statement` interface:

- You can use a `Statement` instance as many times as you want. You could create one as soon as you open the connection, and use it for the connections lifetime. You have to remember that only one `ResultSet` can exist per `Statement`.
- If you need to perform a query while processing a `ResultSet`, you can simply create and use another `Statement`.
- If you are using `Threads`, and several are using the database, you must use a separate `Statement` for each thread. Refer to the sections covering `Threads` and `Servlets` later in this document if you are thinking of using them, as it covers some important points.

Using the ResultSet Interface

The following must be considered when using the `ResultSet` interface:

- Before reading any values, you must call `next()`. This returns `true` if there is a result, but more importantly, it prepares the row for processing.
- Under the JDBC spec, you should access a field only once. It's safest to stick to this rule, although at the current time, the `Postgres` driver will allow you to access a field as many times as you want.
- You must close a `ResultSet` by calling `close()` once you have finished with it.
- Once you request another query with the `Statement` used to create a `ResultSet`, the currently open instance is closed.

An example is as follows:

```
Statement st = db.createStatement();
ResultSet rs = st.executeQuery("select * from mytable");
while(rs.next()) {
    System.out.print("Column 1 returned ");
    System.out.println(rs.getString(1));
}
rs.close();
st.close();
```

Performing Updates

To perform an update (or any other SQL statement that does not return a result), you simply use the `executeUpdate()` method:

```
st.executeUpdate("create table basic (a int2, b int2)");
```

Closing the Connection

To close the database connection, simply call the `close()` method to the `Connection`:

```
db.close();
```

Using Large Objects

In `Postgres`, large objects (also known as *blobs*) are used to hold data in the database that cannot be stored in a normal SQL table. They are stored as a Table/Index pair, and are referred to from your own tables by an `OID` value.

Importante: For `Postgres`, you must access large objects within an SQL transaction. Although this has always been true in principle, it was not strictly enforced until the release of v6.5. You would open a transaction by using the `setAutoCommit()` method with an input parameter of `false`:

```
Connection mycon;
...
mycon.setAutoCommit(false);
... now use Large Objects
```

Now, there are two methods of using Large Objects. The first is the standard JDBC way, and is documented here. The other, uses our own extension to the api, which presents the `libpq` large object API to Java, providing even better access to large objects than the standard. Internally, the driver uses the extension to provide large object support.

In JDBC, the standard way to access them is using the `getBinaryStream()` method in `ResultSet`, and `setBinaryStream()` method in `PreparedStatement`. These methods make the large object appear as a Java stream, allowing you to use the `java.io` package, and others, to manipulate the object.

For example, suppose you have a table containing the file name of an image, and a large object containing that image:

```
create table images (imgname name,imgoid oid);
```

To insert an image, you would use:

```
File file = new File("myimage.gif");
FileInputStream fis = new FileInputStream(file);
PreparedStatement ps = conn.prepareStatement("insert into images values (?,?)");
ps.setString(1,file.getName());
ps.setBinaryStream(2,fis,file.length());
ps.executeUpdate();
ps.close();
fis.close();
```

Now in this example, `setBinaryStream` transfers a set number of bytes from a stream into a large object, and stores the OID into the field holding a reference to it.

Retrieving an image is even easier (I'm using `PreparedStatement` here, but `Statement` can equally be used):

```
PreparedStatement ps = con.prepareStatement("select oid from images where name=?");
ps.setString(1,"myimage.gif");
ResultSet rs = ps.executeQuery();
if(rs!=null) {
    while(rs.next()) {
        InputStream is = rs.getBinaryInputStream(1);
        // use the stream in some way here
        is.close();
    }
    rs.close();
}
ps.close();
```

Now here you can see where the Large Object is retrieved as an `InputStream`. You'll also notice that we close the stream before processing the next row in the result. This is part of the JDBC Specification, which states that any `InputStream` returned is closed when `ResultSet.next()` or `ResultSet.close()` is called.

Postgres Extensions to the JDBC API

Postgres is an extensible database system. You can add your own functions to the backend, which can then be called from queries, or even add your own data types.

Now, as these are facilities unique to us, we support them from Java, with a set of extension API's. Some features within the core of the standard driver actually use these extensions to implement Large Objects, etc.

Accessing the extensions

To access some of the extensions, you need to use some extra methods in the `postgresql.Connection` class. In this case, you would need to case the return value of `Driver.getConnection()`.

For example:

```

    Connection db = Driver.getConnection(url,user,pass);

    // later on
    Fastpath fp = ((postgresql.Connection)db).getFastpathAPI();

```

Class postgresql.Connection

```

java.lang.Object
|
+---postgresql.Connection

```

public class Connection extends Object implements Connection

These are the extra methods used to gain access to our extensions. I have not listed the methods defined by java.sql.Connection.

public Fastpath getFastpathAPI() throws SQLException

This returns the Fastpath API for the current connection.

NOTE: This is not part of JDBC, but allows access to functions on the postgresql backend itself.

It is primarily used by the LargeObject API

The best way to use this is as follows:

```

import postgresql.fastpath.*;
...
Fastpath fp = ((postgresql.Connection)myconn).getFastpathAPI();

```

where myconn is an open Connection to postgresql.

Returns:

Fastpath object allowing access to functions on the postgresql backend.

Throws: SQLException
by Fastpath when initialising for first time

public LargeObjectManager getLargeObjectAPI() throws SQLException

This returns the LargeObject API for the current connection.

NOTE: This is not part of JDBC, but allows access to functions on the postgresql backend itself.

The best way to use this is as follows:

```

import postgresql.largeobject.*;
...
LargeObjectManager lo =
((postgresql.Connection)myconn).getLargeObjectAPI();

```

where myconn is an open Connection to postgresql.

Returns:

LargeObject object that implements the API

Throws: SQLException
by LargeObject when initialising for first time

public void addDataType(String type,

```
String name)
```

This allows client code to add a handler for one of postgresql's more unique data types. Normally, a data type not known by the driver is returned by `ResultSet.getObject()` as a `PGObject` instance.

This method allows you to write a class that extends `PGObject`, and tell the driver the type name, and class name to use.

The down side to this, is that you must call this method each time a connection is made.

NOTE: This is not part of JDBC, but an extension.

The best way to use this is as follows:

```
...
((postgresql.Connection)myconn).addDataType("mytype","my.class.name"-
);
...
```

where `myconn` is an open `Connection` to `postgresql`.

The handling class must extend `postgresql.util.PGObject`

See Also:

`PGObject`

Fastpath

Fastpath is an API that exists within the `libpq` C interface, and allows a client machine to execute a function on the database backend. Most client code will not need to use this method, but it's provided because the Large Object API uses it.

To use, you need to import the `postgresql.fastpath` package, using the line:

```
import postgresql.fastpath.*;
```

Then, in your code, you need to get a `FastPath` object:

```
Fastpath fp = ((postgresql.Connection)conn).getFastpathAPI();
```

This will return an instance associated with the database connection that you can use to issue commands. The casing of `Connection` to `postgresql.Connection` is required, as the `getFastpathAPI()` is one of our own methods, not JDBC's.

Once you have a `Fastpath` instance, you can use the `fastpath()` methods to execute a backend function.

Class `postgresql.fastpath.Fastpath`

```
java.lang.Object
|
+---postgresql.fastpath.Fastpath

public class Fastpath

extends Object
```

This class implements the `Fastpath` api.

This is a means of executing functions imbedded in the postgresql backend from within a java application.

It is based around the file `src/interfaces/libpq/fe-exec.c`

See Also:

`FastpathFastpathArg`, `LargeObject`

Methods

```
public Object fastpath(int fnid,
                       boolean resulttype,
                       FastpathArg args[]) throws SQLException
```

Send a function call to the PostgreSQL backend

Parameters:

`fnid` - Function id
`resulttype` - True if the result is an integer, false
for other results
`args` - `FastpathArguments` to pass to fastpath

Returns:

byte[] null if no data, Integer if an integer result, or
otherwise

Throws: SQLException

if a database-access error occurs.

```
public Object fastpath(String name,
                       boolean resulttype,
                       FastpathArg args[]) throws SQLException
```

Send a function call to the PostgreSQL backend by name.

Note:

the mapping for the procedure name to function id needs to exist, usually to an earlier call to `addfunction()`. This is the preferred method to call, as function id's can/may change between versions of the backend. For an example of how this works, refer to `postgresql.LargeObject`

Parameters:

`name` - Function name
`resulttype` - True if the result is an integer, false
for other results
`args` - `FastpathArguments` to pass to fastpath

Returns:

byte[] null if no data, Integer if an integer result, or
otherwise

Throws: SQLException

occurs. if name is unknown or if a database-access error

See Also:

`LargeObject`

```
public int getInteger(String name,
                     FastpathArg args[]) throws SQLException
```

This convenience method assumes that the return value is an Integer

Parameters:
 name - Function name
 args - Function arguments

Returns:
 integer result

Throws: SQLException
 if a database-access error occurs or no result

```
public byte[] getData(String name,
                     FastpathArg args[]) throws SQLException
```

This convenience method assumes that the return value is binary data

Parameters:
 name - Function name
 args - Function arguments

Returns:
 byte[] array containing result

Throws: SQLException
 if a database-access error occurs or no result

```
public void addFunction(String name,
                      int fnid)
```

This adds a function to our lookup table.

User code should use the addFunctions method, which is based upon a query, rather than hard coding the oid. The oid for a function is not guaranteed to remain static, even on different servers of the same version.

Parameters:
 name - Function name
 fnid - Function id

```
public void addFunctions(ResultSet rs) throws SQLException
```

This takes a ResultSet containing two columns. Column 1 contains the function name, Column 2 the oid.

It reads the entire ResultSet, loading the values into the function table.

REMEMBER to close() the resultset after calling this!!

Implementation note about function name lookups:

PostgreSQL stores the function id's and their corresponding names in the pg_proc table. To speed things up locally, instead of querying each function from that table when required, a Hashtable is used. Also, only the function's required are entered into this table,

keeping connection times as fast as possible.

The `postgresql.LargeObject` class performs a query upon it's startup, and passes the returned `ResultSet` to the `addFunctions()` method here.

Once this has been done, the `LargeObject` api refers to the functions by name.

Dont think that manually converting them to the oid's will work. Ok, they will for now, but they can change during development (there was some discussion about this for V7.0), so this is implemented to prevent any unwarranted headaches in the future.

Parameters:
rs - `ResultSet`

Throws: `SQLException`
if a database-access error occurs.

See Also:
`LargeObjectManager`

```
public int getID(String name) throws SQLException
```

This returns the function id associated by its name

If `addFunction()` or `addFunctions()` have not been called for this name, then an `SQLException` is thrown.

Parameters:
name - Function name to lookup

Returns:
Function ID for fastpath call

Throws: `SQLException`
is function is unknown.

Class `postgresql.fastpath.FastpathArg`

```
java.lang.Object
|
+---postgresql.fastpath.FastpathArg
```

```
public class FastpathArg extends Object
```

Each fastpath call requires an array of arguments, the number and type dependent on the function being called.

This class implements methods needed to provide this capability.

For an example on how to use this, refer to the `postgresql.largeobject` package

See Also:
`Fastpath`, `LargeObjectManager`, `LargeObject`

Constructors

```
public FastpathArg(int value)
```

Constructs an argument that consists of an integer value

```

        Parameters:
            value - int value to set

public FastpathArg(byte bytes[])

        Constructs an argument that consists of an array of bytes

        Parameters:
            bytes - array to store

public FastpathArg(byte buf[],
                    int off,
                    int len)

        Constructs an argument that consists of part of a byte
array

        Parameters:
            buf - source array
            off - offset within array
            len - length of data to include

public FastpathArg(String s)

        Constructs an argument that consists of a String.

        Parameters:
            s - String to store

```

Geometric Data Types

PostgreSQL has a set of datatypes that can store geometric features into a table. These range from single points, lines, and polygons.

We support these types in Java with the `postgresql.geometric` package.

It contains classes that extend the `postgresql.util.PGObject` class. Refer to that class for details on how to implement your own data type handlers.

Class `postgresql.geometric.PGbox`

```

java.lang.Object
|
+---postgresql.util.PGObject
|
+---postgresql.geometric.PGbox

```

```

    public class PGbox extends PGObject implements Serializable,
Cloneable

```

This represents the box datatype within postgresql.

Variables

```

public PGpoint point[]

```

These are the two corner points of the box.

Constructors

```

public PGbox(double x1,

```

```

        double y1,
        double x2,
        double y2)

Parameters:
    x1 - first x coordinate
    y1 - first y coordinate
    x2 - second x coordinate
    y2 - second y coordinate

public PGbox(PGpoint p1,
             PGpoint p2)

Parameters:
    p1 - first point
    p2 - second point

public PGbox(String s) throws SQLException

Parameters:
    s - Box definition in PostgreSQL syntax

Throws: SQLException
    if definition is invalid

public PGbox()

    Required constructor

Methods

public void setValue(String value) throws SQLException

    This method sets the value of this object. It should be
    overridden, but still called by subclasses.

Parameters:
    value - a string representation of the value of the
object
Throws: SQLException
    thrown if value is invalid for this type

Overrides:
    setValue in class PGObject

public boolean equals(Object obj)

Parameters:
    obj - Object to compare with

Returns:
    true if the two boxes are identical

Overrides:
    equals in class PGObject

public Object clone()

    This must be overridden to allow the object to be cloned

Overrides:
    clone in class PGObject

```

```
public String getValue()
```

Returns:

the PGbox in the syntax expected by postgresql

Overrides:

getValue in class PGObject

Class postgresql.geometric.PGcircle

```
java.lang.Object
```

```
|
```

```
+---postgresql.util.PGObject
```

```
|
```

```
+---postgresql.geometric.PGcircle
```

```
public class PGcircle extends PGObject implements Serializable,
Cloneable
```

This represents postgresql's circle datatype, consisting of a point and a radius

Variables

```
public PGpoint center
```

This is the centre point

```
public double radius
```

This is the radius

Constructors

```
public PGcircle(double x,
double y,
double r)
```

Parameters:

x - coordinate of centre

y - coordinate of centre

r - radius of circle

```
public PGcircle(PGpoint c,
double r)
```

Parameters:

c - PGpoint describing the circle's centre

r - radius of circle

```
public PGcircle(String s) throws SQLException
```

Parameters:

s - definition of the circle in PostgreSQL's syntax.

Throws: SQLException

on conversion failure

```
public PGcircle()
```

This constructor is used by the driver.

Methods

```

public void setValue(String s) throws SQLException

    Parameters:
        s - definition of the circle in PostgreSQL's syntax.

    Throws: SQLException
        on conversion failure

    Overrides:
        setValue in class PGObject

public boolean equals(Object obj)

    Parameters:
        obj - Object to compare with

    Returns:
        true if the two boxes are identical

    Overrides:
        equals in class PGObject

public Object clone()

    This must be overridden to allow the object to be cloned

    Overrides:
        clone in class PGObject

public String getValue()

    Returns:
        the PGcircle in the syntax expected by postgresql

    Overrides:
        getValue in class PGObject

```

Class postgresql.geometric.PGline

```

java.lang.Object
|
+---postgresql.util.PGObject
|
+---postgresql.geometric.PGline

```

```

public class PGline extends PGObject implements Serializable,
Cloneable

```

This implements a line consisting of two points. Currently line is not yet implemented in the backend, but this class ensures that when it's done were ready for it.

Variables

```

public PGpoint point[]

    These are the two points.

```

Constructors

```

public PGline(double x1,
              double y1,

```

```

        double x2,
        double y2)

    Parameters:
        x1 - coordinate for first point
        y1 - coordinate for first point
        x2 - coordinate for second point
        y2 - coordinate for second point

    public PGline(PGpoint p1,
        PGpoint p2)

    Parameters:
        p1 - first point
        p2 - second point

    public PGline(String s) throws SQLException

    Parameters:
        s - definition of the circle in PostgreSQL's syntax.

    Throws: SQLException
        on conversion failure

    public PGline()

        reuired by the driver

Methods

    public void setValue(String s) throws SQLException

    Parameters:
        s - Definition of the line segment in PostgreSQL's
syntax

    Throws: SQLException
        on conversion failure

    Overrides:
        setValue in class PGobject

    public boolean equals(Object obj)

    Parameters:
        obj - Object to compare with

    Returns:
        true if the two boxes are identical

    Overrides:
        equals in class PGobject

    public Object clone()

        This must be overridden to allow the object to be cloned

    Overrides:
        clone in class PGobject

    public String getValue()

    Returns:

```


the PGline in the syntax expected by postgresql

Overrides:
 getValue in class PGObject

Class postgresql.geometric.PGline

```

java.lang.Object
|
+---postgresql.util.PGObject
|
+---postgresql.geometric.PGline
  
```

public class PGline extends PGObject implements Serializable,
 Cloneable

This implements a lseg (line segment) consisting of two points

Variables

```

public PGpoint point[]

    These are the two points.
  
```

Constructors

```

public PGline(double x1,
              double y1,
              double x2,
              double y2)
  
```

Parameters:

```

    x1 - coordinate for first point
    y1 - coordinate for first point
    x2 - coordinate for second point
    y2 - coordinate for second point
  
```

```

public PGline(PGpoint p1,
              PGpoint p2)
  
```

Parameters:

```

    p1 - first point
    p2 - second point
  
```

```

public PGline(String s) throws SQLException
  
```

Parameters:

```

    s - definition of the circle in PostgreSQL's syntax.
  
```

Throws: SQLException

```

    on conversion failure
  
```

```

public PGline()
  
```

required by the driver

Methods

```

public void setValue(String s) throws SQLException
  
```

Parameters:

```

    s - Definition of the line segment in PostgreSQL's
  
```

syntax

Throws: SQLException
on conversion failure

Overrides:
setValue in class PGObject

public boolean equals(Object obj)

Parameters:
obj - Object to compare with

Returns:
true if the two boxes are identical

Overrides:
equals in class PGObject

public Object clone()

This must be overridden to allow the object to be cloned

Overrides:
clone in class PGObject

public String getValue()

Returns:
the PGlseg in the syntax expected by postgresql

Overrides:
getValue in class PGObject

Class postgresql.geometric.PGpath

java.lang.Object

```

|
+---postgresql.util.PGObject
|
+---postgresql.geometric.PGpath

```

public class PGpath extends PGObject implements Serializable, Cloneable

This implements a path (a multiple segmented line, which may be closed)

Variables

public boolean open

True if the path is open, false if closed

public PGpoint points[]

The points defining this path

Constructors

public PGpath(PGpoint points[],
boolean open)

```

    Parameters:
        points - the PGpoints that define the path
        open - True if the path is open, false if closed

public PGpath()

    Required by the driver

public PGpath(String s) throws SQLException

    Parameters:
        s - definition of the circle in PostgreSQL's syntax.

    Throws: SQLException
        on conversion failure

Methods

public void setValue(String s) throws SQLException

    Parameters:
        s - Definition of the path in PostgreSQL's syntax

    Throws: SQLException
        on conversion failure

    Overrides:
        setValue in class PGObject

public boolean equals(Object obj)

    Parameters:
        obj - Object to compare with

    Returns:
        true if the two boxes are identical

    Overrides:
        equals in class PGObject

public Object clone()

    This must be overridden to allow the object to be cloned

    Overrides:
        clone in class PGObject

public String getValue()

    This returns the polygon in the syntax expected by
    postgresql

    Overrides:
        getValue in class PGObject

public boolean isOpen()

    This returns true if the path is open

public boolean isClosed()

    This returns true if the path is closed

```

```

public void closePath()

    Marks the path as closed

public void openPath()

    Marks the path as open

Class postgresql.geometric.PGpoint
java.lang.Object
|
+---postgresql.util.PGobject
|
+---postgresql.geometric.PGpoint

    public class PGpoint extends PGobject implements Serializable,
Cloneable

    This implements a version of java.awt.Point, except it uses double
to represent the coordinates.

    It maps to the point datatype in postgresql.

Variables

    public double x

        The X coordinate of the point

    public double y

        The Y coordinate of the point

Constructors

    public PGpoint(double x,
double y)

        Parameters:
            x - coordinate
            y - coordinate

    public PGpoint(String value) throws SQLException

        This is called mainly from the other geometric types, when a
point is imbedded within their definition.

        Parameters:
            value - Definition of this point in PostgreSQL's
syntax

    public PGpoint()

        Required by the driver

Methods

    public void setValue(String s) throws SQLException

        Parameters:
            s - Definition of this point in PostgreSQL's syntax

```

```

    Throws: SQLException
           on conversion failure

    Overrides:
        setValue in class PGObject

    public boolean equals(Object obj)

    Parameters:
        obj - Object to compare with

    Returns:
        true if the two boxes are identical

    Overrides:
        equals in class PGObject

    public Object clone()

        This must be overridden to allow the object to be cloned

    Overrides:
        clone in class PGObject

    public String getValue()

    Returns:
        the PGpoint in the syntax expected by postgresql

    Overrides:
        getValue in class PGObject

    public void translate(int x,
                        int y)

        Translate the point with the supplied amount.

    Parameters:
        x - integer amount to add on the x axis
        y - integer amount to add on the y axis

    public void translate(double x,
                        double y)

        Translate the point with the supplied amount.

    Parameters:
        x - double amount to add on the x axis
        y - double amount to add on the y axis

    public void move(int x,
                    int y)

        Moves the point to the supplied coordinates.

    Parameters:
        x - integer coordinate
        y - integer coordinate

    public void move(double x,
                    double y)

        Moves the point to the supplied coordinates.

```

```

Parameters:
    x - double coordinate
    y - double coordinate

public void setLocation(int x,
                       int y)

    Moves the point to the supplied coordinates. refer to
    java.awt.Point for description of this

Parameters:
    x - integer coordinate
    y - integer coordinate

See Also:
    Point

public void setLocation(Point p)

    Moves the point to the supplied java.awt.Point refer to
    java.awt.Point for description of this

Parameters:
    p - Point to move to

See Also:
    Point

Class postgresql.geometric.PGpolygon
java.lang.Object
|
+---postgresql.util.PGobject
|
+---postgresql.geometric.PGpolygon

    public class PGpolygon extends PGobject implements Serializable,
    Cloneable

    This implements the polygon datatype within PostgreSQL.

Variables

    public PGpoint points[]

        The points defining the polygon

Constructors

    public PGpolygon(PGpoint points[])

        Creates a polygon using an array of PGpoints

Parameters:
    points - the points defining the polygon

    public PGpolygon(String s) throws SQLException

Parameters:
    s - definition of the circle in PostgreSQL's syntax.

Throws: SQLException

```

on conversion failure

```
public PGpolygon()
```

Required by the driver

Methods

```
public void setValue(String s) throws SQLException
```

Parameters:

s - Definition of the polygon in PostgreSQL's syntax

Throws: SQLException

on conversion failure

Overrides:

setValue in class PGObject

```
public boolean equals(Object obj)
```

Parameters:

obj - Object to compare with

Returns:

true if the two boxes are identical

Overrides:

equals in class PGObject

```
public Object clone()
```

This must be overridden to allow the object to be cloned

Overrides:

clone in class PGObject

```
public String getValue()
```

Returns:

the PGpolygon in the syntax expected by postgresql

Overrides:

getValue in class PGObject

Large Objects

Large objects are supported in the standard JDBC specification. However, that interface is limited, and the api provided by PostgreSQL allows for random access to the objects contents, as if it was a local file.

The postgresql.largeobject package provides to Java the libpq C interface's large object API. It consists of two classes, `LargeObjectManager`, which deals with creating, opening and deleting large objects, and `LargeObject` which deals with an individual object.

Class `postgresql.largeobject.LargeObject`

```
java.lang.Object
```

```
|
```

```
+---postgresql.largeobject.LargeObject
```

```
public class LargeObject extends Object
```

This class implements the large object interface to postgresql.

It provides the basic methods required to run the interface, plus a pair of methods that provide InputStream and OutputStream classes for this object.

Normally, client code would use the getAsciiStream, getBinaryStream, or getUnicodeStream methods in ResultSet, or setAsciiStream, setBinaryStream, or setUnicodeStream methods in PreparedStatement to access Large Objects.

However, sometimes lower level access to Large Objects are required, that are not supported by the JDBC specification.

Refer to postgresql.largeobject.LargeObjectManager on how to gain access to a Large Object, or how to create one.

See Also:
LargeObjectManager

Variables

```
public static final int SEEK_SET
```

Indicates a seek from the beginning of a file

```
public static final int SEEK_CUR
```

Indicates a seek from the current position

```
public static final int SEEK_END
```

Indicates a seek from the end of a file

Methods

```
public int getOID()
```

Returns:
the OID of this LargeObject

```
public void close() throws SQLException
```

This method closes the object. You must not call methods in this object after this is called.

Throws: SQLException
if a database-access error occurs.

```
public byte[] read(int len) throws SQLException
```

Reads some data from the object, and return as a byte[] array

Parameters:
len - number of bytes to read

Returns:
byte[] array containing data read

Throws: SQLException


```

        if a database-access error occurs.

public void read(byte buf[],
                int off,
                int len) throws SQLException

    Reads some data from the object into an existing array

Parameters:
    buf - destination array
    off - offset within array
    len - number of bytes to read

Throws: SQLException
    if a database-access error occurs.

public void write(byte buf[]) throws SQLException

    Writes an array to the object

Parameters:
    buf - array to write

Throws: SQLException
    if a database-access error occurs.

public void write(byte buf[],
                int off,
                int len) throws SQLException

    Writes some data from an array to the object

Parameters:
    buf - destination array
    off - offset within array
    len - number of bytes to write

Throws: SQLException
    if a database-access error occurs.

public void seek(int pos,
                int ref) throws SQLException

    Sets the current position within the object.

    This is similar to the fseek() call in the standard C
    library. It allows you to have random access to the large object.

Parameters:
    pos - position within object
    ref - Either SEEK_SET, SEEK_CUR or SEEK_END

Throws: SQLException
    if a database-access error occurs.

public void seek(int pos) throws SQLException

    Sets the current position within the object.

    This is similar to the fseek() call in the standard C
    library. It allows you to have random access to the large object.

Parameters:

```

```

        pos - position within object from beginning

    Throws: SQLException
             if a database-access error occurs.

    public int tell() throws SQLException

    Returns:
             the current position within the object

    Throws: SQLException
             if a database-access error occurs.

    public int size() throws SQLException

    This method is inefficient, as the only way to find out the
    size of the object is to seek to the end, record the current position,
    then return to the original position.

    A better method will be found in the future.

    Returns:
             the size of the large object

    Throws: SQLException
             if a database-access error occurs.

    public InputStream getInputStream() throws SQLException

    Returns an InputStream from this object.

    This InputStream can then be used in any method that
    requires an InputStream.

    Throws: SQLException
             if a database-access error occurs.

    public OutputStream getOutputStream() throws SQLException

    Returns an OutputStream to this object

    This OutputStream can then be used in any method that
    requires an OutputStream.

    Throws: SQLException
             if a database-access error occurs.

    Class postgresql.largeobject.LargeObjectManager

    java.lang.Object
    |
    +---postgresql.largeobject.LargeObjectManager

    public class LargeObjectManager extends Object

    This class implements the large object interface to postgresql.

    It provides methods that allow client code to create, open and
    delete large objects from the database. When opening an object, an
    instance of postgresql.largeobject.LargeObject is returned, and its
    methods then allow access to the object.

    This class can only be created by postgresql.Connection

```

To get access to this class, use the following segment of code:

```
import postgresql.largeobject.*;
Connection conn;
LargeObjectManager lobj;
... code that opens a connection ...
lobj = ((postgresql.Connection)myconn).getLargeObjectAPI();
```

Normally, client code would use the `getAsciiStream`, `getBinaryStream`, or `getUnicodeStream` methods in `ResultSet`, or `setAsciiStream`, `setBinaryStream`, or `setUnicodeStream` methods in `PreparedStatement` to access Large Objects.

However, sometimes lower level access to Large Objects are required, that are not supported by the JDBC specification.

Refer to `postgresql.largeobject.LargeObject` on how to manipulate the contents of a Large Object.

See Also:

`LargeObject`

Variables

```
public static final int WRITE
```

This mode indicates we want to write to an object

```
public static final int READ
```

This mode indicates we want to read an object

```
public static final int READWRITE
```

This mode is the default. It indicates we want read and write access to a large object

Methods

```
public LargeObject open(int oid) throws SQLException
```

This opens an existing large object, based on its OID. This method assumes that `READ` and `WRITE` access is required (the default).

Parameters:

oid - of large object

Returns:

`LargeObject` instance providing access to the object

Throws: `SQLException`

on error

```
public LargeObject open(int oid,
                        int mode) throws SQLException
```

This opens an existing large object, based on its OID

Parameters:

oid - of large object

mode - mode of open

Returns:
 LargeObject instance providing access to the object

Throws: SQLException
 on error

public int create() throws SQLException

 This creates a large object, returning its OID.

 It defaults to READWRITE for the new object's attributes.

Returns:
 oid of new object

Throws: SQLException
 on error

public int create(int mode) throws SQLException

 This creates a large object, returning its OID

Parameters:
 mode - a bitmask describing different attributes of
the
 new object

Returns:
 oid of new object

Throws: SQLException
 on error

public void delete(int oid) throws SQLException

 This deletes a large object.

Parameters:
 oid - describing object to delete

Throws: SQLException
 on error

public void unlink(int oid) throws SQLException

 This deletes a large object.

 It is identical to the delete method, and is supplied as the C API uses unlink.

Parameters:
 oid - describing object to delete

Throws: SQLException
 on error

Object Serialisation

PostgreSQL is not a normal SQL Database. It is far more extensible than most other databases, and does support Object Oriented features that are unique to it.

One of the consequences of this, is that you can have one table refer

to a row in another table. For example:

```
test=> create table users (username name,fullname text);
CREATE
test=> create table server (servername name,adminuser users);
CREATE
test=> insert into users values ('peter','Peter Mount');
INSERT 2610132 1
test=> insert into server values ('maidast',2610132::users);
INSERT 2610133 1
test=> select * from users;
username|fullname
-----+-----
peter   |Peter Mount
(1 row)

test=> select * from server;
servername|adminuser
-----+-----
maidast   | 2610132
(1 row)
```

Ok, the above example shows that we can use a table name as a field, and the row's oid value is stored in that field.

What does this have to do with Java?

In Java, you can store an object to a Stream as long as it's class implements the `java.io.Serializable` interface. This process, known as Object Serialization, can be used to store complex objects into the database.

Now, under JDBC, you would have to use a `LargeObject` to store them. However, you cannot perform queries on those objects.

What the `postgresql.util.Serialize` class does, is provide a means of storing an object as a table, and to retrieve that object from a table. In most cases, you would not need to access this class direct, but you would use the `PreparedStatement.setObject()` and `ResultSet.getObject()` methods. Those methods will check the objects class name against the table's in the database. If a match is found, it assumes that the object is a Serialized object, and retrieves it from that table. As it does so, if the object contains other serialized objects, then it recurses down the tree.

Sound's complicated? In fact, it's simpler than what I wrote - it's just difficult to explain.

The only time you would access this class, is to use the `create()` methods. These are not used by the driver, but issue one or more "create table" statements to the database, based on a Java Object or Class that you want to serialize.

Oh, one last thing. If your object contains a line like:

```
public int oid;
```

then, when the object is retrieved from the table, it is set to the oid within the table. Then, if the object is modified, and re-serialized, the existing entry is updated.

If the oid variable is not present, then when the object is serialized, it is always inserted into the table, and any existing

entry in the table is preserved.

Setting oid to 0 before serialization, will also cause the object to be inserted. This enables an object to be duplicated in the database.

Class postgresql.util.Serialize

```
java.lang.Object
|
+---postgresql.util.Serialize

public class Serialize extends Object
```

This class uses PostgreSQL's object oriented features to store Java Objects. It does this by mapping a Java Class name to a table in the database. Each entry in this new table then represents a Serialized instance of this class. As each entry has an OID (Object Identifier), this OID can be included in another table. This is too complex to show here, and will be documented in the main documents in more detail.

Constructors

```
public Serialize(Connection c,
                  String type) throws SQLException
```

This creates an instance that can be used to serialize or deserialize a Java object from a PostgreSQL table.

Methods

```
public Object fetch(int oid) throws SQLException
```

This fetches an object from a table, given it's OID

Parameters:

oid - The oid of the object

Returns:

Object relating to oid

Throws: SQLException

on error

```
public int store(Object o) throws SQLException
```

This stores an object into a table, returning it's OID.

If the object has an int called OID, and it is > 0, then that value is used for the OID, and the table will be updated. If the value of OID is 0, then a new row will be created, and the value of OID will be set in the object. This enables an object's value in the database to be updateable. If the object has no int called OID, then the object is stored. However if the object is later retrieved, amended and stored again, it's new state will be appended to the table, and will not overwrite the old entries.

Parameters:

o - Object to store (must implement Serializable)

Returns:

oid of stored object

Throws: SQLException

on error

```
public static void create(Connection con,
                        Object o) throws SQLException
```

This method is not used by the driver, but it creates a table, given a Serializable Java Object. It should be used before serializing any objects.

Parameters:

c - Connection to database
o - Object to base table on

Throws: SQLException
on error

Returns:
Object relating to oid

Throws: SQLException
on error

```
public int store(Object o) throws SQLException
```

This stores an object into a table, returning it's OID.

If the object has an int called OID, and it is > 0, then that value is used for the OID, and the table will be updated. If the value of OID is 0, then a new row will be created, and the value of OID will be set in the object. This enables an object's value in the database to be updateable. If the object has no int called OID, then the object is stored. However if the object is later retrieved, amended and stored again, it's new state will be appended to the table, and will not overwrite the old entries.

Parameters:

o - Object to store (must implement Serializable)

Returns:
oid of stored object

Throws: SQLException
on error

```
public static void create(Connection con,
                        Object o) throws SQLException
```

This method is not used by the driver, but it creates a table, given a Serializable Java Object. It should be used before serializing any objects.

Parameters:

c - Connection to database
o - Object to base table on

Throws: SQLException
on error

```
public static void create(Connection con,
                        Class c) throws SQLException
```

This method is not used by the driver, but it creates a table, given a Serializable Java Object. It should be used before

serializing any objects.

Parameters:
 c - Connection to database
 o - Class to base table on

Throws: SQLException
 on error

public static String toPostgreSQL(String name) throws SQLException

This converts a Java Class name to a postgresql table, by replacing . with _

Because of this, a Class name may not have _ in the name.

Another limitation, is that the entire class name (including packages) cannot be longer than 31 characters (a limit forced by PostgreSQL).

Parameters:
 name - Class name

Returns:
 PostgreSQL table name

Throws: SQLException
 on error

public static String toClassName(String name) throws SQLException

This converts a postgresql table to a Java Class name, by replacing _ with .

Parameters:
 name - PostgreSQL table name

Returns:
 Class name

Throws: SQLException
 on error

Utility Classes

The postgresql.util package contains classes used by the internals of the main driver, and the other extensions.

Class postgresql.util.PGmoney

```
java.lang.Object
|
+---postgresql.util.PGobject
|
+---postgresql.util.PGmoney
```

public class PGmoney extends PGobject implements Serializable, Cloneable

This implements a class that handles the PostgreSQL money type

Variables


```
public double val
```

The value of the field

Constructors

```
public PGmoney(double value)
```

Parameters:
value - of field

```
public PGmoney(String value) throws SQLException
```

This is called mainly from the other geometric types, when a point is imbeded within their definition.

Parameters:
value - Definition of this point in PostgreSQL's

syntax

```
public PGmoney()
```

Required by the driver

Methods

```
public void setValue(String s) throws SQLException
```

Parameters:
s - Definition of this point in PostgreSQL's syntax

Throws: SQLException
on conversion failure

Overrides:
setValue in class PGobject

```
public boolean equals(Object obj)
```

Parameters:
obj - Object to compare with

Returns:
true if the two boxes are identical

Overrides:
equals in class PGobject

```
public Object clone()
```

This must be overridden to allow the object to be cloned

Overrides:
clone in class PGobject

```
public String getValue()
```

Returns:
the PGpoint in the syntax expected by postgresql

Overrides:
getValue in class PGobject

Class postgresql.util.PGobject

java.lang.Object

```

|
+---postgresql.util.PGobject

```

public class PGobject extends Object implements Serializable, Cloneable

This class is used to describe data types that are unknown by JDBC Standard.

A call to postgresql.Connection permits a class that extends this class to be associated with a named type. This is how the postgresql.geometric package operates.

ResultSet.getObject() will return this class for any type that is not recognised on having it's own handler. Because of this, any postgresql data type is supported.

Constructors

public PGobject()

This is called by postgresql.Connection.getObject() to create the object.

Methods

public final void setType(String type)

This method sets the type of this object.

It should not be extended by subclasses, hence its final

Parameters:

type - a string describing the type of the object

public void setValue(String value) throws SQLException

This method sets the value of this object. It must be overridden.

Parameters:

value - a string representation of the value of the object

Throws: SQLException

thrown if value is invalid for this type

public final String getType()

As this cannot change during the life of the object, it's final.

Returns:

the type name of this object

public String getValue()

This must be overridden, to return the value of the object, in the form required by postgresql.

Returns:

```

        the value of this object

public boolean equals(Object obj)

    This must be overridden to allow comparisons of objects

Parameters:
    obj - Object to compare with

Returns:
    true if the two boxes are identical

Overrides:
    equals in class Object

public Object clone()

    This must be overridden to allow the object to be cloned

Overrides:
    clone in class Object

public String toString()

    This is defined here, so user code need not override it.

Returns:
    the value of this object, in the syntax expected by
postgresql

    Overrides:
        toString in class Object

Class postgresql.util.PGtokenizer

java.lang.Object
|
+---postgresql.util.PGtokenizer

public class PGtokenizer extends Object

    This class is used to tokenize the text output of postgres.

    We could have used StringTokenizer to do this, however, we needed
    to handle nesting of '(' ')' '[' ']' '<' and '>' as these are used by
    the geometric data types.

    It's mainly used by the geometric classes, but is useful in parsing
    any output from custom data types output from postgresql.

    See Also:
        PGbox, PGcircle, PGLseg, PGpath, PGpoint, PGpolygon

Constructors

public PGtokenizer(String string,
                    char delim)

    Create a tokeniser.

Parameters:
    string - containing tokens
    delim - single character to split the tokens

```

Methods

```
public int tokenize(String string,
                   char delim)
```

This resets this tokenizer with a new string and/or delimiter.

Parameters:

- string - containing tokens
- delim - single character to split the tokens

```
public int getSize()
```

Returns:

- the number of tokens available

```
public String getToken(int n)
```

Parameters:

- n - Token number (0 ... getSize()-1)

Returns:

- The token value

```
public PGtokenizer tokenizeToken(int n,
                                char delim)
```

This returns a new tokenizer based on one of our tokens. The geometric datatypes use this to process nested tokens (usually PGpoint).

Parameters:

- n - Token number (0 ... getSize()-1)
- delim - The delimiter to use

Returns:

- A new instance of PGtokenizer based on the token

```
public static String remove(String s,
                           String l,
                           String t)
```

This removes the lead/trailing strings from a string

Parameters:

- s - Source string
- l - Leading string to remove
- t - Trailing string to remove

Returns:

- String without the lead/trailing strings

```
public void remove(String l,
                  String t)
```

This removes the lead/trailing strings from all tokens

Parameters:

- l - Leading string to remove
- t - Trailing string to remove

```

public static String removePara(String s)

    Removes ( and ) from the beginning and end of a string

Parameters:
    s - String to remove from

Returns:
    String without the ( or )

public void removePara()

    Removes ( and ) from the beginning and end of all tokens

Returns:
    String without the ( or )

public static String removeBox(String s)

    Removes [ and ] from the beginning and end of a string

Parameters:
    s - String to remove from

Returns:
    String without the [ or ]

public void removeBox()

    Removes [ and ] from the beginning and end of all tokens

Returns:
    String without the [ or ]

public static String removeAngle(String s)

    Removes < and > from the beginning and end of a string

Parameters:
    s - String to remove from

Returns:
    String without the < or >

public void removeAngle()

    Removes < and > from the beginning and end of all tokens

Returns:
    String without the < or >

Class postgresql.util.Serialize
This was documented earlier under Object Serialisation.

Class postgresql.util.UnixCrypt
java.lang.Object
|
+---postgresql.util.UnixCrypt
    public class UnixCrypt extends Object

```

This class provides us with the ability to encrypt passwords when sent over the network stream

Contains static methods to encrypt and compare passwords with Unix encrypted passwords.

See John Dumas's Java Crypt page for the original source.

<http://www.zeh.com/local/jfd/crypt.html>

Methods

```
public static final String crypt(String salt,
                                String original)
```

Encrypt a password given the cleartext password and a "salt".

Parameters:

salt - A two-character string representing the salt used to iterate the encryption engine in lots of different ways. If you are generating a new encryption then this value should be randomised.
original - The password to be encrypted.

Returns:

A string consisting of the 2-character salt followed by the encrypted password.

```
public static final String crypt(String original)
```

Encrypt a password given the cleartext password. This method generates a random salt using the 'java.util.Random' class.

Parameters:

original - The password to be encrypted.

Returns:

A string consisting of the 2-character salt followed by the encrypted password.

```
public static final boolean matches(String encryptedPassword,
                                    String enteredPassword)
```

Check that enteredPassword encrypts to encryptedPassword.

Parameters:

encryptedPassword - The encryptedPassword. The first two characters are assumed to be the salt. This string would be the same as one found in a Unix /etc/passwd file.
enteredPassword - The password as entered by the user (or otherwise acquired).

Returns:

true if the password should be considered correct.

Using the driver in a multi Threaded or Servlet environment

A problem with many JDBC drivers, is that only one thread can use a Connection at any one time - otherwise a thread could send a query

while another one is receiving results, and this would be a bad thing for the database engine.

PostgreSQL 6.4, brings thread safety to the entire driver. Standard JDBC was thread safe in 6.3.x, but the Fastpath API wasn't.

So, if your application uses multiple threads (which most decent ones would), then you don't have to worry about complex schemes to ensure only one uses the database at any time.

If a thread attempts to use the connection while another is using it, it will wait until the other thread has finished it's current operation.

If it's a standard SQL statement, then the operation is sending the statement, and retrieving any ResultSet (in full).

If it's a Fastpath call (ie: reading a block from a LargeObject), then it's the time to send, and retrieve that block.

This is fine for applications & applets, but can cause a performance problem with servlets.

With servlets, you can have a heavy load on the connection. If you have several threads performing queries, then each one will pause, which may not be what you are after.

To solve this, you would be advised to create a pool of Connections.

When ever a thread needs to use the database, it asks a manager class for a Connection. It hands a free connection to the thread, and marks it as busy. If a free connection is not available, it opens one.

Once the thread has finished with it, it returns it to the manager, who can then either close it, or add it to the pool. The manager would also check that the connection is still alive, and remove it from the pool if it's dead.

So, with servlets, it's up to you to use either a single connection, or a pool. The plus side for a pool is that threads will not be hit by the bottle neck caused by a single network connection. The down side, is that it increases the load on the server, as a backend is created for each Connection.

It's up to you, and your applications requirements.

Further Reading

If you have not yet read it, I'd advise you read the JDBC API Documentation (supplied with Sun's JDK), and the JDBC Specification. Both are available on JavaSoft's web site².

My own web site³ contains updated information not included in this document, and also includes precompiled drivers for v6.4, and earlier.

Notas

1. peter@retep.org.uk
2. <http://www.javasoft.com>
3. <http://www.retep.org.uk>

Capítulo 51. Interfaz de Programación Lisp

`pg.el` es una interfaz para emacs que permite establecer una conexión por socket con Postgres.

Autor: Escrito por Eric Marsden¹ 21 Jul 1999.

`pg.el` es una interfaz para emacs (extraordinario editor de textos). El módulo es capaz de asignar tipos de SQL al tipo equivalente de Lisp de Emacs. Actualmente no soporta encriptación, autenticación Kerberos, ni objetos grandes (large objects).

El código (version 0.2) está disponible bajo la licencia GNU GPL en <http://www.chez.com/emarsden/dow>

Cambios desde la última versión:

- Incluye funcionalidad con XEmacs (probado con Emacs 19.34 y 20.2, y XEmacs 20.4)
- Añade funciones que proveen de metainformación (lista de bases de datos, de tablas, de columnas).
- los argumentos de 'pg:result' son ahora :keywords
- Resistente a MULE
- Introduce más código de autocomprobación

Por favor, nótese que esta es una API de programadores, y no proporciona ninguna forma de interfaz con el usuario. Ejemplo:

```
(defun demo ()
  (interactive)
  (let* ((conn (pg:connect "template1" "postgres" "postgres"))
        (res (pg:exec conn "SELECT * from scshdemo WHERE a = 42")))
    (message "status is %s" (pg:result res :status))
    (message "metadata is %s" (pg:result res :attributes))
    (message "data is %s" (pg:result res :tuples))
    (pg:disconnect conn)))
```

Notas

1. <mailto:emarsden@mail.dotcom.fr>
2. <http://www.chez.com/emarsden/downloads/pg.el>

Capítulo 52. Código Fuente Postgres

Formateo

El formateo del código fuente utiliza un espacio a 4 columnas tabuladas, actualmente con tabulaciones protegidas (i.e. las tabulaciones no son expandidas a espacios).

Para emacs, añade lo siguiente (o algo similar) a tu archivo de inicialización ~/.emacs:

```
;; comprueba los archivos con un path que contenga "postgres" o "psql"
(setq auto-mode-alist (cons '("\\(postgres\\|pgsql\\).*\\.\\(ch\\|\\)" . pgsql-
c-mode) auto-mode-alist))
(setq auto-mode-alist (cons '("\\(postgres\\|pgsql\\).*\\.cc\\|" . pgsql-
c-mode) auto-mode-alist))

(defun pgsql-c-mode ()
  ;; configura el formateo para el código C de Postgres
  (interactive)
  (c-mode)
  (setq-default tab-width 4)
  (c-set-style "bsd") ; configura c-basic-offset a 4, mas otras cosas
  (c-set-offset 'case-label '+) ; vuelve la indexación de las cajas pa-
ra que se empareje con el cliente PG
  (setq indent-tabs-mode t)) ; nos aseguramos de que mantiene las ta-
bulaciones cuando indexa
```

Para vi, tu ~/.vimrc o archivo equivalente debe contener lo siguiente:

```
set tabstop=4
```

o equivalentemente dentro de vi, intenta

```
:set ts=4
```

Las herramientas para ver textos `more` y `less` pueden ser invocadas como

```
more -x4
less -x4
```


Capítulo 53. Revisión de las características internas de PostgreSQL

Autor: Este capítulo apareció originalmente como parte de la tesis doctoral de Stefan Simkovic preparada en la Universidad de Tecnología de Viena bajo la dirección de O.Univ.Prof.Dr. Georg Gottlob y Univ.Ass. Mag. Katrin Seyr.

Este capítulo da una visión general de la estructura interna del motor de Postgres. Tras la lectura de las siguientes secciones, usted tendrá una idea de como se procesa una consulta. No espere aquí una descripción detallada (¡creo que esa descripción detallada incluyendo todas las estructuras de datos y funciones utilizadas en Postgres excedería de 1000 páginas!). Este capítulo intenta ayudar en la comprensión del control general y del flujo de datos dentro del motor desde que se recibe una consulta hasta que se emiten los resultados.

El camino de una consulta

Damos aquí una corta revisión a los pasos que debe seguir una consulta hasta obtener un resultado.

1. Se ha establecido una conexión desde un programa de aplicación al servidor Postgres. El programa de aplicación transmite una consulta y recibe el resultado enviado por el servidor.
2. La *etapa del parser (traductor)* chequea la consulta transmitida por el programa de aplicación (cliente) para comprobar que la sintaxis es correcta y crear un *árbol de la consulta*.
3. El *sistema de reescritura* toma el árbol de la consulta creado en el paso del traductor y busca *reglas* (almacenadas en los *catálogos del sistema*) que pueda aplicarle al *árbol de la consulta* y realiza las transformaciones que se dan en el/los *cuerpo/s de la/s regla/s*. Encontramos una aplicación del sistema de reescritura en la realización de las *vistas*.

Siempre que se realiza una consulta contra una vista (es decir, una *tabla virtual*), el sistema de reescritura reescribe la consulta del usuario en una consulta que accede a las *tablas base* dadas en la *definición de la vista* inicial.

4. El *planeador/optimizador* toma el árbol de la consulta (reescrita) y crea un *plan de la consulta* que será el input para el *ejecutor*.

Hace esto creando previamente todas las posibles *rutas* que le conducen a un mismo resultado. Por ejemplo, si hay un índice en una relación que debe ser comprobada, hay dos rutas para comprobarla. Una posibilidad es un simple barrido secuencial y la otra posibilidad es utilizar el índice. Luego se estima el coste de ejecución de cada plan, y se elige y ejecuta el plan más rápido.

5. El ejecutor realiza de modo recursivo el *árbol del plan* y recupera tuplas en la forma representada en el plan. El ejecutor hace uso del *sistema de almacenamiento* mientras está revisando las relaciones, realiza *ordenaciones (sorts)* y *joins*, evalúa *cualificaciones* y finalmente devuelve las tuplas derivadas.

En las siguientes secciones, cubriremos todos los pasos listados antes en más detalle, para dar un mejor conocimiento de las estructuras de datos y de control interno de Postgres.

Cómo se establecen las conexiones

Postgres está implementado como un simple modelo cliente/servidor a "proceso por usuario". En este modelo hay un *proceso cliente* conectado a exactamente un *proceso servidor*. Como nosotros no conocemos *per se* cuantas conexiones se harán, utilizaremos un *proceso master* que lanza un nuevo proceso servidor para cada conexión que se solicita. Este proceso master se llama `postmaster` y escucha en un puerto TCP/IP específico a las conexiones entrantes. Cada vez que se detecta un requerimiento de conexión, el proceso `postmaster` lanza un nuevo proceso servidor llamado `postgres`. Las tareas de servidor (los procesos `postgres`) se comunican unos con otros utilizando *semáforos* y *memoria compartida* (shared memory) para asegurar la integridad de los datos a través de los accesos concurrentes a los datos. La figura \ref{connection} ilustra la interacción del proceso master `postmaster`, el proceso servidor `postgres` y una aplicación cliente.

El proceso cliente puede ser el interface de usuario (frontend) `psql` (para realizar consultas SQL interactivas) o cualquier aplicación de usuario implementada utilizando la biblioteca `libpq`. Nótese que las aplicaciones implementadas utilizando `ecpg` (el preprocesador de SQL embebido de Postgres para C) también utiliza esta biblioteca.

Una vez que se ha establecido una conexión, el proceso cliente puede enviar una consulta al servidor (*backend*). Esta consulta se transmite utilizando un texto plano, es decir, no se ha hecho una traducción en el cliente (*frontend*). El servidor traduce la consulta, crea un *plan de ejecución*, ejecuta el plan y remite las tuplas recuperadas al cliente a través de la conexión establecida.

La etapa de traducción

La *etapa de traducción* consiste en dos partes:

- El *traductor* definido en `gram.y` y `scan.l` se construye utilizando las herramientas de Unix `yacc` y `lex`.
- El *proceso de transformación* realiza modificaciones y aumentos a las estructuras de datos devueltas por el traductor.

Traductor

El traductor debe comprobar la cadena de caracteres de la consulta (que le llega como texto ASCII plano) para comprobar la validez de la sintaxis. Si la sintaxis es correcta, se construye un *árbol de traducción* y se devuelve un mensaje de error en otro caso. Para la implementación se han utilizado las bien conocidas herramientas de Unix `lex` y `yacc`.

El *lector* (lexer) se define en el fichero `scan.l` y es el responsable de reconocer los *identificadores*, las *palabras clave de SQL*, etc. Para cada palabra clave o identificador que encuentra, se genera y traslada al traductor traductor una *señal*.

El traductor está definido en el fichero `gram.y` y consiste en un conjunto de *reglas de gramática* y *acciones* que serán ejecutadas cada vez que se dispara una regla. El código de las acciones (que actualmente es código C) se utiliza para construir el árbol de traducción.

El fichero `scan.l` se transforma en el fichero fuente C `scan.c` utilizando el programa `lex` u `gram.y` se transforma en `gram.c` utilizando `yacc`. Una vez se han realizado

estas transformaciones, cualquier compilador C puede utilizarse para crear el traductor. No se deben nunca realizar cambios en los ficheros C generados, pues serán sobrescritos la próxima vez que sean llamados `lex` o `yacc`.

Nota: Las transformaciones y compilaciones mencionadas normalmente se hacen automáticamente utilizando los *makefile* vendidos con la distribución de los fuentes de Postgres.

Más adelante en este mismo documento se dará una descripción detallada de `yacc` o de las reglas de gramática dadas en `gram.y`. Hay muchos libros y documentos relacionados con `lex` y `yacc`. Debería usted familiarizarse con `yacc` antes de empezar a estudiar la gramática mostrada en `gram.y`, pues de otro modo no entenderá usted lo que está haciendo.

Para un mejor conocimiento de las estructuras de datos utilizadas en Postgres para procesar una consulta utilizaremos un ejemplo para ilustrar los cambios hechos a estas estructuras de datos en cada etapa.

Ejemplo 53-1. Una SELECT sencilla

Este ejemplo contiene la siguiente consulta sencilla que será usada en varias descripciones y figuras a lo largo de las siguientes secciones. La consulta asume que las tablas dadas en *The Supplier Database* ya han sido definidas.

```
select s.sname, se.pno
  from supplier s, sells se
 where s.sno > 2 and s.sno = se.sno;
```

La figura \ref{parsetree} muestra el *árbol de traducción* construido por las reglas y acciones de gramática dadas en `gram.y` para la consulta dada en *Una SELECT sencilla*. Este ejemplo contiene la siguiente consulta sencilla que será usada en varias descripciones y figuras a lo largo de las siguientes secciones. La consulta asume que las tablas dadas en *The Supplier Database* ya han sido definidas. `select s.sname, se.pno from supplier s, sells se where s.sno > 2 and s.sno = se.sno;` (sin el árbol de operador para la cláusula `WHERE` que se muestra en la figura \ref{where_clause} porque no había espacio suficiente para mostrar ambas estructuras de datos en una sola figura).

El nodo superior del árbol es un nodo `SelectStmt`. Para cada entrada que aparece en la *cláusula FROM* de la consulta de SQL se crea un nodo `RangeVar` que mantiene el nombre del *alias* y un puntero a un nodo `RelExpr` que mantiene el nombre de la *relación*. Todos los nodos `RangeVar` están recogidos en una lista unida al campo `fromClause` del nodo `SelectStmt`.

Para cada entrada que aparece en la *lista de la SELECT* de la consulta de SQL se crea un nodo `ResTarget` que contiene un puntero a un nodo `Attr`. El nodo `Attr` contiene el *nombre de la relación* de la entrada y un puntero a un nodo `Value` que contiene el nombre del *attribute*. Todos los nodos `ResTarget` están reunidos en una lista que está conectada al campo `targetList` del nodo `SelectStmt`.

La figura \ref{where_clause} muestra el árbol de operador construido para la cláusula `WHERE` de la consulta de SQL dada en el ejemplo *Una SELECT sencilla*. Este ejemplo contiene la siguiente consulta sencilla que será usada en varias descripciones y figuras a lo largo de las siguientes secciones. La consulta asume que las tablas dadas en *The Supplier Database* ya han sido definidas. `select s.sname, se.pno from supplier s, sells se where s.sno`

$s.sno > 2$ and $s.sno = se.sno$; que está unido al campo `qual` del nodo `SelectStmt`. El nodo superior del árbol de operador es un nodo `A_Expr` representando una operación `AND`. Este nodo tiene dos sucesores llamados `lexpr` y `rexpr` apuntando a dos *subárboles*. El subárbol unido a `lexpr` representa la cualificación $s.sno > 2$ y el unido a `rexpr` representa $s.sno = se.sno$. Para cada atributo, se ha creado un nodo `Attr` que contiene el nombre de la relación y un puntero a un nodo `Value` que contiene el nombre del atributo. Para el termino constante que aparece en la consulta, se ha creado un nodo `Const` que contiene el valor.

Proceso de transformación

El *proceso de transformación* toma el árbol producido por el traductor como entrada y procede recursivamente a través suyo. Si se encuentra un nodo `SelectStmt`, se transforma en un nodo `Query` que será el nodo superior de la nueva estructura de datos. La figura \ref{transformed} muestra la estructura de datos transformada (la parte de la *cláusula WHERE* transformada se da en la figura \ref{transformed_where} porque no hay espacio suficiente para mostrarlo entero en una sola figura).

Ahora se realiza una comprobación sobre si los *nombres de relaciones* de la *cláusula FROM* son conocidas por el sistema. Para cada nombre de relación que está presente en los *catálogos del sistema*, se crea un nodo `RTE` que contiene el nombre de la relación, el *nombre del alias* y el *identificador (id) de la relación*. A partir de ahora, se utilizan los identificadores de relación para referirse a las *relaciones* dadas en la consulta. Todos los nodos `RTE` son recogidos en la *lista de entradas de la tabla de rango* que está conectada al campo `rtable` del nodo `Query`. Si se detecta en la consulta un nombre de relación desconocido para el sistema, se devuelve un error y se aborta el procesado de la consulta.

El siguiente paso es comprobar si los *nombres de atributos* utilizados están contenidos en las relaciones dadas en la consulta. Para cada atributo que se encuentra se crea un nodo `TLE` que contiene un puntero a un nodo `Resdom` (que contiene el nombre de la columna) y un puntero a un nodo `Var`. Hay dos números importantes en el nodo `Var`. El campo `varno` da la posición de la relación que contiene el atributo actual en la lista de entradas de la tabla de rango creada antes. El campo `varattno` da la posición del atributo dentro de la relación. Si el nombre de un atributo no se consigue encontrar, se devuelve un error y se aborta el procesado de la consulta.

El sistema de reglas de Postgres

Postgres utiliza un poderoso *sistema de reglas* para la especificación de *vistas* y *actualizaciones de vistas* ambiguas. Originalmente el sistema de reglas de Postgres consistía en dos implementaciones:

- El primero trabajaba utilizando el procesado a *nivel de tupla* y se implementaba en el *ejecutor*. El sistema de reglas se disparaba cada vez que se accedía una tupla individual. Esta implementación se retiró en 1.995 cuando la última versión oficial del proyecto Postgres se transformó en Postgres95.
- La segunda implementación del sistema de reglas es una técnica llamada *reescritura de la consulta*. El *sistema de reescritura* es un módulo que existe entre la *etapa del traductor* y el *planificador/optimizador*. Esta técnica continúa implementada.

Para información sobre la sintaxis y la creación de reglas en sistema Postgres Diríjase a la *Guía del Usuario de PostgreSQL*.

El sistema de reescritura

El *sistema de reescritura de la consulta* es un módulo entre la etapa de traducción y el planificador/optimizador. Procesa el árbol devuelto por la etapa de traducción (que representa una consulta de usuario) y si existe una regla que deba ser aplicada a la consulta reescribe el árbol de una forma alternativa.

Técnicas para implementar vistas

Ahora esbozaremos el algoritmo del sistema de reescritura de consultas. Para una mejor ilustración, mostraremos como implementar vistas utilizando reglas como ejemplo.

Tengamos la siguiente regla:

```
create rule view_rule
as on select
to test_view
do instead
select s.sname, p.pname
from supplier s, sells se, part p
where s.sno = se.sno and
      p.pno = se.pno;
```

Esta regla se *disparará* cada vez que se detecte una SELECT contra la relación `test_view`. En lugar de seleccionar las tuplas de `test_view`, se ejecutará la instrucción SELECT dada en la *parte de la acción* de la regla.

Tengamos la siguiente consulta de usuario contra `test_view`:

```
select sname
from test_view
where sname <> 'Smith';
```

Tenemos aquí una lista de los pasos realizados por el sistema de reescritura de la consulta cada vez que aparece una consulta de usuario contra `test_view`. (El siguiente listado es una descripción muy informal del algoritmo únicamente para una comprensión básica. Para una descripción detallada diríjase a *Stonebraker et al, 1989*).

Reescritura de `test_view`

1. Toma la consulta dada por la parte de acción de la regla.
2. Adapta la lista-objetivo para recoger el número y orden de los atributos dados en la consulta del usuario.
3. Añade la cualificación dada en la cláusula WHERE de la consulta del usuario a la cualificación de la consulta dada en la parte de la acción de la regla.

Dada la definición de la regla anterior, la consulta del usuario será reescrita a la siguiente forma (Nótese que la reescritura se hace en la representación interna de la

consulta del usuario devuelta por la etapa de traducción, pero la nueva estructura de datos representará la siguiente consulta):

```
select s.sname
from supplier s, sells se, part p
where s.sno = se.sno and
      p.pno = se.pno and
      s.sname <> 'Smith';
```

Planificador/optimizador

La tarea del *planificador/optimizador* es crear un plan de ejecución óptimo. Primero combina todas las posibles vías de *barrer* (scanear) y *cruzar* (join) las relaciones que aparecen en una consulta. Todas las rutas creadas conducen al mismo resultado y es el trabajo del optimizador estimar el coste de ejecutar cada una de ellas para encontrar cual es la más económica.

Generando planes posibles

El planificador/optimizador decide qué planes deberían generarse basándose en los tipos de índices definidos sobre las relaciones que aparecen en una consulta. Siempre existe la posibilidad de realizar un barrido secuencial de una relación, de modo que siempre se crea un plan que sólo utiliza barridos secuenciales. Se asume que hay definido un índice en una relación (por ejemplo un índice B-tree) y una consulta contiene la restricción `relation.attribute OPR constant`. Si `relation.attribute` acierta a coincidir con la clave del índice B-tree y `OPR` es distinto de '`<>`' se crea un plan utilizando el índice B-tree para barrer la relación. Si hay otros índices presentes y las restricciones de la consulta aciertan con una clave de un índice, se considerarán otros planes.

Tras encontrar todos los planes utilizables para revisar relaciones únicas, se crean los planes para cruzar (join) relaciones. El planificador/optimizador considera sólo cruces entre cada dos relaciones para los cuales existe una cláusula de cruce correspondiente (es decir, para las cuales existe una restricción como `WHERE rel1.attr1=rel2.attr2`) en la cualificación de la WHERE. Se generan todos los posibles planes para cada cruce considerado por el planificador/optimizador. Las posibles estrategias son:

- *Cruce de iteración anidada* (nested iteration join): La relación derecha se recorre para cada tupla encontrada en la relación izquierda. Esta estrategia es fácil de implementar pero puede consumir mucho tiempo.
- *Cruce de ordenación mezclada* (merge sort join): Cada relación es ordenada por los atributos del cruce antes de iniciar el cruce mismo. Después se mezclan las dos relaciones teniendo en cuenta que ambas relaciones están ordenadas por los atributos del cruce. Este modelo de cruce es más atractivo porque cada relación debe ser barrida sólo una vez.
- *Cruce indexado* (hash join): La relación de la derecha se indexa primero sobre sus atributos para el cruce. A continuación, se barre la relación izquierda, y los valores apropiados de cada tupla encontrada se utilizan como clave indexada para localizar las tuplas de la relación derecha.

Estructura de datos del plan

Daremos ahora una pequeña descripción de los nodos que aparecen en el plan. La figura \ref{plan} muestra el plan producido para la consulta del ejemplo \ref{simple_select}.

El nodo superior del plan es un nodo *Cruce Mezclado* (*MergeJoin*) que tiene dos sucesores, uno unido al campo *árbol izquierdo* (*lefttree*) y el segundo unido al campo *árbol derecho* (*righttree*). Cada uno de los subnodos representa una relación del cruce. Como se mencionó antes, un cruce de mezcla ordenada requiere que cada relación sea ordenada. Por ello encontramos un nodo *Sort* en cada subplan. La cualificación adicional dada en la consulta (*s.sno > 2*) se envía tan lejos como es posible y se une al campo *ppqual* de la rama *SeqScan* del nodo del correspondiente subplan.

La lista unida al campo *mergeclauses* del nodo *Cruce Mezclado* (*MergeJoin*) contiene información sobre los atributos de cruce. Los valores 65000 y 65001 de los campos *varno* y los nodos *VAR* que aparecen en la lista *mergeclauses* (y también en la lista *objetivo*) muestran que las tuplas del nodo actual no deben ser consideradas, sino que se deben utilizar en su lugar las tuplas de los siguientes nodos "más profundos" (es decir, los nodos superiores de los subplanes).

Nótese que todos los nodos *Sort* y *SeqScan* que aparecen en la figura \ref{plan} han tomado una lista *objetivo*, pero debido a la falta de espacio sólo se ha dibujado el correspondiente al *Cruce Mezclado*.

Otra tarea realizada por el planificador/optimizador es fijar los *identificadores de operador* en los nodos *Expr* y *Oper*. Como se mencionó anteriormente, Postgres soporta una variedad de tipos diferentes de datos, e incluso se pueden utilizar tipos definidos por el usuario. Para ser capaz de mantener la gran cantidad de funciones y operadores, es necesario almacenarlos en una tabla del sistema. Cada función y operador toma un identificador de operador único. De acuerdo con los tipos de los atributos usados en las cualificaciones, etc, se utilizan los identificadores de operador apropiados.

Ejecutor

El *ejecutor* toma el plan devuelto por el planificador/optimizador y arranca procesando el nodo superior. En el caso de nuestro ejemplo (la consulta dada en el ejemplo \ref{simple_select}), el nodo superior es un nodo *Cruce Mezclado* (*MergeJoin*).

Antes de poder hacer ninguna mezcla, se deben leer dos tuplas, una de cada subplan. De este modo, el ejecutor mismo llama recursivamente a procesar los subplanes (arranca con el subplan unido al *árbol izquierdo*). El nuevo nodo superior (el nodo superior del subplan izquierdo) es un nodo *SeqScan*, y de nuevo se debe tomar una tupla antes de que el nodo mismo pueda procesarse. El ejecutor mismo llama recursivamente otra vez al subplan unido al *árbol izquierdo* del nodo *SeqScan*.

El nuevo nodo superior es un nodo *Sort*. Como un *sort* se debe realizar sobre la relación completa, el ejecutor arranca leyendo tuplas desde el subplan del nodo *Sort* y las ordena en una relación temporal (en memoria o en un fichero) cuando se visita por primera vez el nodo *Sort*. (Posteriormente exámenes del nodo *Sort* devolverán siempre únicamente una tupla de la relación temporalmente ordenada).

Cada vez que el procesado del nodo *Sort* necesita de una nueva tupla, se llama de forma recursiva al ejecutor para que trate el nodo *SeqScan* unido como subplan. La

relación (a la que se refiere internamente por el valor dado en el campo `scanrelid`) se recorre para encontrar la siguiente tupla. Si la tupla satisface la cualificación dada por el árbol unido a `qual` se da por buena para su tratamiento, y en otro caso se lee la siguiente tupla hasta la primera que satisfaga la cualificación. Si se ha procesado la última tupla de la relación, se devuelve un puntero `NULL`.

Una vez que se ha recuperado una tupla en el árbol izquierdo del Cruce Mezclado (`MergeJoin`), se procesa del mismo modo el árbol derecho. Si se tienen presentes ambas tuplas, el ejecutor procesa el Cruce Mezclado. Siempre que se necesita una nueva tupla de uno de los subplanes, se realiza una llamada recursiva al ejecutor para obtenerla. Si se pudo crear una tupla para cruzarla, se devuelve y se da por terminado el procesado completo de árbol del plan.

Se realizan ahora los pasos descritos para cada una de las tuplas, hasta que se devuelve un puntero `NULL` para el procesado del nodo Cruce Mezclado, indicando que hemos terminado.

Capítulo 54. pg_options

Nota: Aportado por Massimo Dal Zotto¹

El fichero opcional `data/pg_options` contiene opciones de tiempo de ejecución utilizadas por el servidor para controlar los mensajes de seguimiento y otros parámetros ajustables de servidor. Lo que hace a este fichero interesante es el hecho de que es re-leído por un servidor que recibe una señal `SIGHUP`, haciendo así posible cambiar opciones de tiempo de ejecución sobre la marcha sin necesidad de reorganizar Postgres. Las opciones especificadas en este fichero pueden ser banderas de debugging utilizadas por el paquete de seguimiento (`backend/utils/misc/trace.c`) o parámetros numéricos que pueden ser usados por el servidor para controlar su comportamiento. Las nuevas opciones y parámetros deben ser definidos en `backend/utils/misc/trace.c` y `backend/include/utils/trace.h`.

Por ejemplo, supongamos que queremos añadir mensajes de seguimiento condicional y un parámetro numérico ajustable al código en el fichero `foo.c`. Todo lo que necesitamos hacer es añadir la constante `TRACE_FOO` y `OPT_FOO_PARAM` en `backend/include/utils/trace.h`:

```
/* file trace.h */
enum pg_option_enum {
    ...
    TRACE_FOO, /* trace foo functions */
    OPT_FOO_PARAM, /* foo tunable parameter */

    NUM_PG_OPTIONS /* must be the last item of enum */
};
```

y una línea correspondiente en `backend/utils/misc/trace.c`:

```
/* file trace.c */
static char *opt_names[] = {
    ...
    "foo", /* trace foo functions */
    "fooparam" /* foo tunable parameter */
};
```

Las opciones se deben especificar en los dos ficheros exactamente en el mismo orden. En los ficheros fuente `foo` podemos ahora hacer referencia a las nuevas banderas con:

```
/* file foo.c */
#include "trace.h"
#define foo_param pg_options[OPT_FOO_PARAM]

int
foo_function(int x, int y)
{
    TPRINTF(TRACE_FOO, "entering foo_function, foo_param=%d", foo_param);
    if (foo_param > 10) {
        do_more_foo(x, y);
    }
}
```

Los ficheros existentes que utilizan banderas de seguimiento privadas pueden cambiarse simplemente añadiendo el siguiente código:

```
#include "trace.h"
/* int my_own_flag = 0; - removed */
#define my_own_flag pg_options[OPT_MY_OWN_FLAG]
```

Todas las *pg_options* son inicializadas a cero en el arranque del servidor. Si necesitamos un valor de defecto diferente necesitaremos añadir algún código de inicialización en el principio de *PostgresMain*. Ahora podemos fijar el parámetro *foo_param* y activar el seguimiento *foo* escribiendo valores en el fichero *data/pg_options*:

```
# file pg_options
....
foo=1
fooparam=17
```

Las nuevas opciones serán leídas por todos los nuevos servidores conforme van arrancando. Para hacer efectivos los cambios para todos los servidores que están en funcionamiento, necesitaremos enviar un *SIGHUP* al *postmaster*. La señal será enviada automáticamente a todos los servidores. Podemos activar los cambios también para un servidor específico individual enviándole la señal *SIGHUP* directamente a él.

Las *pg_options* pueden también especificarse con el interruptor (switch) *-T* de *Postgres*:

```
postgres options -T "verbose=2,query,hostlookup-"
```

Las funciones utilizadas para imprimir los errores y los mensajes de debug pueden hacer uso ahora de la facilidad *sislog(2)*. Los mensajes impresos en *stdout* y *stderr* son preformatados con una marca horaria que contiene también la identificación del proceso del servidor:

```
#timestamp      #pid      #message
980127.17:52:14.173 [29271] StartTransactionCommand
980127.17:52:14.174 [29271] ProcessUtility: drop table t;
980127.17:52:14.186 [29271] SIIncNumEntries: table is 70% full
980127.17:52:14.186 [29286] Async_NotifyHandler
980127.17:52:14.186 [29286] Waking up sleeping backend process
980127.19:52:14.292 [29286] Async_NotifyFrontEnd
980127.19:52:14.413 [29286] Async_NotifyFrontEnd done
980127.19:52:14.466 [29286] Async_NotifyHandler done
```

Este formato incrementa también la capacidad de leer los ficheros de mensajes y permite a las personas el conocimiento exacto de lo que cada servidor está haciendo y en qué momento. También hace más fácil escribir programas con *awk* o *perl* que revisen

el rastro para detectar errores o problemas en la base de datos, o calcular estadísticas de tiempo de las transacciones.

Los mensajes impresos en el syslog utilizan la facilidad de rastro LOG_LOCAL0. El uso de syslog puede ser controlada con la `pg_option syslog`. Desgraciadamente, muchas funciones llaman directamente a `printf()` para imprimir sus mensajes en stdout o stderr y su salida no puede ser redirigida a syslog o tener indicaciones cronológicas en ella. Sería deseable que todas las llamadas a `printf` fueran reemplazadas con la macro `PRINTF` y la salida a stderr fuese cambiada para utilizar `EPRINTF` en su lugar, de modo que podamos controlar todas las salidas de un modo uniforme.

El nuevo mecanismo de las `pg_options` es más conveniente que definir nuevas opciones de switch en los servidores porque:

- No tenemos que definir un switch diferente para cada idea que queramos controlar. Todas las opciones están definidas como palabras claves en un fichero externo almacenado en el directorio de datos.
- No tenemos que rearrancar Postgres para cambiar el valor de alguna opción. Normalmente las opciones del servidor se especifican al postmaster y pasados a cada servidor cuando sea arrancado. Ahora son leídos de un fichero.
- Podemos cambiar las opciones sobre la marcha mientras el servidor está corriendo. Podemos de este modo investigar algunos problemas activando los mensajes de seguimiento sólo cuando aparece el problema. Podemos también intentar diferentes valores de parámetros ajustables.

El formato de las `pg_options` es como sigue:

```
# comment
option=integer_value  # set value for option
option                # set option = 1
option+               # set option = 1
option-               # set option = 0
```

Notese que *keyword* puede también ser una abreviatura del nombre de opción definida en `backend/utils/misc/trace.c`.

Refierase al capítulo de la *Guía del Administrador* sobre las opciones de tiempo de ejecución para una lista completa de opciones soportadas actualmente.

Algo del código existente que utiliza variables privadas e interruptores de opciones se han cambiado para hacer uso de las posibilidades de las `pg_options`, fundamentalmente en `postgres.c`. Sería deseable modificar todo el código existente en este sentido, de modo que podamos hacer uso de muchos de los switches en la línea de comando de Postgres y poder tener más opciones ajustables con un lugar único para situar los valores de las opciones.

Notas

1. <mailto:dz@cs.unitn.it>

Capítulo 55. Optimización Genética de Consulta en Sistemas de Base de Datos

Author: Escrito por Martin Utesch¹ del Instituto de Control Automático de la Universidad de Minería y Tecnología en Freiberg, Alemania.

Planificador de consulta para un Problema Complejo de Optimización

Entre todos los operadores relacionales, uno de los más difícil de procesar y optimizar es la *unión*. El número de vías alternativas para responder a una consulta crece exponencialmente con el número de **uniones** incluidas en ella. EL esfuerzo adicional de optimización esta causado por la existencia de una variedad de *metodos de unión* para procesar **uniones** individuales (p.e., bucle anidado, exploración de índice, fusión de unión en Postgres) y de una gran variedad de *indices* (e.p., árbol-r, árbol-b, hash en Postgres) como camino de acceso para las relaciones.

La actual implementación del optimizador de Postgres realiza una *busqueda cercana y exhaustiva* sobre el espacio de las estrategias alternativas. Esta técnica de optimización de consulta no es adecuada para soportar los dominios de la aplicación de base de datos que implica la necesidad de consultas extensivas, tales como la inteligencia artificial.

El Instituto de Control Automático de la Universidad de Minería y Tecnología, en Freiberg, Alemania, se encontró los problemas descritos cuando su personal quiso utilizar la DBMS Postgres como software base para sistema de soporte de decisión basado en el conocimiento para mantener un red de energía eléctrica. La DBMS necesitó manejar consultas con **unión** para el motor de inferencia del sistema basado en el conocimiento.

Las dificultades del rendimiento al explorar el espacio de los posibles planes de la consulta hizo surgir la demanda de un nueva técnica de optimización que se ha desarrollado.

A continuación, proponemos la implementación de un *Algoritmo Genético* como una opción para el problema de la optimización de consultas de la base de datos.

Algoritmo Genéticos (AG)

El AG es un método de búsqueda heurística que opera mediante búsqueda determinada y aleatoria. El conjunto de soluciones posibles para el problema de la optimización se considera como una *población de individuos*. El grado de adaptación de un individuo en su entorno esta determinado por su *adaptabilidad*.

Las coordenadas de un individuo en el espacio de la búsquedas están representadas por los *cromosomas*, en esencia un conjunto de cadenas de caracteres. Un *gen* es una subsección de un cromosoma que codifica el valor de un único parámetro que ha de ser optimizado. Las Codificaciones típicas para un gen pueden ser *binarias* o *enteras*.

Mediante la simulación de operaciones evolutivas *recombinación*, *mutación*, y *selección* se encuentran nuevas generaciones de puntos de búsqueda, los cuales muestran un mayor nivel de adaptabilidad que sus antecesores.

Según la FAQ de "comp.ai.genetic" no se puede enfatizar más claramente que un AG no es un búsqueda puramente aleatoria para una solución del problema. El AG usa procesos estocástico, pero el resultado es claramente no aleatorio (mejor que el aleatorio).

Diagrama Estructurado de un AG:

```

-----

P(t)    generación de antecesores en un tiempo t
P''(t)   generación de descendientes en un tiempo t

+=====+
| »»»»»> Algoritmo AG «««««««|
+=====+
| INICIALIZACIÓN t := 0          |
+=====+
| INICIALIZACIÓN P(t)          |
+=====+
| evaluación ADAPTABILIDAD de P(t) |
+=====+
| mientras no CRITERIO DE PARADA hacer |
|   +-----+                  |
|   | P'(t) := RECOMBINACIÓN{P(t)} | |
|   +-----+                  |
|   | P''(t) := MUTACIÓN{P'(t)}   | |
|   +-----+                  |
|   | P(t+1) := SELECCIÓN{P''(t) + P(t)} | |
|   +-----+                  |
|   | evaluación ADAPTABILIDAD de P''(t) | |
|   +-----+                  |
|   | t := t + 1                    | |
| +-----+                  |
+=====+

```

Optimización Genética de Consultas (GEQO) en Postgres

El módulo OGEC esta previsto para solucionar el problema de optimización de consultas similares al problema del viajante (PV). Los planes posibles de consulta son codificados por cadenas de enteros. Cada cadena representa el orden de la una relación de **unión** de la consulta a la siguiente. P. e., el árbol de la consulta

```

      /\
     /\ 2
    /\ 3
   4  1

```

esta codificado por la cadena de enteros '4-1-3-2', que significa, la primera relación de unión '4' y '1', después '3', y después '2', donde 1, 2, 3, 4 son relids en Postgres.

Partes del módulo OGEC han sido adaptadas del algoritmo Genitor de D. Whitley.

Las características específicas de la implementación de OGEC en Postgres son:

- El uso de un AG en *estado constante* (reemplazo de los individuos con menor adaptación de la población, no el reemplazo total de un generación) permite converger rápidamente hacia planes mejorados de consulta. Esto es esencial para el manejo de la consulta en un tiempo razonable;

- El uso de *cruce de recombinación limitada* que esta especialmente adaptado para quedarse con el límite menor de pérdidas para la solución del PV por medio de un AG;
- La mutación como operación genética se recomienda a fin de que no sean necesarios mecanismos de reparación para generar viajes legales del PV.

El módulo OGEC proporciona los siguientes beneficios para la DBMS Postgres comparado con la implementación del optimizador de consultas de Postgres:

- El manejo de grandes consultas de tipo **unión** a través de una búsqueda no-exhaustiva;
- Es necesario una mejora en la aproximación del tamaño del coste de los planes de consulta desde la fusión del plan más corto (el módulo OGEC evalúa el coste de un plan de consulta como un individuo).

Futuras Tareas de Implementación para el OGEC de Postgres

Mejoras Básicas

Mejora en la liberación de memoria cuando la consulta ya se ha procesado

Para largas consultas de tipo **unión** el gasto de tiempo de computación para un optimizar genética de la consulta parece ser una simple *fracción* del tiempo que necesita Postgres para la liberación de memoria mediante la rutina `MemoryContextFree`, del archivo `backend/utils/mmgr/mcxt.c`. Depurando se mostró que se atascaba en un bucle de la rutina `OrderedElemPop`, archivo `backend/utils/mmgr/oset.c`. Los mismos problemas aparecieron con consultas largas cuando se usa el algoritmo normal de optimización de Postgres.

Mejora de las configuraciones de los parámetros del algoritmo genético

En el archivo `backend/optimizer/gego/gego_params.c`, rutinas `gimme_pool_size` y `gimme_number_generations`, ha de encontrarse un compromiso entre las configuraciones de parámetros para satisfacer dos demandas que compiten:

- Optimización del plan de consulta
- Tiempo de computación

Busqueda de una mejor solución para el desbordamiento de entero

En el archivo `backend/optimizer/gego/gego_eval.c`, rutina `gego_joinrel_size`, el valor para el desbordamiento MAXINT esta definido por el valor entero de Postgres, `rel->size` como su logaritmo. Una modificación de `Rel` en `backend/nodes/relation.h` tendrá seguramente impacto en la implementación global de Postgres.

Encotrar solución para la falta de memoria

La falta de memoria puede ocurrir cuando hay más de 10 relaciones involucradas en la consulta. El archivo `backend/optimizer/gego/gego_eval.c`, rutina `gimme_tree` es llamado recursivamente. Puede ser que olvidase algo para ser liberado correctamente, pero no se que es. Por supuesto la estructura de datos rel de la **unión** continua creciendo y creciendo; muchas relaciones están empaquetadas dentro de ella. Las sugerencias son bienvenidas :-)

Referencias

Información de referencia para algoritmos GEQ.

Notas

1. utesch@aut.tu-freiberg.de
1. <news://comp.ai.genetic>
2. <ftp://ftp.Germany.EU.net/pub/research/softcomp/EC/Welcome.html>

Capítulo 56. Protocolo Frontend/Backend

Nota: Escrito por Phil Thompson¹. Actualizaciones del protocolo por Tom Lane².

Postgres utiliza un protocolo basado en mensajes para la comunicación entre frontend y backends. El protocolo está implementado sobre TCP/IP y también sobre Unix sockets. Postgres v6.3 introdujo números de versión en el protocolo. Esto fue hecho de tal forma que aún permite conexiones desde versiones anteriores de los frontends, pero este documento no cubre el protocolo utilizado por esas versiones.

Este documento describe la versión 2.0 del protocolo, implementada en Postgres v6.4 y posteriores.

Las características de alto nivel sobre este protocolo (por ejemplo, como `libpq` pasa ciertas variables de entorno después de que la comunicación es establecida), son tratadas en otros lugares.

Introducción

Los tres principales componentes son el frontend (ejecutándose en el cliente) y el postmaster y backend (ejecutándose en el servidor). El postmaster y backend juegan diferentes roles pero pueden ser implementados por el mismo ejecutable.

Un frontend envía un paquete de inicio al postmaster. Este incluye los nombres del usuario y base de datos a la que el usuario quiere conectarse. El postmaster entonces utiliza esto, y la información en el fichero `pg_hba.conf`(5) para determinar que información adicional de autenticación necesita del frontend (si existe) y responde al frontend en concordancia.

El frontend envía entonces cualquier información de autenticación requerida. Una vez que el postmaster valida esta información responde al frontend que está autenticado y entrega una conexión a un backend. El backend entonces envía un mensaje indicando arranque correcto (caso normal) o fallo (por ejemplo, un nombre de base de datos inválido).

Las subsiguientes comunicaciones son paquetes de consulta y resultados intercambiados entre el frontend y backend. El postmaster no interviene ya en la comunicación ordinaria de consultas/resultados. Sin embargo el postmaster se involucra cuando el frontend desea cancelar una consulta que se esté efectuando en su backend. Más detalles sobre esto aparecen más abajo.

Cuando el frontend desea desconectar envía un paquete apropiado y cierra la conexión sin esperar una respuesta del backend.

Los paquetes son enviados como un flujo de datos. El primer byte determina que se debería esperar en el resto del paquete. La excepción son los paquetes enviados desde un frontend al postmaster, los cuales incluyen la longitud del paquete y el resto de él. Esta diferencia es histórica.

Protocolo

Esta sección describe el flujo de mensajes. Existen cuatro tipos diferentes de flujo dependiendo del estado de la conexión: inicio, consulta, llamada de función y final.

Existen también provisiones especiales para notificación de respuestas y cancelación de comandos, que pueden ocurrir en cualquier instante después de la fase de inicio.

Inicio

El inicio se divide en fase de autenticación y fase de arranque del backend.

Inicialmente, el frontend envía un `StartupPacket`. El postmaster utiliza esta información y el contenido del fichero `pg_hba.conf(5)` para determinar que método de autenticación debe emplear. El postmaster responde entonces con uno de los siguientes mensajes:

ErrorResponse

El postmaster cierra la comunicación inmediatamente.

AuthenticationOk

El postmaster entonces cede la comunicación al backend. El postmaster no toma parte en la comunicación posteriormente.

AuthenticationKerberosV4

El frontend debe tomar parte en un diálogo de autenticación Kerberos V4 (no descrito aquí) con el postmaster. En caso de éxito, el postmaster responde con un `AuthenticationOk`, en caso contrario responde con un `ErrorResponse`.

AuthenticationKerberosV5

El frontend debe tomar parte en un diálogo de autenticación Kerberos V5 (no descrito aquí) con el postmaster. En caso de éxito, el postmaster responde con un `AuthenticationOk`, en otro caso responde con un `ErrorResponse`.

AuthenticationUnencryptedPassword

El frontend debe enviar un `UnencryptedPasswordPacket`. Si este es el password correcto, el postmaster responde con un `AuthenticationOk`, en caso contrario responde con un `ErrorResponse`.

AuthenticationEncryptedPassword

El frontend debe enviar un `EncryptedPasswordPacket`. Si este es el password correcto, el postmaster responde con un `AuthenticationOk`, en caso contrario responde con un `ErrorResponse`.

Si el frontend no soporta el método de autenticación requerido por el postmaster, debería cerrar inmediatamente la conexión.

Después de enviar `AuthenticationOk`, el postmaster intenta lanzar un proceso backend. Como esto podría fallar, o el backend podría encontrar un error durante el arranque, el frontend debe esperar por una confirmación de inicio correcto del backend. El frontend no debería enviar mensajes en este momento. Los posibles mensajes procedentes del backend durante esta fase son:

BackendKeyData

Este mensaje es enviado después de un inicio correcto del backend. Proporciona una clave secreta que el frontend debe guardar si quiere ser capaz de enviar

peticiones de cancelación más tarde. El frontend no debería responder a este mensaje, pero podría continuar escuchando por un mensaje ReadyForQuery.

ReadyForQuery

El arranque del backend tuvo éxito. El frontend puede ahora enviar mensajes de peticiones o llamadas a función.

ErrorResponse

El arranque del backend no tuvo éxito. La conexión es cerrada después de enviar este mensaje.

NoticeResponse

Se envía un mensaje de advertencia. El frontend debería mostrar un mensaje pero debería continuar a la espera de un mensaje ReadyForQuery o ErrorResponse.

El mensaje ReadyForQuery es el mismo que el backend debe enviar después de cada ciclo de consulta. Dependiendo de las necesidades de codificado del frontend, es razonable considerar ReadyForQuery como iniciando un ciclo de consulta (y entonces BackendKeyData indica una conclusión correcta de la fase de inicio), o considerar ReadyForQuery como finalizando la fase de arranque y cada subsiguiente ciclo de consulta.

Consulta

Un ciclo de consulta se inicia por el frontend enviando un mensaje Query al backend. El backend entonces envía uno o más mensajes de respuesta dependiendo del contenido de la cadena de consulta, y finalmente un mensaje ReadyForQuery. ReadyForQuery informa al frontend que puede enviar una nueva consulta o llamada de función de forma segura.

Los posibles mensajes del backend son:

CompletedResponse

Una sentencia SQL se completó con normalidad.

CopyInResponse

El backend está preparado para copiar datos del frontend a una relación. El frontend debería enviar entonces un mensaje CopyDataRows. El backend responde con un mensaje CompletedResponse con un tag de "COPY".

CopyOutResponse

El backend está listo para copiar datos de una relación al frontend. El envía entonces un mensaje CopyDataRows, y un mensaje CompletedResponse con un tag de "COPY".

CursorResponse

La consulta fue bien un insert(l), delete(l), update(l), fetch(l) o una sentencia select(l). Si la transacción ha sido abortada entonces el backend envía un mensaje CompletedResponse con un tag "*ABORT STATE*". En otro caso las siguientes respuestas son enviadas.

Para una sentencia `insert(l)`, el backend envía un mensaje `CompletedResponse` con un tag de `"INSERT oid rows"` donde `rows` es el número de filas insertadas, y `oid` es el ID de objeto de la fila insertada si `rows` es 1, en otro caso `oid` es 0.

Para una sentencia `delete(l)`, el backend envía un mensaje `CompletedResponse` con un tag de `"DELETE rows"` donde `rows` es el número de filas borradas.

Para una sentencia `update(l)` el backend envía un mensaje `CompletedResponse` con un tag de `"UPDATE rows"` donde `rows` es el número de filas modificadas.

para una sentencia `fetch(l)` o `select(l)`, el backend envía un mensaje `RowDescription`. Es seguido después con un mensaje `AsciiRow` o `BinaryRow` (dependiendo de si fué especificado un cursor binario) para cada fila que es enviada al frontend. Por último, el backend envía un mensaje `CompletedResponse` con un tag de `"SELECT"`.

EmptyQueryResponse

Se encontro una caden de consulta vacía. (La necesidad de distinguir este caso concreto es histórica).

ErrorResponse

Ocurrió un error.

ReadyForQuery

El procesado de la cadena de consulta se completó. Un mensaje separado es enviado para indicar esto debido a que la cadena de consulta puede contener múltiples sentencias SQL. (`CompletedResponse` marca el final el procesado del una sentencia SQL, no de toda la cadena). Siempre se enviará `ReadyForQuery`, bien el procesado terminase con éxito o con error.

NoticeResponse

Un mensaje de advertencia fué enviado en relación con la consulta. Estas advertencias se envían en adición a otras respuestas, es decir, el backend continuará procesando la sentencia.

Un frontend debe estar preparado para aceptar mensaje `ErrorResponse` y `NoticeResponse` cuando se espere cualquier otro tipo de mensaje.

De hecho, es posible que `NoticeResponse` se reciba incluso cuando el frontend no está esperando ningún tipo de mensaje, es decir, cuando el backend está normalmente inactivo. En particular, el frontend puede solicitar la finalización del backend. En este caso se envía una `NoticeResponse` antes de cerrar la conexión. Se recomienda que el frontend compruebe esas advertencias asíncronas antes de enviar cada sentencia.

También, si el frontend envía cualquier comando `listen(l)`, entonces debe estar preparado para aceptar mensajes `NotificationResponse` en cualquier momento. Véase más abajo.

Llamada a función

Un ciclo de llamada a función se inicia por el frontend enviando un mensaje `FunctionCall` al backend. El backend entonces envía uno o más mensajes de respuesta dependiendo de los resultados de la llamada a función, y finalmente un mensaje

ReadyForQuery. ReadyForQuery informa al frontend que puede enviar una nueva consulta o llamada a función de forma segura.

Los posibles mensajes de respuesta provenientes de backend son:

ErrorResponse

Ocurrió un error.

FunctionResultResponse

La llamada a función fue ejecutada y devolvió un resultado.

FunctionVoidResponse

La llamada a función fue ejecutada y no devolvió resultados.

ReadyForQuery

El procesamiento de la llamada a función se completó. ReadyForQuery se enviará siempre, aunque el procesamiento termine con éxito o error.

NoticeResponse

Un mensaje de advertencia se generó en relación con la llamada a función. Estas advertencias aparecen en adición a otras respuestas, es decir, el backend continuará procesando el comando.

El frontend debe estar preparado para aceptar mensajes ErrorResponse y NoticeResponse cuando se esperen otro tipo de mensajes. También si envía cualquier comando listen(l) debe estar preparado para aceptar mensajes NotificationResponse en cualquier momento, véase más abajo.

Respuestas de notificación

Si un frontend envía un comando listen(l), entonces el backend enviará un mensaje NotificationResponse (no se confunda con NoticeResponse!) cuando un comando notify(l) sea ejecutado para el mismo nombre de notificación.

Las respuestas de notificación son permitidas en cualquier punto en el protocolo (después del inicio), excepto dentro de otro mensaje del backend. Así, el frontend debe estar preparado para reconocer un mensaje NotificationResponse cuando está esperando cualquier mensaje. De hecho debería ser capaz de manejar mensajes NotificationResponse incluso cuando no está envuelto en una consulta.

NotificationResponse

Un comando notify(l) ha sido ejecutado para un nombre para el que se ejecutó previamente un comando listen(l). Se pueden enviar notificaciones en cualquier momento.

Puede merecer la pena apuntar que los nombres utilizados en los comandos listen y notify no necesitan tener nada que ver con los nombres de relaciones (tablas) y bases de datos SQL. Los nombres de notificación son simplemente nombres arbitrariamente seleccionados.

Cancelación de peticiones en progreso

Durante el procesado de una consulta, el frontend puede solicitar la cancelación de la consulta mediante el envío de una petición apropiada al postmaster. La petición de cancelación no es enviada directamente al backend por razones de eficiencia de implementación: no deseamos tener al backend constantemente esperando nuevos datos del frontend durante el procesado de consultas. Las peticiones de cancelación deberían ser relativamente infrecuentes, por lo que las hacemos un poco mas voluminosas con el fin de evitar una penalización en el caso normal.

Para enviar una petición de cancelación, el frontend abre una nueva conexión con el postmaster y envía un mensaje `CancelRequest`, en vez del mensaje `StartupPacket` que enviaría normalmente en una nueva conexión. El postmaster procesará esta petición y cerrará la conexión. Por razones de seguridad, no se envía una respuesta directa al mensaje de cancelación.

Un mensaje `CancelRequest` será ignorado a menos que contenga los mismos datos clave (PID y clave secreta) enviados al frontend durante el inicio de la conexión. Si la petición contiene el PID e clave secreta el backend aborta el procesado de la consulta actual.

La señal de cancelación puede tener o no tener efectos - por ejemplo, si llega despues de que el backend haya finalizado de procesar la petición, entonces no tendrá efecto. Si la cancelación es efectiva, produce la terminación prematura del comando actual dando un mensaje de error.

La consecuencia de todo esto es que por razones tanto de seguridad como de eficiencia, el frontend no tiene forma directa de decidir cuando una petición de cancelación tuvo éxito. Debe continuar esperando hasta que el backend responda a al petición. Enviar unha petición de cancelación simplemente aumenta las probabilidades de que la consulta actual finalice pronto, y aumenta las probabilidades de que falle con un mensaje de error en vez de terminar con éxito.

Ya que la petición de cancelación es enviada al postmaster y no a través del enlace normal frontend/backend, es posible que cualquier proceso realice la petición, no sólo el frontend cuya consulta va a ser cancelada. Esto puede tener algún beneficio de cara a aumentar la flexibilidad al diseñar aplicaciones multi-proceso. Tambien introduce un riesgo de seguridad, ya que personas no autorizadas podrían intentar cancelar consultas. El riesgo de seguridad es afrontado requiriendo la clave secreta generada dinámicamente.

Finalización

El procedimiento de finalización normal es que el frontend envíe un mensaje `Terminate` y cierre inmediatamente la conexión. Al recibir el mensaje, el backend cierra inmediatamente la conexión y finaliza.

Una finalización anormal puede ocurrir debido a fallos de software (i.e. core dump) en cualquier extremo. Si el frontend o el backend ve un cierre inexperado de la conexión, debería liberar recursos y finalizar. El frontend tiene la opción de lanzar un nuevo backen recontactando el postmaster, si lo desea.

Tipos de Datos de Mensajes

Esta sección describo los tipos básicos de datos utilizados en los mensajes.

`Int n (i)`

Un entero n en orden de bytes de red. Si i está especificado es el valor literal. P.e. `Int16`, `Int32(42)`.

`LimString n (s)`

Un array de caracteres de exactamente n bytes interpretado como una cadena terminada en `'\0'`. El `'\0'` se omite si no existe espacio suficiente. Si s está especificado entonces es el valor literal. P.e. `LimString32`, `LimString64("user")`.

`String(s)`

Una cadena de C convencional terminada en `'\0'` sin limitación de longitud. Si s está especificada es el valor literal. P.e. `String`, `String("user")`.

Nota: *No existe límite predefinido* para la longitud de una cadena que puede ser retornada por el backend. Una buena estrategia a utilizar por el frontend consiste en usar un buffer expandible para que cualquier cosa que quepa en memoria pueda ser aceptada. Si esto no es posible, se debe leer toda la cadena y deshechar los caracteres que no quepan en el buffer de longitud fija.

`Byte n (c)`

Exactamente n bytes. Si c está especificado es el valor literal. P.e. `Byte`, `Byte1('\n')`.

Formatos de Mensajes

Esta sección describe el formato detallado de cada mensaje. Cada uno puede ser enviado por un frontend (F), por un backend (B) o por ambos (F y B).

`AsciiRow (B)`

`Byte1('D')`

Identifica el mensaje como una fila de datos ASCII. (Un mensaje previo `RowDescription` define el número de campos en la fila y sus tipos de datos).

`Byte n`

Un mapa de bits con un bit para cada campo en la fila. El primer campo corresponde al bit 7 (MSB) del primer byte, el segundo campo corresponde al bit 6 del primer byte, el octavo campo corresponde al bit 0 (LSB) del primer byte, el noveno campo corresponde al bit 7 del segundo byte, y así sucesivamente. Cada bit está activo si el valor del campo correspondiente no es NULL. Si el número de campos no es un múltiplo de 8, el resto del último byte en el mapa de bits no es utilizado.

Por lo tanto, para cada campo con un valor no NULL, tenemos lo siguiente:

Int32

Especifica el tamaño del valor del campo, incluyendo este tamaño.

Byten

Especifica el valor del campo mismo en caracteres ASCII. *n* es el anterior tamaño menos 4. No hay '\0' al final del campo de datos, el frontend debe añadirlo si quiere uno.

AuthenticationOk (B)

Byte1('R')

Identifica el mensaje como una petición de autenticación.

Int32(0)

Especifica que la autenticación tuvo éxito.

AuthenticationKerberosV4 (B)

Byte1('R')

Identifica el mensaje como una petición de autenticación.

Int32(1)

Especifica que se requiere autenticación Kerberos V4.

AuthenticationKerberosV5 (B)

Byte1('R')

Identifica el mensaje como una petición de autenticación.

Int32(2)

Especifica que se requiere autenticación Kerberos V5.

AuthenticationUnencryptedPassword (B)

Byte1('R')

Identifica el mensaje como una petición de autenticación.

Int32(3)

Especifica que se requiere una contraseña no encriptada.

AuthenticationEncryptedPassword (B)

Byte1('R')

Identifica el mensaje como una petición de autenticación.

Int32(4)

Especifica que se requiere una contraseña encriptada.

Byte2

El salto a utilizar al encriptar la contraseña.

BackendKeyData (B)

Byte1('K')

Identifica el mensaje como una clave de cancelación. El frontend debe guardar estos valores si desea poder enviar mensajes CancelRequest posteriormente.

Int32

El ID de proceso del backend.

Int32

La clave secreta de este backend.

BinaryRow (B)

Byte1('B')

Identifica el mensaje como una fila de datos binarios. (Un mensaje RowDescription previo define el número de campos en la fila y sus tipos de datos)

Byten

Un mapa de bits con un bit para cada campo en la fila. El primer campo corresponde al bit 7 (MSB) del primer byte, el segundo campo corresponde al bit 6 del primer byte, el octavo campo corresponde al bit 0 (LSB) del primer byte, el noveno campo corresponde al bit 7 del segundo byte, y así sucesivamente. Cada bit está activo si el valor del campo correspondiente no es NULL. Si el número de campos no es un múltiplo de 8, el resto del último byte en el mapa de bits no es utilizado.

Para cada campo con un valor distinto de NULL, tenemos lo siguiente:

Int32

Especifica el tamaño del valor del campo, excluyendo este tamaño.

Comprobar esto, por que aquí dice `_excluyendo_` y antes (línea 756) dice incluyendo????????????*****

Byte*n*

Especifica el valor del campo mismo en formato binario. *n* es el tamaño previo.

CancelRequest (F)**Int32(16)**

El tamaño del paquete en bytes.

Int32(80877102)

El código de cancelación de petición. El valor es elegido para que contenga "1234" en los 16 bits más significativos, y "5678" en los 16 bits menos significativos. Para evitar confusión, este código no debe ser el mismo que ningún número de versión del protocolo.

Int32

El ID de proceso del backend objetivo.

Int32

La clave secreta para el backend objetivo.

CompletedResponse (B)**Byte1('C')**

Identifica este mensaje como una petición completada.

String

El comando. Normalmente (pero no siempre) una palabra simple que identifica que comando SQL se completó.

CopyDataRows (B y F)

Es un flujo de filas donde cada una está terminada por un Byte1('\n'). Se completa con una secuencia Byte1('\'), Byte1('.'), Byte1('\n').

CopyInResponse (B)

Byte1('G')

Identifica el mensaje como una respuesta Start Copy In. El frontend debe enviar un mensaje CopyDataRows.

CopyOutResponse (B)

Byte1('H')

Identifica el mensaje como una respuesta Start Copy Out. Este mensaje será seguido por un mensaje CopyDataRows.

CursorResponse (B)

Byte1('P')

Identifica el mensaje como un cursor.

String

El nombre del cursor. Será "blanco" si el cursor es implícito.

EmptyQueryResponse (B)

Byte1('I')

Identifica este mensaje como una respuesta a una sentencia vacía.

String("")

Sin utilizar.

EncryptedPasswordPacket (F)

Int32

El tamaño del paquete en bytes.

String

La contraseña encriptada (mediante crypt()).

ErrorResponse (B)

Byte1('E')

Identifica el mensaje como un error.

String

El mensaje de error mismo.

FunctionCall (F)

Byte1('F')

Identifica el mensaje como una llamada a función.

String("")

Sin utilizar.

Int32

Especifica el ID de objeto de la función a llamar.

Int32

Especifica el número de argumentos que se suministran a la función.

Para cada argumento, se tiene lo siguiente:

Int32

Especifica el tamaño del valor del argumento, excluyendo este tamaño.

Byten

Especifica el valor del campo mismo en formato binario. *n* es el tamaño anterior.

FunctionResultResponse (B)

Byte1('V')

Identifica el mensaje como un resultado de llamada a función.

Byte1('G')

Especifica que se devolvió un resultado no vacío.

Int32

Especifica el tamaño del valor del resultado, excluyendo este tamaño.

Byten

Especifica el valor del resultado en formato binario. *n* Es el tamaño anterior.

Byte1('0')

Sin utilizar. (Hablando propiamente, FunctionResultResponse y Function-VoidResponse son lo mismo pero con algunas partes opcionales en el mensaje).

FunctionVoidResponse (B)

Byte1('V')

Identifica el mensaje como un resultado de llamada a función.

Byte1('0')

Especifica que se devolvió un resultado vacío.

NoticeResponse (B)

Byte1('N')

Identifica el mensaje como una advertencia.

String

El mensaje de advertencia mismo.

NotificationResponse (B)

Byte1('A')

Identifica el mensaje como una respuesta de notificación.

Int32

El ID de proceso del proceso backend.

String

El nombre de la condición en la que se lanzó la notificación.

Query (F)

Byte1('Q')

Identifica el mensaje como una petición.

String

La petición misma.

ReadyForQuery (B)

Byte1('Z')

Identifica el tipo de mensaje. ReadyForQuery es enviado cuando el backend está listo para un nuevo ciclo de petición.

RowDescription (B)

Byte1('T')

Identifica el mensaje como una descripción de fila.

Int16

Especifica el número de campos en una fila (puede ser cero).

Para cada campo tenemos lo siguiente:

String

Especifica el nombre del campo.

Int32

Especifica el ID de objeto del tipo de campo.

Int16

Especifica el tamaño del tipo.

Int32

Especifica el modificador del tipo.

StartupPacket (F)

Int32(296)

El tamaño del paquete en bytes.

Int32

El número de versión del protocolo. Los 16 bits más significativos son el número de versión mayor. Los 16 bits menos significativos son el número de versión menor.

LimString64

El nombre de la base de datos, por defecto el nombre del usuario si no se especifica.

LimString32

El nombre del usuario.

LimString64

Cualquier línea de argumentos para pasar al backend por el postmaster.

LimString64

Sin utilizar.

LimString64

La tty opcional que el backen debería utilizar para mensajes de depuración.

Terminate (F)

Byte1('X')

Identifica el mensaje como una terminación.

UnencryptedPasswordPacket (F)

Int32

El tamaño del paquete en bytes.

String

La contraseña sin encriptar.

Notas

1. <mailto:phil@river-bank.demon.co.uk>
2. <mailto:tgl@sss.pgh.pa.us>

Capítulo 57. Señales de Postgres

Nota: Contribución de Massimo Dal Zotto¹

Postgres usa las siguientes señales para la comunicación entre el postmaster y los backends:

Tabla 57-1. Señales Postgres

Signal	Acción postmaster	Acción del servidor
SIGHUP	kill(*,sighup)	read_pg_options
SIGINT	die	cancela la consulta
SIGQUIT	kill(*,sigterm)	handle_warn
SIGTERM	kill(*,sigterm), kill(*,9), die	muerte
SIGPIPE	ignored	muerte
SIGUSR1	kill(*,sigusr1), die	muerte rápida
SIGUSR2	kill(*,sigusr2)	notificación asíncrona
SIGCHLD	reaper	ignorado (prueba de f
SIGTTIN	ignorado	
SIGTTOU	ignorado	
SIGCONT	dumpstatus	
SIGFPE		FloatExceptionHandler

Nota: “kill(*,signal)” significa enviar una señal a todo los backends.

Los principales cambios del viejo gestor de señal es el uso de SIGQUIT en lugar de SIGHUP para gestionar los avisos, SIGHUP intenta releer el fichero de pg_options y lo redirecciona a todos los backends activos de SIGHUP, SIGTERM, SIGUSR1 y SIGUSR2 llamados por el postmaster. Por este camino estas señales enviada al postmaster pueden ser enviadas automáticamente hacia todos los backends sin necesidad de conocer sus pids. Para bajar postgres lo único que se necesita es enviar un SIGTERM al postmaster y esto parará automáticamente todos los backends.

La señal SIGUSR2 es también usado para prevenir el desbordamiento del cache de la tabla SI esto pasa cuando algún backend no procesa la cache SI durante un largo periodo de tiempo. Cuando el backend detecta que la tabla SI esta a mas de un 70% simplemente envía una señal al postmaster el cual despertará a todos los backends desocupados y los hace que vacíe el cache.

El uso típico de las señales por los programadores puede ser el siguiente:

```
# stop postgres
kill -TERM $postmaster_pid
```

```
# kill all the backends
kill -QUIT $postmaster_pid

# kill only the postmaster
kill -INT $postmaster_pid

# change pg_options
cat new_pg_options > $DATA_DIR/pg_options
kill -HUP $postmaster_pid

# change pg_options only for a backend
cat new_pg_options > $DATA_DIR/pg_options
kill -HUP $backend_pid
cat old_pg_options > $DATA_DIR/pg_options
```

Notas

1. <mailto:dz@cs.unitn.it>

Capítulo 58. gcc Default Optimizations

Nota: Contributed by Brian Gallew¹

Para configurar gcc para usar ciertas opciones por defecto, simplemente hay que editar el fichero `/usr/local/lib/gcc-lib/platform/version/specs`. El formato de este fichero es bastante simple. El fichero está dividido en secciones, cada una de tres líneas de longitud. La primera es `"*section_name:"` (e.g. `"*asm:"`). La segunda es una línea de opciones, y la tercera es una línea en blanco.

El cambio más sencillo es añadir las opciones deseadas a la lista en la sección apropiada. Por ejemplo, supongamos que tenemos Linux ejecutandose en un 486 con gcc 2.7.2 instalado en su lugar por defecto. En el fichero `/usr/local/lib/gcc-lib/i486-linux/2.7.2/specs`, 13 líneas más abajo se encuentra la siguiente sección:

```
- -----SECTION-----  
*ccl:
```

```
- -----SECTION-----
```

Como puede verse, no hay ninguna opción por defecto. Si siempre compila código C usando `"-m486 -fomit-frame-pointer"`, tendría que cambiarlo de este modo:

```
- -----SECTION-----  
*ccl:  
- -m486 -fomit-frame-pointer
```

```
- -----SECTION-----
```

Si quiero poder generar código 386 para otro equipo Linux más antiguo que tenga por ahí, tendríamos que hacer algo así:

```
- -----SECTION-----  
*ccl:  
%{!m386:-m486} -fomit-frame-pointer
```

```
- -----SECTION-----
```

Esto omite siempre los punteros de marco; se construirá código optimizado para 486 a menos que se especifique `-m386` en la línea de ordenes.

Pueden realizarse bastantes personalizaciones usando el fichero `spect`. Sin embargo, reuerde siempre que esos cambios son globales, y afectarán a todos los usuarios del sistema.

Notas

1. <mailto:geek+@cmu.edu>

Capítulo 59. Interfaces de Backend

Los ficheros de interfaces (BKI) son scripts que sirven de entrada para los backend de Postgres ejecutándose en un modo especial "bootstrap" permite realizar funciones de base de datos sin que exista todavía el sistema de la base de datos. Los ficheros BKI pueden por lo tanto ser usados para crear el sistema de base de datos al inicio. `initdb` usa ficheros BKI para hacer exactamente eso: crear el sistema de base de datos. De cualquier manera, los ficheros BKI de `initdb` son generalmente internos. Los genera usando los ficheros `global1.bki.source` y `local1.template1.bki.source`, que se encuentran en el directorio de "librerías" de Postgres. Estos se instalan aquí como parte de la instalación de Postgres. Estos ficheros `.source` se generan como parte del proceso de construcción de Postgres, construido por un programa llamado `genbki`. `genbki` toma como entrada los ficheros fuente de Postgres que sirven como entrada de `genbki` que construye tablas y ficheros de cabecera de C que describen estas tablas.

Se puede encontrar información al respecto en la documentación de `initdb`, `createdb`, y en el comando de SQL **CREATE DATABASE**.

Formato de fichero BKI

Los backend de Postgres interpretan los ficheros BKI como se describe abajo. Esta descripción será más fácil de entender si cogemos el fichero `global1.bki.source` como ejemplo. (como se explica arriba, este fichero `.source` no es exactamente un fichero BKI, pero de todos modos le servirá para comprender el resultado si lo fuese.

Los comandos estan compuestos por un identificador seguido por argumentos separados por espacios. Los argumentos de los comandos que comienzan por "\$" se tratan de forma especial. Si "\$\$" son los primeros dos caracteres, entonces el primer "\$" se ignora y el argumento se procesa normalmente. Si el "\$" va seguido por espacio, entonces se trata como un valor NULL. De otro modo, los caracteres seguidos de "\$" se interpretan como el nombre de una macro, lo que provoca que el argumento se reemplace por el valor de la macro. Si la macro no está definida se genera un error.

Las macros se definen usando

```
define macro macro_name = macro_value
```

y se quita la definición usando

```
undefine macro macro_name
```

y se redefine usando la misma sintaxis que en la definición.

Seguidamente se listan los comandos generales y los comandos de macro.

Comandos Generales

OPEN *classname*

Abre la clase llamada *classname* para futuras manipulaciones.

CLOSE [*classname*]

Cierra la clase abierta llamada *classname*. Se genera un error si *classname* no está actualmente abierta. Si no aparece *classname*, entonces la clase que actualmente está abierta se cierra.

PRINT

Imprime la clase que actualmente está abierta.

INSERT [OID=*oid_value*] (*value1 value2* ...)

Inserta una nueva instancia para la clase abierta usando *value1*, *value2*, etc., como valores de los atributos y *oid_value* como OID. Si *oid_value* no es "0", entonces este valor se usará como identificador del objeto instancia. De otro modo, provoca un error.

INSERT (*value1 value2* ...)

Como arriba, pero el sistema genera un identificador de objeto único.

CREATE *classname* (*name1* = *type1* [, *name2* = *type2* [...]])

Crea una clase llamada *classname* con los atributos introducidos entre paréntesis.

OPEN (*name1* = *type1* [, *name2* = *type2* [...]]) AS *classname*

Abre una clase llamada *classname* para escritura pero no graba su existencia en los catálogos de sistema. (Esto es primordialmente lo que ayuda al bootstrapping.)

DESTROY *classname*

Destruye la clase llamada *classname*.

DEFINE INDEX *indexname* ON *class_name* USING *amname* (*opclass attr* | (*function(attr)*)

Crea un índice llamado *indexname* para la clase llamada *classname* usando el método de acceso *amname*. Los campos se llaman *name1*, *name2* etc., y los operadores de recogida que usa son *collection_1*, *collection_2* etc., respectivamente.

Nota: Esta última sentencia no referencia a nada del ejemplo. Debería ser cambiado para que tenga sentido. - Thomas 1998-08-04

Macro Commands

DEFINE FUNCTION *macro_name* AS *rettype function_name*(*args*)

Define un prototipo de función para la función llamada *macro_name* la cual tienen el tipo de valor *rettype* calculada desde la ejecución de *function_name* con los argumentos *args* declarados a la manera de C.

DEFINE MACRO *macro_name* FROM FILE *filename*

Define una macro llamada *macro_name* la cual tendrá un valor que se leerá del archivo *filename*.

Comandos de Depuración

Nota: Esta sección de los comandos de depuración fue comentada por completo en la documentación original. Thomas 1998-08-05

r
Visualiza aleatoriamente la clase abierta.

m -1
Cambia la visualización de la información del tiempo.

m 0
Activa la recuperación inmediatamente.

m 1 Jan 1 01:00:00 1988
Activa la recuperación de la foto para un tiempo específico.

m 2 Jan 1 01:00:00 1988, Feb 1 01:00:00 1988
Activa la recuperación para un rango específico de tiempo. Ambos tiempos deben ser reemplazados por un espacio en blanco si el rango de tiempo deseado es ilimitado.

&A *classname natts name1 type1 name2 type2 . . .*
Añade atributos 'chivato' llamados *name1*, *name2*, etc. de tipos *type1*, *type2*, etc. para la clase *classname*.

&RR *oldclassname newclassname*
Renombra la clase *oldclassname* por *newclassname*.

&RA *classname oldattname newattname classname oldattname newattname*
Renombra el atributo *oldattname* de la clase llamada *classname* por el *newattname*.

Ejemplo

El siguiente conjunto de comandos creará la clase "pg_opclass" conteniendo una colección de *int_ops* como un objeto con un OID igual a 421, visualiza la clase, y después la cierra.

```
create pg_opclass (opcname=name)
open pg_opclass
insert oid=421 (int_ops)
print
close pg_opclass
```


Capítulo 60. Ficheros de páginas.

Una descripción del formato de página de defecto de los ficheros de la base de datos.

Esta sección proporciona una visión general del formato de página utilizado por las clases de Postgres. Los métodos de acceso definidos por el usuario no necesitan utilizar este formato de página.

En la siguiente explicación, se asume que un *byte* contiene 8 bits. Además, el término *campo* se refiere a los datos almacenados en una clase de Postgres.

Estructura de la página.

La siguiente tabla muestra como están estructuradas las páginas tanto en las clases normales de Postgres como en las clases de índices de Postgres (es decir, un índice B-tree).

Tabla 60-1. Muestra de Dibujo de Página

Campo	Descripción
Puntero a Datos (ItemPointerData)	
Espacio Libre (filler)	
Campo de datos....	
Espacio desocupado	
Campo de Continuación de Datos (ItemContinuationData)	
Espacio Especial	
“Campo de datos 2”	
“Campo de datos 1”	
Datos de Identificación de Campo (ItemIdData)	
Datos de Cabecera de Página (PageHeaderData)	

Los primeros 8 bytes de cada página consisten en la cabecera de la página (PageHeaderData). Dentro de la cabecera, los primeros tres campos enteros de 2 bytes (*menor*, *mayor* y *especial*) representan bytes que reflejan el principio del espacio desocupado, el final del espacio desocupado, y el principio del *espacio especial*. El espacio especial es una región al final de la página que se ocupa en la inicialización de la página y que contiene información específica sobre un método de acceso. Los dos últimos 2 bytes de la cabecera de página, *opaco*, codifica el tamaño de la página e información sobre la fragmentación interna de la misma. El tamaño de la página se almacena en cada una de ellas, porque las estructuras del pool de buffers pueden estar subdivididas en una forma estructura por estructura dentro de una clase. La información sobre la fragmentación interna se utiliza para ayudar a determinar cuando debería realizarse la reorganización de la página.

Siguiendo a la cabecera de la página están los identificadores de campo (ItemIdData). Se sitúan nuevos identificadores de campo a partir de los primeros cuatro bytes de espacio libre. Debido a que un identificador de campo nunca se mueve hasta que se elimina, este índice se puede utilizar para indicar la situación de un campo en la página.

gina. De hecho, cada puntero a un campo (*ItemPointer*) creado por Postgres consiste en un número de estructura y un índice de un identificador de campo. Un identificador de campo contiene un byte de referencia al principio de un campo, su longitud en bytes, y un conjunto de bits de atributos que pueden afectar a su interpretación.

Los campos mismos están almacenados en un espacio situado más allá del final del espacio libre. Habitualmente, los campos no son interpretados. Sin embargo, cuando el campo es demasiado largo para ser situado en una única página o cuando se desea la fragmentación del campo, éste mismo se divide y cada parte se manipula como campos distintos de la siguiente manera. Cada una de las partes en que se descompone se sitúa en una estructura de continuación de campo (*ItemContinuationData*). Esta estructura contiene un puntero (*ItemPointerData*) hacia la siguiente parte. La última de estas partes se manipula normalmente.

Ficheros

`data/`

Localización de los ficheros de base de datos compartidos (globales).

`data/base/`

Localización de los ficheros de base de datos locales.

Bugs

El formato de la página puede cambiar en el futuro para proporcionar un acceso más eficiente a los objetos largos.

Esta sección contiene detalles insuficiente para ser de alguna asistencia en la escritura de un nuevo método de acceso.

Capítulo 61. SQL

Este capítulo apareció originariamente como parte de la tesis doctoral de Stefan Simkovics. (*Simkovics, 1998*).

SQL se ha convertido en el lenguaje de consulta relacional más popular. El nombre "SQL" es una abreviatura de *Structured Query Language* (Lenguaje de consulta estructurado). En 1974 Donald Chamberlain y otros definieron el lenguaje SEQUEL (*Structured English Query Language*) en IBM Research. Este lenguaje fue implementado inicialmente en un prototipo de IBM llamado SEQUEL-XRM en 1974-75. En 1976-77 se definió una revisión de SEQUEL llamada SEQUEL/2 y el nombre se cambió a SQL.

IBM desarrolló un nuevo prototipo llamado System R en 1977. System R implementó un amplio subconjunto de SEQUEL/2 (now SQL) y un número de cambios que se le hicieron a (now SQL) durante el proyecto. System R se instaló en un número de puestos de usuario, tanto internos en IBM como en algunos clientes seleccionados. Gracias al éxito y aceptación de System R en los mismos, IBM inició el desarrollo de productos comerciales que implementaban el lenguaje SQL basado en la tecnología System R.

Durante los años siguientes, IBM y bastantes otros vendedores anunciaron productos SQL tales como SQL/DS (IBM), DB2 (IBM), ORACLE (Oracle Corp.), DG/SQL (Data General Corp.), y SYBASE (Sybase Inc.).

SQL es también un estándar oficial hoy. En 1982, la American National Standards Institute (ANSI) encargó a su Comité de Bases de Datos X3H2 el desarrollo de una propuesta de lenguaje relacional estándar. Esta propuesta fue ratificada en 1986 y consistía básicamente en el dialecto de IBM de SQL. En 1987, este estándar ANSI fue también aceptado por la Organización Internacional de Estandarización (ISO). Esta versión estándar original de SQL recibió informalmente el nombre de "SQL/86". En 1989, el estándar original fue extendido, y recibió el nuevo nombre, también informal, de "SQL/89". También en 1989 se desarrolló un estándar relacionado llamado *Database Language Embedded SQL* (ESQL).

Los comités ISO y ANSI han estado trabajando durante muchos años en la definición de una versión muy ampliada del estándar original, llamado informalmente SQL2 o SQL/92. Esta versión se convirtió en un estándar ratificado durante 1992: *International Standard ISO/IEC 9075:1992, Database Language SQL*. SQL/92 es la versión a la que normalmente la gente se refiere cuando habla de «SQL estándar». Se da una descripción detallada de SQL/92 en *Date and Darwen, 1997*. En el momento de escribir este documento, se está desarrollando un nuevo estándar denominado informalmente como SQL3. Se plantea hacer de SQL un lenguaje de alcance completo (e Turing-complete language), es decir, serán posibles todas las consultas computables, (por ejemplo consultas recursivas). Esta es una tarea muy compleja y por ello no se debe esperar la finalización del nuevo estándar antes de 1999.

El Modelo de Datos Relacional

Como mencionamos antes, SQL es un lenguaje relacional. Esto quiere decir que se basa en el *modelo de datos relacional* publicado inicialmente por E.F.Codd en 1970. Daremos una descripción formal del modelo de datos relacional más tarde (en *Formalidades del Modelo Relacional de Datos*), pero primero queremos dar una mirada desde un punto de vista más intuitivo.

Una *base de datos relacional* es una base de datos que se percibe por los usuarios como una *colección de tablas* (y nada más que tablas). Una tabla consiste en filas y columnas, en las que cada fila representa un registro, y cada columna representa un atributo del registro contenido en la tabla. La *Base de Datos de Proveedores y Artículos* muestra un ejemplo de base de datos consistente en tres tablas.

- SUPPLIER es una tabla que recoge el número (SNO), el nombre (SNAME) y la ciudad (CITY) de un proveedor.
- PART es una tabla que almacena el número (PNO) el nombre (PNAME) y el precio (PRICE) de un artículo.
- SELLS almacena información sobre qué artículo (PNO) es vendido por qué proveedor (SNO). Esto sirve en un sentido para conectar las dos tablas entre ellas.

Ejemplo 61-1. La Base de Datos de Proveedores y Artículos

SUPPLIER	SNO	SNAME	CITY	SELLS	SNO	PNO
	1	Smith	London		1	1
	2	Jones	Paris		1	2
	3	Adams	Vienna		2	4
	4	Blake	Rome		3	1
					3	3
					4	2
					4	3
PART	PNO	PNAME	PRICE			
	1	Tornillos	10	4	4	
	2	Tuercas	8			
	3	Cerrosjos	15			
	4	Levas	25			

Las tablas PART y SUPPLIER se pueden ver como *entidades* y SELLS se puede ver como una *relación* entre un artículo particular y un proveedor particular.

Como veremos más tarde, SQL opera en las tablas tal como han sido definidas, pero antes de ello estudiaremos la teoría del modelo relacional.

Formalidades del Modelo Relacional de Datos

El concepto matemático que subyace bajo el modelo relacional es la *relación* de la teoría de conjuntos, la cual es un subconjunto del producto cartesiano de una lista de dominios. Esta relación de la teoría de conjuntos proporciona al modelo su nombre (no confundir con la relación del *Modelo Entidad-Relación*). Formalmente, un dominio es simplemente un conjunto de valores. Por ejemplo, el conjunto de los enteros es un dominio. También son ejemplos de dominios las cadenas de caracteres de longitud 20 y los números reales.

El *producto cartesiano* de los dominios D_1, D_2, \dots, D_k , escritos $D_1 \times D_2 \times \dots \times D_k$ es el conjunto de las k -tuplas v_1, v_2, \dots, v_k tales que $v_1 \in D_1, v_2 \in D_2, \dots, v_k \in D_k$.

Por ejemplo, cuando tenemos $k=2, D_1=\{0, 1\}$ y $D_2=\{a, b, c\}$ entonces $D_1 \times D_2$ es $\{(0, a), (0, b), (0, c), (1, a), (1, b), (1, c)\}$.

Una Relación es cualquier subconjunto del producto cartesiano de uno o más dominios: $R \subseteq D_1 \times D_2 \times \dots \times D_k$.

Por ejemplo, $\{(0, a), (0, b), (1, a)\}$ es una relación; De hecho es un subconjunto de $D_1 \times D_2$ mencionado antes.

Los miembros de una relación se llaman tuplas. Cada relación de algún producto cartesiano $D_1 \times D_2 \times \dots \times D_k$ se dice que tiene nivel k y de este modo es un subconjunto de k -tuplas.

Una relación se puede ver como una tabla (como ya dijimos, recuerde *La Base de Datos de Proveedores y Artículos* donde cada tupla se representa como una fila y cada columna corresponde a un componente de la tupla. Dando nombres (llamados atributos) a las columnas, nos acercamos a la definición de un *esquema relacional*.

Un *esquema relacional* R es un conjunto finito de atributos A_1, A_2, \dots, A_k . Hay un dominio D_i , para cada atributo A_i , $1 \leq i \leq k$, de donde se toman los valores de los atributos. Entonces escribimos el esquema relacional como $R(A_1, A_2, \dots, A_k)$.

Nota: Un *esquema relacional* es sólo un juego de plantillas mientras que una *relación* es un ejemplo de un *esquema relacional*. La relación consiste en las tuplas (y pueden ser vistas como una tabla); no así el esquema relacional.

Dominios contra Tipos de Datos

Ya hemos hablado de *dominios* en la sección anterior. Recalcar que el dominio es, formalmente, un conjunto de valores (por ejemplo el conjunto de los enteros o el de los números reales). En términos de sistemas de base de datos, hemos hablado de *tipos de datos* más que de dominios. Cuando hemos definido una tabla, hemos tomado una decisión sobre qué atributos incluir. Adicionalmente, hemos decidido qué juego de datos deberá ser almacenado en valores de los atributos. Por ejemplo, los valores de *SNAME* de la tabla *SUPPLIER* serán cadenas de caracteres, mientras que *SNO* almacenará enteros. Definimos esto asignando un tipo de datos a cada atributo. El tipo de *SNAME* será *VARCHAR(20)* (este es el tipo SQL para cadenas de caracteres de longitud ≤ 20), el tipo de *SNO* será *INTEGER*. Con la asignación de tipos de datos, también habremos seleccionado un dominio para un atributo. El dominio de *SNAME* es el conjunto de todas las cadenas de caracteres de longitud ≤ 20 , mientras el dominio de *SNO* es el conjunto de todos los números enteros.

Operaciones en el Modelo de Datos Relacional

En la sección previa (*Formalidades del Modelo Relacional de Datos*) definimos la noción matemática del modelo relacional. Ahora conocemos como los datos pueden almacenarse utilizando un modelo de datos relacional, pero no conocemos qué podemos hacer con todas estas tablas para recuperar algo desde esa base de datos todavía. Por ejemplo, alguien podría preguntar por los nombres de todos los proveedores que vendan el artículo 'tornillo'. Hay dos formas diferentes de notaciones para expresar las operaciones entre relaciones.

- El *Álgebra Relacional* es una notación algebraica, en la cual las consultas se expresan aplicando operadores especializados a las relaciones.
- El *Cálculo Relacional* es una notación lógica, donde las consultas se expresan formulando algunas restricciones lógicas que las tuplas de la respuesta deban satisfacer.

Álgebra Relacional

El *Álgebra Relacional* fue introducida por E.F.Codd en 1972. Consiste en un conjunto de operaciones con las relaciones.

- **SELECT (σ):** extrae *tuplas* a partir de una relación que satisfagan una restricción dada. Sea R una tabla que contiene un atributo A . $\sigma_{A=a}(R) = \{t \in R \mid t(A) = a\}$ donde t denota una tupla de R y $t(A)$ denota el valor del atributo A de la tupla t .
- **PROJECT (π):** extrae *atributos* (columnas) específicos de una relación. Sea R una relación que contiene un atributo X . $\pi_X(R) = \{t(X) \mid t \in R\}$, donde $t(X)$ denota el valor del atributo X de la tupla t .
- **PRODUCT (\times):** construye el producto cartesiano de dos relaciones. Sea R una tabla de rango (arity) k_1 y sea S una tabla con rango (arity) k_2 . $R \times S$ es el conjunto de las $k_1 + k_2$ -tuplas cuyos primeros k_1 componentes forman una tupla en R y cuyos últimos k_2 componentes forman una tupla en S .
- **UNION (\cup):** supone la unión de la teoría de conjuntos de dos tablas. Dadas las tablas R y S (y ambas deben ser del mismo rango), la unión $R \cup S$ es el conjunto de las tuplas que están en R o en las dos.
- **INTERSECT (\cap):** Construye la intersección de la teoría de conjuntos de dos tablas. Dadas las tablas R y S , $R \cap S$ es el conjunto de las tuplas que están en R y en S . De nuevo requiere que R y S tengan el mismo rango.
- **DIFFERENCE ($-$ or \setminus):** supone el conjunto diferencia de dos tablas. Sean R y S de nuevo dos tablas con el mismo rango. $R - S$ Es el conjunto de las tuplas que están en R pero no en S .
- **JOIN (\Join):** conecta dos tablas por sus atributos comunes. Sea R una tabla con los atributos A, B y C y sea S una tabla con los atributos C, D y E . Hay un atributo común para ambas relaciones, el atributo C . $R \Join S = \pi_{R.A, R.B, R.C, S.D, S.E}(\sigma_{R.C=S.C}(R \times S))$. ¿Qué estamos haciendo aquí? Primero calculamos el producto cartesiano $R \times S$. Entonces seleccionamos las tuplas cuyos valores para el atributo común C sea igual ($\sigma_{R.C=S.C}$). Ahora tenemos una tabla que contiene el atributo C dos veces y lo corregimos eliminando la columna duplicada.

Ejemplo 61-2. Una Inner Join (Una Join Interna)

Veamos las tablas que se han producido evaluando los pasos necesarios para una join. Sean las siguientes tablas dadas:

R	A	B	C	S	C	D	E
	1	2	3		3	a	b
	4	5	6		6	c	d
	7	8	9				

Primero calculamos el producto cartesiano $R \times S$ y tendremos:

R x S	A	B	R.C	S.C	D	E
	1	2	3	3	a	b
	1	2	3	6	c	d
	4	5	6	3	a	b

4	5	6	6	c	d
7	8	9	3	a	b
7	8	9	6	c	d

Tras la selección $\sigma_{R.C=S.C}(R \times S)$ tendremos:

A	B	R.C	S.C	D	E
1	2	3	3	a	b
4	5	6	6	c	d

Para eliminar las columnas duplicadas S.C realizamos la siguiente operación: $\pi_{R.A,R.B,R.C,S.D,S.E}(\sigma_{R.C=S.C}(R \times S))$ y obtenemos:

A	B	C	D	E
1	2	3	a	b
4	5	6	c	d

- **DIVIDE (\div):** Sea R una tabla con los atributos A, B, C , y D y sea S una tabla con los atributos C y D . Definimos la división como: $R \div S = \{t \mid \forall t_s \in S \exists t_r \in R \text{ tal que } t_r(A,B)=t \wedge t_r(C,D)=t_s\}$ donde $t_r(x,y)$ denota una tupla de la tabla R que consiste sólo en los componentes x y y . Nótese que la tupla t consiste sólo en los componentes A y B de la relación R .

Dadas las siguientes tablas

R	A	B	C	D	S	C	D
	a	b	c	d		c	d
	a	b	e	f		e	f
	b	c	e	f			
	e	d	c	d			
	e	d	e	f			
	a	b	d	e			

$R \div S$ se deriva como

A	B
a	b
e	d

Para una descripción y definición más detallada del Álgebra Relacional diríjanse a [Ullman, 1988] o [Date, 1994].

Ejemplo 61-3. Una consulta utilizando Álgebra Relacional

Recaltar que hemos formulado todos estos operadores relacionales como capaces de recuperar datos de la base de datos. Volvamos a nuestro ejemplo de la sección previa (*Operaciones en el Modelo de Datos Relacional*) donde alguien quería conocer los

nombres de todos los proveedores que venden el artículo Tornillos. Esta pregunta se responde utilizando el álgebra relacional con la siguiente operación:

$$\pi_{\text{SUPPLIER.SNAME}}(\sigma_{\text{PART.PNAME}='Tornillos'}(\text{SUPPLIER} \bowtie \text{SELLS} \bowtie \text{PART}))$$

Llamamos a estas operaciones una consulta. Si evaluamos la consulta anterior contra las tablas de nuestro ejemplo (*La Base de Datos de Proveedores y Artículos*) obtendremos el siguiente ejemplo:

SNAME
Smith
Adams

Cálculo Relacional

El Cálculo Relacional se basa en la *lógica de primer orden*. Hay dos variantes del cálculo relacional:

- El *Cálculo Relacional de Dominios* (DRC), donde las variables esperan componentes (atributos) de las tuplas.
- El *Cálculo Relacional de Tuplas* The *Tuple Relational Calculus* (TRC), donde las variables esperan tuplas.

Expondremos sólo el cálculo relacional de tuplas porque es el único utilizado por la mayoría de lenguajes relacionales. Para una discusión detallada de DRC (y también de TRC) vea [Date, 1994] o [Ullman, 1988].

Cálculo Relacional de Tuplas

Las consultas utilizadas en TRC tienen el siguiente formato: $x(A) \mid F(x)$ donde x es una variable de tipo tupla, A es un conjunto de atributos y F es una fórmula. La relación resultante consiste en todas las tuplas $t(A)$ que satisfagan $F(t)$.

Si queremos responder la pregunta del ejemplo *Una consulta utilizando Álgebra Relacional* utilizando TRC formularemos la siguiente consulta:

$$\{x(\text{SNAME}) \mid x \in \text{SUPPLIER} \wedge \neg \text{number} \\ \exists y \in \text{SELLS} \exists z \in \text{PART} (y(\text{SNO})=x(\text{SNO}) \wedge \neg \text{number} \\ z(\text{PNO})=y(\text{PNO}) \wedge \neg \text{number} \\ z(\text{PNAME})='Tornillos')\} \neg \text{number}$$

Evaluando la consulta contra las tablas de *La Base de Datos de Proveedores y Artículos* encontramos otra vez el mismo resultado de *Una consulta utilizando Álgebra Relacional*.

Álgebra Relacional contra Cálculo Relacional

El álgebra relacional y el cálculo relacional tienen el mismo *poder de expresión*; es decir, todas las consultas que se pueden formular utilizando álgebra relacional pueden también formularse utilizando el cálculo relacional, y viceversa. Esto fue probado por E. F. Codd en 1972. Este profesor se basó en un algoritmo ("algoritmo de reducción de Codd") mediante el cual una expresión arbitraria del cálculo relacional se puede reducir a la expresión semánticamente equivalente del álgebra relacional. Para una discusión más detallada sobre este punto, diríjase a [Date, 1994] y [Ullman, 1988].

Se dice a veces que los lenguajes basados en el cálculo relacional son de "más alto nivel" o "más declarativos" que los basados en el álgebra relacional porque el álgebra especifica (parcialmente) el orden de las operaciones, mientras el cálculo lo traslada a un compilador o interprete que determina el orden de evaluación más eficiente.

El Lenguaje SQL

Como en el caso de los más modernos lenguajes relacionales, SQL está basado en el cálculo relacional de tuplas. Como resultado, toda consulta formulada utilizando el cálculo relacional de tuplas (o su equivalente, el álgebra relacional) se puede formular también utilizando SQL. Hay, sin embargo, capacidades que van más allá del cálculo o del álgebra relacional. Aquí tenemos una lista de algunas características proporcionadas por SQL que no forman parte del álgebra y del cálculo relacionales:

- Comandos para inserción, borrado o modificación de datos.
- Capacidades aritméticas: En SQL es posible incluir operaciones aritméticas así como comparaciones, por ejemplo $A < B + 3$. Nótese que ni $+$ ni otros operadores aritméticos aparecían en el álgebra relacional ni en cálculo relacional.
- Asignación y comandos de impresión: es posible imprimir una relación construida por una consulta y asignar una relación calculada a un nombre de relación.
- Funciones agregadas: Operaciones tales como *promedio* (*average*), *suma* (*sum*), *máximo* (*max*), etc. se pueden aplicar a las columnas de una relación para obtener una cantidad única.

Select

El comando más usado en SQL es la instrucción SELECT, que se utiliza para recuperar datos. La sintaxis es:

```
SELECT [ALL|DISTINCT]
      { * | expr_1 [AS c_alias_1] [, ...
        [, expr_k [AS c_alias_k]]}]
FROM table_name_1 [t_alias_1]
   [, ... [, table_name_n [t_alias_n]]]
[WHERE condition]
[GROUP BY name_of_attr_i
   [, ... [, name_of_attr_j]] [HAVING condition]]
[{UNION [ALL] | INTERSECT | EXCEPT} SELECT ...]
[ORDER BY name_of_attr_i [ASC|DESC]
   [, ... [, name_of_attr_j [ASC|DESC]]]];
```

Ilustraremos ahora la compleja sintaxis de la instrucción SELECT con varios ejemplos. Las tablas utilizadas para los ejemplos se definen en: *La Base de Datos de Proveedores y Artículos*.

Select sencillas

Aquí tenemos algunos ejemplos sencillos utilizando la instrucción SELECT:

Ejemplo 61-4. Query sencilla con cualificación

Para recuperar todas las tuplas de la tabla PART donde el atributo PRICE es mayor que 10, formularemos la siguiente consulta:

```
SELECT * FROM PART
WHERE PRICE > 10;
```

y obtenemos la siguiente tabla:

PNO	PNAME	PRICE
3	Cerrojos	15
4	Levas	25

Utilizando "*" en la instrucción SELECT solicitaremos todos los atributos de la tabla. Si queremos recuperar sólo los atributos PNAME y PRICE de la tabla PART utilizaremos la instrucción:

```
SELECT PNAME, PRICE
FROM PART
WHERE PRICE > 10;
```

En este caso el resultado es:

PNAME	PRICE
Cerrojos	15
Levas	25

Nótese que la SELECT SQL corresponde a la "proyección" en álgebra relacional, no a la "selección" (vea *Álgebra Relacional* para más detalles).

Las cualificaciones en la clausula WHERE pueden también conectarse lógicamente utilizando las palabras claves OR, AND, y NOT:

```
SELECT PNAME, PRICE
FROM PART
WHERE PNAME = 'Cerrojos' AND
(PRICE = 0 OR PRICE < 15);
```

dará como resultado:

PNAME	PRICE
Cerrojos	15

Las operaciones aritméticas se pueden utilizar en la lista de objetivos y en la clausula WHERE. Por ejemplo, si queremos conocer cuanto cuestan si tomamos dos piezas de un artículo, podríamos utilizar la siguiente consulta:

```
SELECT PNAME, PRICE * 2 AS DOUBLE
FROM PART
WHERE PRICE * 2 < 50;
```

y obtenemos:

PNAME	DOUBLE
Tornillos	20
Tuercas	16
Cerrojos	30

Nótese que la palabra DOBLE tras la palabra clave AS es el nuevo título de la segunda columna. Esta técnica puede utilizarse para cada elemento de la lista objetivo para asignar un nuevo título a la columna resultante. Este nuevo título recibe el calificativo de "un alias". El alias no puede utilizarse en todo el resto de la consulta.

Joins (Cruces)

El siguiente ejemplo muestra como las *joins* (*cruces*) se realizan en SQL.

Para cruzar tres tablas SUPPLIER, PART y SELLS a través de sus atributos comunes, formularemos la siguiente instrucción:

```
SELECT S.SNAME, P.PNAME
FROM SUPPLIER S, PART P, SELLS SE
WHERE S.SNO = SE.SNO AND
      P.PNO = SE.PNO;
```

y obtendremos la siguiente tabla como resultado:

SNAME	PNAME
Smith	Tornillos
Smith	Tuercas
Jones	Levas
Adams	Tornillos
Adams	Cerrojos
Blake	Tuercas
Blake	Cerrojos
Blake	Levas

En la clausula FROM hemos introducido un alias al nombre para cada relación porque hay atributos con nombre común (SNO y PNO) en las relaciones. Ahora podemos distinguir entre los atributos con nombre común simplificando la adicción de un prefijo al nombre del atributo con el nombre del alias seguido de un punto. La join se calcula de la misma forma, tal como se muestra en *Una Inner Join (Una Join Interna)*. Primero el producto cartesiano: SUPPLIER \times PART \times SELLS Ahora seleccionamos únicamente aquellas tuplas que satisfagan las condiciones dadas en la clausula WHERE (es decir, los atributos con nombre común deben ser iguales). Finalmente eliminamos las columnas repetidas (S.SNAME, P.PNAME).

Operadores Agregados

SQL proporciona operadores agregados (como son AVG, COUNT, SUM, MIN, MAX) que toman el nombre de un atributo como argumento. El valor del operador agregado se calcula sobre todos los valores de la columna especificada en la tabla completa.

Si se especifican grupos en la consulta, el cálculo se hace sólo sobre los valores de cada grupo (vean la siguiente sección).

Ejemplo 61-5. Aggregates

Si queremos conocer el coste promedio de todos los artículos de la tabla PART, utilizaremos la siguiente consulta:

```
SELECT AVG(PRICE) AS AVG_PRICE
FROM PART;
```

El resultado es:

```
      AVG_PRICE
-----
        14.5
```

Si queremos conocer cuantos artículos se recogen en la tabla PART, utilizaremos la instrucción:

```
SELECT COUNT(PNO)
FROM PART;
```

y obtendremos:

```
      COUNT
-----
         4
```

Agregación por Grupos

SQL nos permite particionar las tuplas de una tabla en grupos. En estas condiciones, los operadores agregados descritos antes pueden aplicarse a los grupos (es decir, el valor del operador agregado no se calculan sobre todos los valores de la columna especificada, sino sobre todos los valores de un grupo. El operador agregado se calcula individualmente para cada grupo).

El particionamiento de las tuplas en grupos se hace utilizando las palabras clave **GROUP BY** seguidas de una lista de atributos que definen los grupos. Si tenemos **GROUP BY A₁, ..., A_k** habremos particionado la relación en grupos, de tal modo que dos tuplas son del mismo grupo si y sólo si tienen el mismo valor en sus atributos A₁, ..., A_k.

Ejemplo 61-6. Agregados

Si queremos conocer cuántos artículos han sido vendidos por cada proveedor formularemos la consulta:

```
SELECT S.SNO, S.SNAME, COUNT(SE.PNO)
FROM SUPPLIER S, SELLS SE
WHERE S.SNO = SE.SNO
GROUP BY S.SNO, S.SNAME;
```

y obtendremos:

SNO	SNAME	COUNT
1	Smith	2
2	Jones	1
3	Adams	2
4	Blake	3

Demos ahora una mirada a lo que está ocurriendo aquí. Primero, la join de las tablas SUPPLIER y SELLS:

S.SNO	S.SNAME	SE.PNO
1	Smith	1
1	Smith	2
2	Jones	4
3	Adams	1
3	Adams	3
4	Blake	2
4	Blake	3
4	Blake	4

Ahora particionamos las tuplas en grupos reuniendo todas las tuplas que tiene el mismo atributo en S.SNO y S.SNAME:

S.SNO	S.SNAME	SE.PNO
1	Smith	1
		2
2	Jones	4
3	Adams	1
		3
4	Blake	2
		3
		4

En nuestro ejemplo, obtenemos cuatro grupos y ahora podemos aplicar el operador agregado COUNT para cada grupo, obteniendo el resultado total de la consulta dada anteriormente.

Nótese que para el resultado de una consulta utilizando GROUP BY y operadores agregados para dar sentido a los atributos agrupados, debemos primero obtener la lista objetivo. Los demás atributos que no aparecen en la cláusula GROUP BY se seleccionarán utilizando una función agregada. Por otro lado, no se pueden utilizar funciones agregadas en atributos que aparecen en la cláusula GROUP BY.

Having

La cláusula HAVING trabaja de forma muy parecida a la cláusula WHERE, y se utiliza para considerar sólo aquellos grupos que satisfagan la cualificación dada en la misma. Las expresiones permitidas en la cláusula HAVING deben involucrar funciones agregadas. Cada expresión que utilice sólo atributos planos deberá recogerse en la cláusula WHERE. Por otro lado, toda expresión que involucre funciones agregadas debe aparecer en la cláusula HAVING.

Ejemplo 61-7. Having

Si queremos solamente los proveedores que venden más de un artículo, utilizaremos la consulta:

```
SELECT S.SNO, S.SNAME, COUNT(SE.PNO)
FROM SUPPLIER S, SELLS SE
WHERE S.SNO = SE.SNO
```

```
GROUP BY S.SNO, S.SNAME
HAVING COUNT(SE.PNO) > 1;
```

y obtendremos:

SNO	SNAME	COUNT
1	Smith	2
3	Adams	2
4	Blake	3

Subconsultas

En las cláusulas WHERE y HAVING se permite el uso de subconsultas (subselects) en cualquier lugar donde se espere un valor. En este caso, el valor debe derivar de la evaluación previa de la subconsulta. El uso de subconsultas amplía el poder expresivo de SQL.

Ejemplo 61-8. Subselect

Si queremos conocer los artículos que tienen mayor precio que el artículo llamado 'Tornillos', utilizaremos la consulta:

```
SELECT *
FROM PART
WHERE PRICE > (SELECT PRICE FROM PART
               WHERE PNAME='Tornillos');
```

El resultado será:

PNO	PNAME	PRICE
3	Cerrojos	15
4	Levas	25

Cuando revisamos la consulta anterior, podemos ver la palabra clave SELECT dos veces. La primera al principio de la consulta - a la que nos referiremos como la SELECT externa - y la segunda en la cláusula WHERE, donde empieza una consulta anidada - nos referiremos a ella como la SELECT interna. Para cada tupla de la SELECT externa, la SELECT interna deberá ser evaluada. Tras cada evaluación, conoceremos el precio de la tupla llamada 'Tornillos', y podremos chequear si el precio de la tupla actual es mayor.

Si queremos conocer todos los proveedores que no venden ningún artículo (por ejemplo, para poderlos eliminar de la base de datos), utilizaremos:

```
SELECT *
FROM SUPPLIER S
WHERE NOT EXISTS
  (SELECT * FROM SELLS SE
   WHERE SE.SNO = S.SNO);
```

En nuestro ejemplo, obtendremos un resultado vacío, porque cada proveedor vende al menos un artículo. Nótese que utilizamos S.SNO de la SELECT externa en la cláusula WHERE de la SELECT interna. Como hemos descrito antes, la subconsulta se evalúa para cada tupla de la consulta externa, es decir, el valor de S.SNO se toma siempre de la tupla actual de la SELECT externa.

Unión, Intersección, Excepción

Estas operaciones calculan la unión, la intersección y la diferencia de la teoría de conjuntos de las tuplas derivadas de dos subconsultas.

Ejemplo 61-9. Union, Intersect, Except

La siguiente consulta es un ejemplo de UNION:

```
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNAME = 'Jones'
UNION
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNAME = 'Adams';
```

Dará el resultado:

SNO	SNAME	CITY
2	Jones	Paris
3	Adams	Vienna

Aquí tenemos un ejemplo para INTERSECT:

```
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNO > 1
INTERSECT
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNO > 2;
```

que dará como resultado:

SNO	SNAME	CITY
2	Jones	Paris

La única tupla devuelta por ambas partes de la consulta es la única que tiene \$SNO=2\$.

Finalmente, un ejemplo de EXCEPT:

```
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNO > 1
EXCEPT
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNO > 3;
```

que dará como resultado:

SNO	SNAME	CITY
2	Jones	Paris
3	Adams	Vienna

Definición de Datos

El lenguaje SQL incluye un conjunto de comandos para definición de datos.

Create Table

El comando fundamental para definir datos es el que crea una nueva relación (una nueva tabla). La sintaxis del comando CREATE TABLE es:

```
CREATE TABLE table_name
    (name_of_attr_1 type_of_attr_1
    [, name_of_attr_2 type_of_attr_2
    [, ...]]);
```

Ejemplo 61-10. Creación de una tabla

Para crear las tablas definidas en *La Base de Datos de Proveedores y Artículos* se utilizaron las siguientes instrucciones de SQL:

```
CREATE TABLE SUPPLIER
    (SNO    INTEGER,
     SNAME  VARCHAR(20),
     CITY   VARCHAR(20));

CREATE TABLE PART
    (PNO    INTEGER,
     PNAME  VARCHAR(20),
     PRICE  DECIMAL(4 , 2));

CREATE TABLE SELLS
    (SNO    INTEGER,
     PNO    INTEGER);
```

Tipos de Datos en SQL

A continuación sigue una lista de algunos tipos de datos soportados por SQL:

- INTEGER: entero binario con signo de palabra completa (31 bits de precisión).
- SMALLINT: entero binario con signo de media palabra (15 bits de precisión).
- DECIMAL(p,q): número decimal con signo de p dígitos de precisión, asumiendo q a la derecha para el punto decimal. ($15 \geq p \geq q \geq 0$). Si q se omite, se asume que vale 0.
- FLOAT: numérico con signo de doble palabra y coma flotante.
- CHAR(n): cadena de caracteres de longitud fija, de longitud n .
- VARCHAR(n): cadena de caracteres de longitud variable, de longitud máxima n .

Create Index

Se utilizan los índices para acelerar el acceso a una relación. Si una relación R tiene un índice en el atributo A podremos recuperar todas la tuplas t que tienen $t(A) = a$ en un tiempo aproximadamente proporcional al número de tales tuplas t más que en un tiempo proporcional al tamaño de R .

Para crear un índice en SQL se utiliza el comando CREATE INDEX. La sintaxis es:

```
CREATE INDEX index_name
ON table_name ( name_of_attribute );
```

Ejemplo 61-11. Create Index

Para crear un índice llamado I sobre el atributo SNAME de la relación SUPPLIER, utilizaremos la siguiente instrucción:

```
CREATE INDEX I
ON SUPPLIER ( SNAME );
```

El índice creado se mantiene automáticamente. es decir, cada vez que una nueva tupla se inserte en la relación SUPPLIER, se adaptará el índice I. Nótese que el único cambio que un usuario puede percibir cuando se crea un índice es un incremento en la velocidad.

Create View

Se puede ver una vista como una *tabla virtual*, es decir, una tabla que no existe físicamente en la base de datos, pero aparece al usuario como si existiese. Por contra, cuando hablamos de una *tabla base*, hay realmente un equivalente almacenado para cada fila en la tabla en algún sitio del almacenamiento físico.

Las vistas no tienen datos almacenados propios, distinguibles y físicamente almacenados. En su lugar, el sistema almacena la definición de la vista (es decir, las reglas para acceder a las tablas base físicamente almacenadas para materializar la vista) en algún lugar de los catálogos del sistema (vea *System Catalogs*). Para una discusión de las diferentes técnicas para implementar vistas, refiérase a *SIM98*.

En SQL se utiliza el comando **CREATE VIEW** para definir una vista. La sintaxis es:

```
CREATE VIEW view_name
AS select_stmt
```

donde *select_stmt* es una instrucción select válida, como se definió en *Select*. Nótese que *select_stmt* no se ejecuta cuando se crea la vista. Simplemente se almacena en los *catálogos del sistema* y se ejecuta cada vez que se realiza una consulta contra la vista.

Sea la siguiente definición de una vista (utilizamos de nuevo las tablas de *La Base de Datos de Proveedores y Artículos*):

```
CREATE VIEW London_Suppliers
```

```

AS SELECT S.SNAME, P.PNAME
FROM SUPPLIER S, PART P, SELLS SE
WHERE S.SNO = SE.SNO AND
      P.PNO = SE.PNO AND
      S.CITY = 'London';

```

Ahora podemos utilizar esta *relación virtual* `London_Suppliers` como si se tratase de otra tabla base:

```

SELECT *
FROM London_Suppliers
WHERE P.PNAME = 'Tornillos';

```

Lo cual nos devolverá la siguiente tabla:

SNAME	PNAME
Smith	Tornillos

Para calcular este resultado, el sistema de base de datos ha realizado previamente un acceso *oculto* a las tablas de la base `SUPPLIER`, `SELLS` y `PART`. Hace esto ejecutando la consulta dada en la definición de la vista contra aquellas tablas base. Tras eso, las cualificaciones adicionales (dadas en la consulta contra la vista) se podrán aplicar para obtener la tabla resultante.

Drop Table, Drop Index, Drop View

Se utiliza el comando `DROP TABLE` para eliminar una tabla (incluyendo todas las tuplas almacenadas en ella):

```
DROP TABLE table_name;
```

Para eliminar la tabla `SUPPLIER`, utilizaremos la instrucción:

```
DROP TABLE SUPPLIER;
```

Se utiliza el comando `DROP INDEX` para eliminar un índice:

```
DROP INDEX index_name;
```

Finalmente, eliminaremos una vista dada utilizando el comando `DROP VIEW`:

```
DROP VIEW view_name;
```

Manipulación de Datos

Insert Into

Una vez que se crea una tabla (vea *Create Table*), puede ser llenada con tuplas mediante el comando **INSERT INTO**. La sintaxis es:

```
INSERT INTO table_name (name_of_attr_1
                        [, name_of_attr_2 [, ...]])
VALUES (val_attr_1
       [, val_attr_2 [, ...]]);
```

Para insertar la primera tupla en la relación SUPPLIER (de *La Base de Datos de Proveedores y Artículos*) utilizamos la siguiente instrucción:

```
INSERT INTO SUPPLIER (SNO, SNAME, CITY)
VALUES (1, 'Smith', 'London');
```

Para insertar la primera tupla en la relación SELLS, utilizamos:

```
INSERT INTO SELLS (SNO, PNO)
VALUES (1, 1);
```

Update

Para cambiar uno o más valores de atributos de tuplas en una relación, se utiliza el comando UPDATE. La sintaxis es:

```
UPDATE table_name
SET name_of_attr_1 = value_1
  [, ... [, name_of_attr_k = value_k]]
WHERE condition;
```

Para cambiar el valor del atributo PRICE en el artículo 'Tornillos' de la relación PART, utilizamos:

```
UPDATE PART
SET PRICE = 15
WHERE PNAME = 'Tornillos';
```

El nuevo valor del atributo PRICE de la tupla cuyo nombre es 'Tornillos' es ahora 15.

Delete

Para borrar una tupla de una tabla particular, utilizamos el comando DELETE FROM. La sintaxis es:

```
DELETE FROM table_name
WHERE condition;
```

Para borrar el proveedor llamado 'Smith' de la tabla SUPPLIER, utilizamos la siguiente instrucción:

```
DELETE FROM SUPPLIER
WHERE SNAME = 'Smith';
```

System Catalogs

En todo sistema de base de datos SQL se emplean *catálogos de sistema* para mantener el control de qué tablas, vistas, índices, etc están definidas en la base de datos. Estos catálogos del sistema se pueden investigar como si de cualquier otra relación normal se tratase. Por ejemplo, hay un catálogo utilizado para la definición de vistas. Este catálogo almacena la consulta de la definición de la vista. Siempre que se hace una consulta contra la vista, el sistema toma primero la *consulta de definición de la vista* del catálogo y materializa la vista antes de proceder con la consulta del usuario (vea SIM98 para obtener una descripción más detallada). Diríjase a DATE para obtener más información sobre los catálogos del sistema.

SQL Embebido

En esta sección revisaremos como se puede embeber SQL en un lenguaje de host (p.e. C). Hay dos razones principales por las que podríamos querer utilizar SQL desde un lenguaje de host:

- Hay consultas que no se pueden formular utilizando SQL puro (por ejemplo, las consultas recursivas). Para ser capaz de realizar esas consultas necesitamos un lenguaje de host de mayor poder expresivo que SQL.
- Simplemente queremos acceder a una base de datos desde una aplicación que está escrita en el lenguaje del host (p.e. un sistema de reserva de billetes con una interface gráfica escrita en C, y la información sobre los billetes está almacenada en una base de datos que puede accederse utilizando SQL embebido).

Un programa que utiliza SQL embebido en un lenguaje de host consiste en instrucciones del lenguaje del host e instrucciones de *SQL embebido* (ESQL). Cada instrucción de ESQL empieza con las palabras claves **EXEC SQL**. Las instrucciones ESQL se transforman en instrucciones del lenguaje del host mediante un *precompilador* (que habitualmente inserta llamadas a rutinas de librerías que ejecutan los variados comandos de SQL).

Cuando vemos los ejemplos de *Select* observamos que el resultado de las consultas es algo muy próximo a un conjunto de tuplas. La mayoría de los lenguajes de host no están diseñados para operar con conjuntos, de modo que necesitamos un mecanismo para acceder a cada tupla única del conjunto de tuplas devueltas por una instrucción SELECT. Este mecanismo puede ser proporcionado declarando un *cursor*. Tras ello, podemos utilizar el comando FETCH para recuperar una tupla y apuntar el cursor hacia la siguiente tupla.

Para una discusión más detallada sobre el SQL embebido, diríjase a [*Date and Darwen, 1997*], [*Date, 1994*], o [*Ullman, 1988*].

Capítulo 62. Arquitectura

Postgres Conceptos de arquitectura

Antes de comenzar, debería comprender las bases de la arquitectura del sistema Postgres. Entendiendo como las partes de Postgres interactúan le hará el siguiente capítulo mucho más sencillo. En la jerga de bases de datos, Postgres usa un modelo cliente/sevidor conocido como "proceso por usuario". Una sesión Postgres consiste en los siguientes procesos cooperativos de Unix (programas):

- Un proceso demonio supervisor (`postmaster`),
- la aplicación sobre la que trabaja el usuario (frontend) (e.g., el `programapsql`), y
- uno o más servidores de bases dedatos en segundo plano (el mismo proceso `postgres`).

Un único `postmaster` controla una colección de bases de datos dadas en un único host. Debido a esto una colección de bases de datos se suele llamar una instalación o un sitio. Las aplicaciones de frontend que quieren acceder a una determinada base de datos dentro de una instalación hacen llamadas a la librería La librería envía peticiones de usuario a través de la red al `postmaster` (*Como se establece una conexión*), el cual en respuesta inicia un nuevo proceso en el servidor (backend)

Figura 62-1. Como se establece una conexión

y conecta el proceso de frontend al nuevo servidor. A partir de este punto, el proceso de frontend y el servidor en backend se comunican sin la intervención del `postmaster`. Aunque, el `postmaster` siempre se está ejecutando, esperando peticiones, tanto los procesos de frontend como los de backend vienen y se van.

La librería `libpq` permite a un único proceso en frontend realizar múltiples conexiones a procesos en backend. Aunque, la aplicación frontend todavía es un proceso en un único thread. Conexiones multithread entre el frontend y el backend no están soportadas de momento en `libpq`. Una implicación de esta arquitectura es que el `postmaster` y el proceso backend siempre se ejecutan en la misma máquina (el servidor de base de datos), mientras que la aplicación en frontend puede ejecutarse desde cualquier sitio. Debe tener esto en mente, porque los archivos que pueden ser accedidos en la máquina del cliente pueden no ser accesibles (o sólo pueden ser accedidos usando un nombre de archivo diferente) el máquina del servidor de base de datos.

Tenga en cuenta que los servicios `postmaster` y `postgres` se ejecutan con el identificador de usuario del "superusuario" Postgres. Note que el superusuario Postgres no necesita ser un usuario especial (ej. un usuario llamado "postgres"). De todas formas, el superusuario Postgres definitivamente no tiene que ser el superusuario de Unix

("root")! En cualquier caso, todos los archivos relacionados con la base de datos deben pertenecer a este superusuario Postgres. Postgres.

Capítulo 63. Empezando

¿Cómo empezar a trabajar con Postgres?

Algunos de los pasos necesarios para usar Postgres pueden ser realizados por cualquier usuario, y algunos los deberá realizar el administrador de la base de datos. Este administrador es la persona que instaló el software, creó los directorios de las bases de datos e inició el proceso `postmaster`. Esta persona no tiene que ser el superusuario Unix (“root”) o el administrador del sistema. Una persona puede instalar y usar Postgres sin tener una cuenta especial o privilegiada

Si está instalando Postgres, consulte las instrucciones de instalación en la Guía de Administración y regrese a esta guía cuando haya concluido la instalación.

Mientras lee este manual, cualquier ejemplo que vea que comience con el carácter “%” son órdenes que se escribirán en la línea de órdenes de Unix. Los ejemplos que comienzan con el carácter “*” son órdenes en el lenguaje de consulta Postgres, Postgres SQL.

Configurando el entorno

Esta sección expone la manera de configurar el entorno, para las aplicaciones. Asumimos que Postgres ha sido instalado e iniciado correctamente; consulte la Guía del Administrador y las notas de instalación si desea instalar Postgres.

Postgres es una aplicación cliente/servidor. Como usuario, únicamente necesita acceso a la parte cliente (un ejemplo de una aplicación cliente es el monitor interactivo `psql`) Por simplicidad, asumiremos que Postgres ha sido instalado en el directorio `/usr/local/pgsql`. Por lo tanto, donde vea el directorio `/usr/local/pgsql`, deberá sustituirlo por el nombre del directorio donde Postgres esté instalado realmente. Todos los programas de Postgres se instalan (en este caso) en el directorio `/usr/local/pgsql/bin`. Por lo tanto, deberá añadir este directorio a la de su shell ruta de órdenes. Si usa una variante del C shell de Berkeley, tal como `tcsh` o `csh`, deberá añadir

```
% set path = ( /usr/local/pgsql/bin path )
```

en el archivo `.login` de su directorio personal. Si usa una variante del Bourne shell, tal como `sh`, `ksh` o `bash` entonces deberá añadir

```
% PATH=/usr/local/pgsql/bin:$PATH
% export PATH
```

en el archivo `.profile` de su directorio personal. Desde ahora, asumiremos que ha añadido el directorio `bin` de Postgres a su `path`. Además, haremos referencia frecuentemente a “configurar una variable de shell” o “configurar una variable de entorno” a lo largo de este documento. Si no entiende completamente el último párrafo al respecto de la modificación de su `path`, antes de continuar debería consultar los manuales de Unix que describen el shell que utiliza.

Si el administrador del sistema no tiene la configuración en el modo por defecto, tendrá que realizar trabajo extra. Por ejemplo, si la máquina servidor de bases de datos es una máquina remota, necesitará configurar la variable de entorno `PGHOST` con el nombre de la máquina servidor de bases de datos. También deberá especificar la variable de entorno `PGPORT`. Si trata de iniciar un programa de aplicación y éste

notifica que no puede conectarse al `postmaster`, deberá consultar al administrador para asegurarse de que su entorno está configurado adecuadamente.

Ejecución del Monitor Interactivo (`psql`)

Asumiendo que su administrador haya ejecutado adecuadamente el proceso `postmaster` y le haya autorizado a utilizar la base de datos, puede comenzar a ejecutar aplicaciones como usuario. Como mencionamos previamente, debería añadir `/usr/local/pgsql/bin` al “path” de búsqueda de su intérprete de órdenes. En la mayoría de los casos, es lo único que tendrá que hacer en términos de preparación.

Desde Postgres v6.3, se soportan dos tipos diferentes de conexión. El administrador puede haber elegido permitir conexiones por red TCP/IP, o restringir los accesos a la base de datos a través de conexiones locales (en la misma máquina). Esta elección puede ser significativa si encuentra problemas a la hora de conectar a la base de datos.

Si obtiene los siguientes mensajes de error de una orden Postgres (tal como `psql` o `createdb`):

```
% psql template1
Connection to database 'postgres' failed.
connectDB() failed: Is the postmaster running and accepting connections
    at 'UNIX Socket' on port '5432'?
```

O

```
% psql -h localhost template1
Connection to database 'postgres' failed.
connectDB() failed: Is the postmaster running and accepting TCP/IP
    (with -i) connections at 'localhost' on port '5432'?
```

normalmente es debido a que (1) el `postmaster` no está en funcionamiento, o (2) está intentando conectar al servidor equivocado. Si obtiene el siguiente mensaje de error:

```
FATAL 1:Feb 17 23:19:55:process userid (2360) != database owner (268)
```

Significa que el administrador ejecutó el `postmaster` mediante el usuario equivocado. Dígame que lo reinicie utilizando el superusuario de Postgres.

Administrando una Base de datos

Ahora que Postgres está ejecutándose podemos crear alguna base de datos para experimentar con ella. Aquí describimos las órdenes básicas para administrar una base de datos

La mayoría de las aplicaciones Postgres asumen que el nombre de la base de datos, si no se especifica, es el mismo que el de su cuenta en el sistema.

Si el administrador de bases de datos ha configurado su cuenta sin privilegios de creación de bases de datos, entonces deberán decirle el nombre de sus bases de datos. Si este es el caso, entonces puede omitir la lectura de esta sección sobre creación y destrucción de bases de datos.

Creación de una base de datos

Digamos que quiere crear una base de datos llamada mydb. Puede hacerlo con la siguiente orden:

```
% createdb mydb
```

Si no cuenta con los privilegios requeridos para crear bases de datos, verá lo siguiente:

```
% createdb mydb
NOTICE:user "su nombre de usuario" is not allowed to create/destroy databases
createdb: database creation failed on mydb.
```

Postgres le permite crear cualquier número de bases de datos en un sistema dado y automáticamente será el administrador de la base de datos que creó. Los nombres de las bases de datos deben comenzar por un carácter alfabético y están limitados a una longitud de 32 caracteres. No todos los usuarios están autorizados para ser administrador de una base de datos. Si Postgres le niega la creación de bases de datos, seguramente es debido a que el administrador del sistema ha de otorgarle permisos para hacerlo. En ese caso, consulte al administrador del sistema.

Acceder a una base de datos

Una vez que ha construido una base de datos, puede acceder a ella:

- Ejecutando los programas de monitorización de Postgres (por ejemplo `psql`) los cuales le permiten introducir, editar y ejecutar órdenes SQL interactivamente
- Escribiendo un programa en C usando la librería de subrutinas LIBPQ, la cual le permite enviar órdenes SQL desde C y obtener mensajes de respuesta en su programa. Esta interfaz es discutida más adelante en la *Guía de Programadores de PostgreSQL*

Puede que desee ejecutar `psql`, para probar los ejemplos en este manual. Lo puede activar para la base de datos mydb escribiendo la orden:

```
% psql mydb
```

Se le dará la bienvenida con el siguiente mensaje:

```
Welcome to the POSTGRESQL interactive sql monitor:
Please read the file COPYRIGHT for copyright terms of POSTGRESQL

type \? for help on slash commands
type \q to quit
type \g or terminate with semicolon to execute query
You are currently connected to the database: templatel

mydb=>
```

Este prompt indica que el monitor está listo y puede escribir sus consultas SQL dentro de un espacio de trabajo mantenido por el monitor. El programa `psql` responde a los

códigos de escape que empiezan por el carácter “\”. Por ejemplo, puede obtener la ayuda acerca de la sintaxis de varios órdenes SQL Postgres escribiendo:

```
mydb=> \h
```

Una vez que haya terminado de introducir consultas, puede pasar el contenido del espacio de trabajo al servidor Postgres escribiendo:

```
mydb=> \g
```

Esto le dice al servidor que procese la consulta. Si termina su consulta con un punto y coma, la “\g” no es necesaria. `psql` procesará automáticamente las consultas terminadas con punto y coma. Para leer consultas desde un archivo, digamos `myFile`, en lugar de introducirlas interactivamente, escriba:

```
mydb=> \i nombreDelFichero
```

Para salir de `psql` y regresar a Unix escriba:

```
mydb=> \q
```

y `psql` terminará y volverá a la línea de órdenes. (Para conocer más códigos de escape, escriba `\h` en el prompt del monitor). Se pueden utilizar espacios en blanco (por ejemplo espacios, tabulador y el carácter de nueva línea) en las consultas SQL. Las líneas simples comentadas comienzan por “-”. Lo que haya después de los guiones hasta el final de línea será ignorado. Los comentarios múltiples y los que ocupan más de una línea se señalan con “/* ... */”

Eliminando bases de datos

Si es el administrador de la base de datos `mydb`, puede eliminarla utilizando la siguiente orden Unix:

```
% dropdb mydb
```

Esta acción elimina físicamente todos los archivos Unix asociados a la base de datos y no pueden recuperarse, así que deberá hacerse con precaución.

Capítulo 64. El Lenguaje de consultas

El lenguaje de consultas de Postgres es una variante del estándar SQL3. Tiene muchas extensiones, tales como tipos de sistema extensibles, herencia, reglas de producción y funciones. Estas son características tomadas del lenguaje de consultas original de Postgres (PostQuel). Esta sección proporciona un primer vistazo de cómo usar Postgres SQL para realizar operaciones sencillas. La intención de este manual es simplemente la de proporcionarle una idea de nuestra versión de SQL y no es de ningún modo un completo tutorial acerca de SQL. Se han escrito numerosos libros sobre SQL, incluyendo [MELT93] and [DATE97]. Tenga en cuenta que algunas características del lenguaje son extensiones del estándar ANSI.

Monitor interactivo

En los ejemplos que siguen, asumimos que ha creado la base de datos `mydb` como se describe en la subsección anterior y que ha arrancado `psql`. Los ejemplos que aparecen en este manual también se pueden encontrar en `/usr/local/pgsql/src/tutorial/`. Consulte el fichero `README` en ese directorio para saber cómo usarlos. Para empezar con el tutorial haga lo siguiente:

```
% cd /usr/local/pgsql/src/tutorial
% psql -s mydb
Welcome to the POSTGRES interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRES

  type \? for help on slash commands
  type \q to quit
  type \g or terminate with semicolon to execute query
  You are currently connected to the database: postgres

mydb=> \i basics.sql
```

El comando `\i` lee en las consultas desde los ficheros especificados. La opción `-s` le pone en modo single step, que hace una pausa antes de enviar la consulta al servidor. Las consultas de esta sección están en el fichero `basics.sql`.

`psql` tiene varios comandos `\d` para mostrar información de sistema. Consulte éstos comandos para ver más detalles y teclee `\?` desde el prompt `psql` para ver un listado de comandos disponibles.

Conceptos

La noción fundamental en Postgres es la de clase, que es una colección de instancias de un objeto. Cada instancia tiene la misma colección de atributos y cada atributo es de un tipo específico. Más aún, cada instancia tiene un *identificador de objeto* (OID) permanente, que es único a lo largo de toda la instalación. Ya que la sintaxis SQL hace referencia a tablas, usaremos los términos *tabla* y *clase* indistintamente. Asimismo, una *fila* SQL es una *instancia* y las *columnas* SQL son *atributos*. Como ya se dijo anteriormente, las clases se agrupan en bases de datos y una colección de bases de datos gestionada por un único proceso `postmaster` constituye una instalación o sitio.

Creación de una nueva clase

Puede crear una nueva clase especificando el nombre de la clase , además de todos los nombres de atributo y sus tipos:

```
CREATE TABLE weather (
    city          varchar(80),
    temp_lo       int,           - temperatura mínima
    temp_hi       int,           - temperatura máxima
    prcp          real,         - precipitación
    date          date
);
```

Tenga en cuenta que las palabras clave y los identificadores son sensibles a las mayúsculas y minúsculas. Los identificadores pueden llegar a ser sensibles a mayúsculas o minúsculas si se les pone entre dobles comillas, tal como lo permite SQL92. Postgres SQL soporta los tipos habituales de SQL como: int, float, real, smallint, char(N), varchar(N), date, time, y timestamp, así como otros de tipo general y otros con un rico conjunto de tipos geométricos. Tal como veremos más tarde, Postgres puede ser configurado con un número arbitrario de tipos de datos definidos por el usuario. Consecuentemente, los nombres de tipo no son sintácticamente palabras clave, excepto donde se requiera para soportar casos especiales en el estándar SQL92 . Yendo más lejos, el comando Postgres **CREATE** es idéntico al comando usado para crear una tabla en el sistema relacional de siempre . Sin embargo, veremos que las clases tienen propiedades que son extensiones del modelo relacional.

Llenando una clase con instancias

La declaración **insert** se usa para llenar una clase con instancias:

```
INSERT INTO weather
VALUES ('San Francisco', 46, 50, 0.25, '11/27/1994');
```

También puede usar el comando **copy** para cargar grandes cantidades de datos desde ficheros (ASCII) . Generalmente esto suele ser más rápido porque los datos son leídos (o escritos) como una única transacción directamente a o desde la tabla destino. Un ejemplo sería:

```
COPY weather FROM '/home/user/weather.txt'
USING DELIMITERS '|' ;
```

donde el path del fichero origen debe ser accesible al servidor backend , no al cliente, ya que el servidor lee el fichero directamente

Consutar a una clase

La clase weather puede ser consultada con una selección relacional normal y consultas de proyección. La declaración SQL **select** se usa para hacer esto. La declaración se divide en una lista destino (la parte que lista los atributos que han de ser devueltos)

y una cualificación (la parte que especifica cualquier restricción). Por ejemplo, para recuperar todas las filas de `weather`, escriba:

```
SELECT * FROM weather;
```

and the output should be:

```
+-----+-----+-----+-----+
|city      | temp_lo | temp_hi | prcp | date      |
+-----+-----+-----+-----+
|San Francisco | 46      | 50      | 0.25 | 11-27-1994 |
+-----+-----+-----+-----+
|San Francisco | 43      | 57      | 0     | 11-29-1994 |
+-----+-----+-----+-----+
|Hayward      | 37      | 54      |      | 11-29-1994 |
+-----+-----+-----+-----+
```

Puede especificar cualquier expresión en la lista de destino. Por ejemplo, puede hacer:

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

Los operadores booleanos (**and**, **or** and **not**) se pueden usar en la cualificación de cualquier consulta. Por ejemplo,

```
SELECT * FROM weather
    WHERE city = 'San Francisco'
    AND prcp > 0.0;
```

da como resultado:

```
+-----+-----+-----+-----+
|city      | temp_lo | temp_hi | prcp | date      |
+-----+-----+-----+-----+
|San Francisco | 46      | 50      | 0.25 | 11-27-1994 |
+-----+-----+-----+-----+
```

Como apunte final, puede especificar que los resultados de un select puedan ser devueltos de *manera ordenada* o quitando las *instancias duplicadas*.

```
SELECT DISTINCT city
    FROM weather
    ORDER BY city;
```

Redireccionamiento de consultas SELECT

Cualquier consulta select puede ser redireccionada a una nueva clase:

```
SELECT * INTO TABLE temp FROM weather;
```

Esto forma de manera implícita un comando **create**, creándose una nueva clase temp con el atributo names y types especificados en la lista destino del comando **select into**. Entonces podremos, por supuesto, realizar cualquier operación sobre la clase resultante como lo haríamos sobre cualquier otra clase.

Joins (uniones) entre clases

Hasta ahora, nuestras consultas sólo accedían a una clase a la vez. Las consultas pueden acceder a múltiples clases a la vez, o acceder a la misma clase de tal modo que múltiples instancias de la clase sean procesadas al mismo tiempo. Una consulta que acceda a múltiples instancias de las mismas o diferentes clases a la vez se conoce como una consulta join. Como ejemplo, digamos que queremos encontrar todos los registros que están en el rango de temperaturas de otros registros. En efecto, necesitamos comparar los atributos temp_lo y temp_hi de cada instancia EMP con los atributos temp_lo y temp_hi de todas las demás instancias EMP.

Nota: Esto es sólo un modelo conceptual. El verdadero join puede hacerse de una manera más eficaz, pero esto es invisible para el usuario.

Podemos hacer esto con la siguiente consulta:

```
SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
       W2.city, W2.temp_lo AS low, W2.temp_hi AS high
FROM   weather W1, weather W2
WHERE  W1.temp_lo < W2.temp_lo
AND    W1.temp_hi > W2.temp_hi;
```

```
+-----+-----+-----+-----+-----+
|city      | low | high | city      | low | high |
+-----+-----+-----+-----+-----+
|San Francisco | 43  | 57   | San Francisco | 46  | 50   |
+-----+-----+-----+-----+-----+
|San Francisco | 37  | 54   | San Francisco | 46  | 50   |
+-----+-----+-----+-----+-----+
```

Nota : Los matices de este join están en que la cualificación es una expresión verdadera definida por el producto cartesiano de las clases indicadas en la consulta. Para estas instancias en el producto cartesiano cuya cualificación sea verdadera, Postgres calcula y devuelve los valores especificados en la lista de destino. Postgres SQL no da ningún significado a los valores duplicados en este tipo de expresiones. Esto significa que Postgres en ocasiones recalcula la misma lista de destino varias veces. Esto ocurre frecuentemente cuando las expresiones booleanas se conectan con un "or". Para eliminar estos duplicados, debe usar la declaración **select distinct**.

En este caso, tanto W1 como W2 son sustituidos por una instancia de la clase weather y se extienden por todas las instancias de la clase. (En la terminología de la mayoría

de los sistemas de bases de datos W1 y W2 se conocen como *range variables* (*variables de rango*.) Una consulta puede contener un número arbitrario de nombres de clases y sustituciones.

Actualizaciones

Puede actualizar instancias existentes usando el comando `update`. Suponga que descubre que la lectura de las temperaturas el 28 de Noviembre fue 2 grados superior a la temperatura real. Puede actualizar los datos de esta manera:

```
UPDATE weather
  SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
  WHERE date > '11/28/1994';
```

Borrados

Los borrados se hacen usando el comando **delete**:

```
DELETE FROM weather WHERE city = 'Hayward';
```

Todos los registros de `weather` pertenecientes a Hayward son borrados. Debería ser precavido con las consultas de la forma

```
DELETE FROM classname;
```

Sin una cualificación, **delete** simplemente borrará todas las instancias de la clase dada, dejándola vacía. El sistema no pedirá confirmación antes de hacer esto.

Uso de funciones de conjunto

Como otros lenguajes de consulta, PostgreSQL soporta funciones de conjunto. Una función de conjunto calcula un único resultado a partir de múltiples filas de entrada. Por ejemplo, existen funciones globales para calcular `count` (contar), `sum` (sumar), `avg` (media), `max` (máximo) and `min` (mínimo) sobre un conjunto de instancias.

Es importante comprender la relación entre las funciones de conjunto y las cláusulas SQL **where** y **having**. La diferencia fundamental entre **where** y **having** es que: **where** selecciona las columnas de entrada antes de los grupos y entonces se computan las funciones de conjunto (de este modo controla qué filas van a la función de conjunto), mientras que **having** selecciona grupos de filas después de los grupos y entonces se computan las funciones de conjunto. De este modo la cláusula **where** puede no contener funciones de conjunto puesto que no tiene sentido intentar usar una función de conjunto para determinar qué fila será la entrada de la función. Por otra parte, las cláusulas **having** siempre contienen funciones de conjunto. (Estrictamente hablando, usted puede escribir una cláusula **having** que no use funciones de grupo, pero no merece la pena. La misma condición podría ser usada de un modo más eficaz con **where**.)

Como ejemplo podemos buscar la mínima temperatura en cualquier parte con

```
SELECT max(temp_lo) FROM weather;
```

Si queremos saber qué ciudad o ciudades donde se dieron estas temperaturas, podemos probar

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo);
```

pero no funcionará debido a que la función `max()` no puede ser usada en **where**. Sin embargo, podemos replantar la consulta para llevar a cabo lo que buscamos. En este caso usando una *subselección*:

```
SELECT city FROM weather WHERE temp_lo = (SELECT max(temp_lo) FROM weather);
```

Lo que ya es correcto, ya que la subselección es una operación independiente que calcula su propia función de grupo sin importar lo que pase en el select exterior.

Las funciones de grupo son también muy útiles combinándolas con cláusulas *group by*. Por ejemplo, podemos obtener la temperatura mínima tomada en cada ciudad con:

```
SELECT city, max(temp_lo)
FROM weather
GROUP BY city;
```

que nos devuelve una fila por ciudad. Podemos filtrar estas filas agrupadas usando **having**:

```
SELECT city, max(temp_lo)
FROM weather
GROUP BY city
HAVING min(temp_lo) < 0;
```

que nos da los mismos resultados, pero de ciudades con temperaturas bajo cero. Finalmente, si sólo nos interesan las ciudades cuyos nombres empiecen por 'P', deberíamos hacer:

```
SELECT city, max(temp_lo)
FROM weather
WHERE city like 'P%'
GROUP BY city
HAVING min(temp_lo) < 0;
```

Tenga en cuenta que podemos aplicar la restricción del nombre de ciudad en **where**, ya que no necesita funciones de conjunto. Esto es más eficaz que añadir la restricción a **having**, debido a que evitamos hacer los cálculos de grupo para todas las filas que no pasan el filtro de **where**.

Capítulo 65. Características Avanzadas de SQL en Postgres

Habiendo cubierto los aspectos básicos de Postgre SQL para acceder a los datos, discutiremos ahora aquellas características de Postgres que los distinguen de los gestores de bases de datos convencionales. Estas características incluyen herencia, time travel (viaje en el tiempo) y valores no-atómicos de datos (atributos basados en vectores y conjuntos). Los ejemplos de esta sección pueden encontrarse también en `advance.sql` en el directorio del tutorial. (Consulte el Capítulo 64 para ver la forma de utilizarlo).

Herencia

Creemos dos clases. La clase `capitals` contiene las capitales de los estados, las cuales son también ciudades. Naturalmente, la clase `capitals` debería heredar de `cities`.

```
CREATE TABLE cities (
    name          text,
    population     float,
    altitude       int      - (in ft)
);

CREATE TABLE capitals (
    state          char(2)
) INHERITS (cities);
```

En este caso, una instancia de `capitals` *hereda* todos los atributos (`name`, `population` y `altitude`) de su padre, `cities`. El tipo del atributo `name` (nombre) es `text`, un tipo nativo de Postgres para cadenas ASCII de longitud variable. El tipo del atributo `population` (población) es `float`, un tipo de datos, también nativo de Postgres, para números de punto flotante de doble precisión. La clase `capitals` tiene un atributo extra, `state`, que muestra a qué estado pertenecen. En Postgres, una clase puede heredar de ninguna o varias otras clases, y una consulta puede hacer referencia tanto a todas las instancias de una clase como a todas las instancias de una clase y sus descendientes.

Nota: La jerarquía de la herencia es un gráfico acíclico dirigido.

Por ejemplo, la siguiente consulta encuentra todas aquellas ciudades que están situadas a un altura de 500 o más pies:

```
SELECT name, altitude
FROM cities
WHERE altitude > 500;
```

```
+-----+-----+
|name      | altitude |
+-----+-----+
|Las Vegas | 2174     |
+-----+-----+
|Mariposa  | 1953     |
+-----+-----+
```

Por otro lado, para encontrar los nombres de todas las ciudades, incluidas las capitales estatales, que estén situadas a una altitud de 500 o más pies, la consulta es:

```
SELECT c.name, c.altitude
```

```
FROM cities* c
WHERE c.altitude > 500;
```

which returns:

```
+-----+-----+
|name      | altitude |
+-----+-----+
|Las Vegas | 2174     |
+-----+-----+
|Mariposa  | 1953     |
+-----+-----+
|Madison   | 845      |
+-----+-----+
```

Aquí el `""` después de `cities` indica que la consulta debe realizarse sobre `cities` y todas las clases que estén por debajo de ella en la jerarquía de la herencia. Muchos de los comandos que ya hemos discutido (**`select`**, **`and>upand>`** and **`delete`**) brindan soporte a esta notación de `""` al igual que otros como **`alter`**.

Valores No-Atómicos

Uno de los principios del modelo relacional es que los atributos de una relación son atómicos. Postgres no posee esta restricción; los atributos pueden contener sub-valores a los que puede accederse desde el lenguaje de consulta. Por ejemplo, se pueden crear atributos que sean vectores de alguno de los tipos base.

Vectores

Postgres permite que los atributos de una instancia sean definidos como vectores multidimensionales de longitud fija o variable. Puede crear vectores de cualquiera de los tipos base o de tipos definidos por el usuario. Para ilustrar su uso, creemos primero una clase con vectores de tipos base.

```
CREATE TABLE SAL_EMP (
    name          text,
    pay_by_quarter int4[],
    schedule      text[][]
);
```

La consulta de arriba creará una clase llamada `SAL_EMP` con una cadena del tipo `text` (`name`), un vector unidimensional del tipo `int4` (`pay_by_quarter`), el cual representa el salario trimestral del empleado y un vector bidimensional del tipo `text` (`schedule`), que representa la agenda semanal del empleado. Ahora realizamos algunos `INSERTS`; note que cuando agregamos valores a un vector, encerramos los valores entre llaves y los separamos mediante comas. Si usted conoce C, esto no es distinto a la sintaxis para inicializar estructuras.

```
INSERT INTO SAL_EMP
VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {}}');
```



```
INSERT INTO SAL_EMP
VALUES ('Carol',
       '{20000, 25000, 25000, 25000}',
       '{{"talk", "consult"}, {"meeting"}}');
```

Postgres utiliza de forma predeterminada la convención de vectores "basados en uno" – es decir, un vector de n elementos comienza con `vector[1]` y termina con `vector[n]`. Ahora podemos ejecutar algunas consultas sobre `SAL_EMP`. Primero mostramos como acceder a un solo elemento del vector por vez. Esta consulta devuelve los nombres de los empleados cuyos pagos han cambiado en el segundo trimestre:

```
SELECT name
FROM SAL_EMP
WHERE SAL_EMP.pay_by_quarter[1] <>
      SAL_EMP.pay_by_quarter[2];
```

```
+-----+
|name   |
+-----+
|Carol  |
+-----+
```

La siguiente consulta recupera el pago del tercer trimestre de todos los empleados:

```
SELECT SAL_EMP.pay_by_quarter[3] FROM SAL_EMP;
```

```
+-----+
|pay_by_quarter |
+-----+
|10000          |
+-----+
|25000          |
+-----+
```

También podemos acceder a cualquier porción de un vector, o subvectores. Esta consulta recupera el primer ítem de la agenda de Bill para los primeros dos días de la semana.

```
SELECT SAL_EMP.schedule[1:2][1:1]
FROM SAL_EMP
WHERE SAL_EMP.name = 'Bill';
```

```
+-----+
|schedule          |
+-----+
|{{"meeting"}, {" "}} |
+-----+
```

Time Travel (Viaje en el tiempo)

Al momento de salir la versión 6.2 de Postgres v6.2, *la característica de viaje en el tiempo (time travel) ya no está soportada*. Existen varias razones para esto: impacto sobre el rendimiento, el tamaño de almacenamiento, y un archivo pg_time que crece hasta el infinito en poco tiempo.

En cambio, dispone de nuevas características como los disparadores (triggers) que permiten imitar el comportamiento del viaje en el tiempo cuando se desee, sin incurrir en sobrecarga cuando no se necesita (en general, la mayor parte del tiempo). Vea los ejemplos en el directorio contrib para mayor información.

Time travel ha sido descartado:: : El texto restante en esta sección se conserva solamente hasta que pueda ser reescrito ajustándose al contexto de las nuevas técnicas que permiten obtener los mismos resultados. ¿Voluntarios? - thomas 12-01-1998.

Postgres soporta la idea del viaje en el tiempo. Esto permite a un usuario correr consultas históricas. Por ejemplo, para encontrar la población actual de la ciudad de Mariposa, usted debería realizar la siguiente consulta:

```
SELECT * FROM cities WHERE name = 'Mariposa';
```

```
+-----+-----+-----+
|name      | population | altitude |
+-----+-----+-----+
|Mariposa  | 1320       | 1953     |
+-----+-----+-----+
```

Postgres automáticamente encontrará la versión del registro de Mariposa válida para este momento. Usted también podría especificar un intervalo de tiempo. Por ejemplo, para ver la población pasada y presente de la ciudad de Mariposa, usted correría la siguiente consulta:

```
SELECT name, population
       FROM cities['epoch', 'now']
       WHERE name = 'Mariposa';
```

donde "epoch" indica el comienzo del reloj del sistema.

Nota: En los sistemas Unix, esto siempre es la medianoche del 1 de enero de 1970, GMT.

Si ha realizado todos los ejemplos hasta ahora, la consulta anterior devolverá:

```
+-----+-----+
|name      | population |
+-----+-----+
|Mariposa  | 1200       |
+-----+-----+
|Mariposa  | 1320       |
+-----+-----+
```

El valor predeterminado para el comienzo del intervalo de tiempo es el menor valor que pueda representar el sistema, mientras que el valor predeterminado para el final del intervalo es la hora actual. Por lo tanto, el intervalo de tiempo utilizado en la consulta anterior podría haberse abreviado como “[].”

Más características avanzadas

Postgres posee muchas características que no se han visto en este tutorial, el cual ha sido orientado hacia usuarios nuevos de SQL. Las mencionadas características se discuten tanto en la Guía del Usuario como en la del Programador.

Apéndice UG1. ayuda de fecha/hora

Zonas horarias

Postgres debe tener información tabular interna para decodificar la zona horaria, desde que no hay un sistema estandar de interface *nix para proveer acceso a lo general, información de zona de tiempo cruzada. El SO subyacente *es* usado para proveer información de zona de tiempo para *salidas*.

Tabla UG1-1. Zonas de tiempo reconocidas por Postgres

Zona de Tiempo	fuera de UTC	descripción
NZDT	+13:00	Hora de luz del día de nueva Zelanda
IDLE	+12:00	Fecha internacional lineal, Este
NZST	+12:00	Hora Std de Nueva Zelanda
NZT	+12:00	Hora de Nueva Zelanda
AESST	+11:00	Hora de verano Std de Australia del este
ACSST	+10:30	Hora de verano Std de Australia Central
CADT	+10:30	Hora de luz del día de Australia
SADT	+10:30	Hora de luz del día de Australia del sur
AEST	+10:00	Hora Std de Australia del este
EAST	+10:00	Hora Std de Australia del Este
GST	+10:00	Hora de Guam Std, Zona 9 de USSR
LIGT	+10:00	Melbourne, Australia
ACST	+09:30	Hora Std de Australia Central
CAST	+09:30	Hora Std de Australia Central
SAT	+9:30	Hora Std de Australia del sur
AWSST	+9:00	Hora Std de verano de Australia del oeste
JST	+9:00	Hora Std de Japón, Zona 8 de USSR
KST	+9:00	Hora estandar de Korea

Zona de Tiempo	fuera de UTC	descripción
WDT	+9:00	Hora de luz del día del Oeste de Australia
MT	+8:30	Hora de Moluccas
AWST	+8:00	Hora Std de Australia del oeste
CCT	+8:00	Hora de la costa de China
WADT	+8:00	Hora de luz del día del oeste de australia
WST	+8:00	Hora Std del Oeste de Australia
JT	+7:30	Hora de Java
WAST	+7:00	Hora Std del Oeste de Australia
IT	+3:30	Hora de Irán
BT	+3:00	Hora de Baghdad
EETDST	+3:00	Hora de luz del día en Europa del este
CETDST	+2:00	Hora de luz del día en Europa Central
EET	+2:00	Europa del Este,Zona 1 de USSR
FWT	+2:00	Hora de invierno Frances
IST	+2:00	Hora Std de Israel
MEST	+2:00	Hora de verano de Europa del centro
METDST	+2:00	Hora de luz del día en Europa del centro
SST	+2:00	Hora de verano de Suecia
BST	+1:00	Hora de verano de Inglaterra
CET	+1:00	Hora de Europa central
DNT	+1:00	Hora normal de Dansk
DST	+1:00	Hora estandart de Dansk(?)
FST	+1:00	Hora de verano Francesa
MET	+1:00	Hora de Europa del Centro
MEWT	+1:00	Hora de invierno de Europa del Centro
MEZ	+1:00	Zona de Europa del Centro
NOR	+1:00	Hora estandart de Norway
SET	+1:00	Hora de Seychelles
SWT	+1:00	Hora de invierno de Suecia

Zona de Tiempo	fuera de UTC	descripción
WETDST	+1:00	Hora de luz del día del Oeste de Europa
GMT	0:00	Hora principal de Greenwich
WET	0:00	Europa del Oeste
WAT	-1:00	Hora del oeste de Africa
NDT	-2:30	Hora de luz del día de Newfoundland
ADT	-03:00	Hora de luz del día de Atlantic
NFT	-3:30	Hora estandar de Newfoundland
NST	-3:30	Hora estandar de Newfoundland
AST	-4:00	Hora Std de Atlantic(Canada)
EDT	-4:00	Hora de luz del día del este
ZP4	-4:00	GMT +4 hours
CDT	-5:00	Hora de luz del día Central
EST	-5:00	Hora estandar del este
ZP5	-5:00	GMT +5 hours
CST	-6:00	Hora Std Central
MDT	-6:00	Hora de luz del día de la Montaña
ZP6	-6:00	GMT +6 hours
MST	-7:00	Hora estandar de la montaña
PDT	-7:00	Hora de luz del día del Pacífico
PST	-8:00	Hora Std del Pacífico
YDT	-8:00	Hora de luz del día de Yukon
HDT	-9:00	Hora de luz del día en Hawaii/ Alaska
YST	-9:00	Hora estandar de Yukon
AHST	-10:00	Hora Std de Alaska-Hawaii
CAT	-10:00	Hora de Alaska Central
NT	-11:00	Hora Nome
IDLW	-12:00	Linea de Fecha Internacional, Oeste

Zonas Horarias Australianas

Las zonas horarias Australianas y sus variantes de denominación cuentan con un curato de la totalidad de las zonas horarias de la tabla de búsqueda de las zonas horarias de Postgres. Hay dos conflictos de denominación con zonas horarias en común definidas en los Estados Unidos, CST y EST.

Si la opción del compilador USE_AUSTRALIAN_RULES esta activa entonces CST y EST se interpretaran siguiendo los convenios Australianos.

Tabla UG1-2. Zonas Horarias Australianas de Postgres

Zona Horaria	Desplazamiento desde UTC	Descripción
CST	+10:30	Tiempo Estándar Central de Australia
EST	+10:00	Tiempo Estándar Oriental de Australia

Interpretación de las entradas de Fecha/tiempo

Los tipos de fecha/tiempo son todos decodificados usando un conjunto de rutinas comunes.

Interpretación de las entradas de Fecha/tiempo

- Partiendo la cadena de entrada en muestras y clasificando cada uno de las marcas como cadena, tiempo, zona horaria, o número.
 - Si la muestra contiene dos puntos (":"), esto es una cadena de tiempo.
 - si la muestra contiene un guión ("-"), barra ("/"), o un punto ("."), esto es una cadena de fecha que puede tener el nombre del mes.
 - Si la muestra es solamente numérica, entonces es cualquiera de estas opciones un campo sencillo un fecha concatenada ISO-8601 (p.e. "19990113" para 13 Enero del 1999) o tiempo (p. e. 141516 para 14:15:16).
 - Si la muestra comienza con un mas ("+") o un menos ("-"), entonces es o una zona horaria o un campo especial.
- Si la muestra es una cadena de texto, compara con posibles cadenas.
 - Hacer un búsqueda binaria en la tabla de consulta de la muestra para cada cadena especial (p. e. today), day (p. e. Thursday), month (p. e. January), o noise word (p. e. on).
Pone los valores del campo y la mascara de bit para los campos. Por ejemplo, pone año, mes, día para today, y adicionalmente hora, minutos, segundos para now.
 - Si no lo encuentra, hace una búsqueda binaria similar en la tabla de consulta para encontrar la muestra a la zona horaria.
 - Si no lo encuentra, lanza un error.
- La muestra es un número o un campo numérico.

- a. Si hay más de 4 dígitos, y si no se ha leído con posterioridad otro campo de tipo fecha, entonces lo interpretará como un "fecha concatenada" (e.g. 19990118). Con 8 y 6 dígitos se interpreta como año, mes, y día, mientras que con 7 y 5 dígitos se interpreta como año, día del año, respectivamente.
 - b. Si la muestra tiene 3 dígitos y un año ha sido decodificado, entonces se interpreta como día del año.
 - c. Si es más largo que dos dígitos, entonces se interpreta como el año.
 - d. Si está en modo fecha Europea, y si el campo día no ha sido leído todavía, y si el valor es más pequeño o igual a 31, entonces se interpreta como un día.
 - e. Si el campo mes no ha sido leído todavía, y si el valor es más pequeño o igual que 12, entonces se interpreta como un mes.
 - f. Si el campo día no ha sido leído todavía, y si el valor es más pequeño o igual que 31, entonces se interpreta como un día.
 - g. Si no, se interpreta como un año.
4. Si se ha especificado AC, anula el año y desplaza uno al almacenado interno (no hay año cero en el calendario Gregoriano, pero numéricamente 1AC es el año cero).
 5. Si no se ha especificado, y si el campo año tiene dos dígitos de longitud, entonces ajustamos el año a 4 dígitos. Si el campo no es más pequeño que 70, entonces sumamos 2000; si no, sumamos 1900.

Sugerencia: Los años Gregorianos 1-99AD pueden ser introducidos usando 4 dígitos precedidos por ceros (p. e. 0099 es 99AD). Los tres dígitos también son aceptados como un año bajo muchas circunstancias, sin embargo dependiendo de la posición la cadena numérica puede ser interpretada en lugar de un día.

Historia

Nota: Contrido por José Soares¹.

El día Juliano fue inventado por erudito francés Joseph Justus Scaliger (1540-1609) y probablemente coge su nombre del padre Scaliger, el erudito italiano Julius Caesar Scaliger (1484-1558). Los astrónomos tienen que usar el periodo Juliano para asignar un único número cada día desde 1 de Enero de 4713 AC. Esto es el llamado Día Juliano (JD). JD 0 designa 24 horas, del mediodía UTC del 1 de Enero de 4713 AC hasta el mediodía UTC del 2 Enero 4713 AC.

“Día Juliano” es diferente que “Fecha Juliana”. El calendario Juliano fue introducido por Julius Caesar en 45 AC. Fue usado comúnmente hasta el 1582, donde países empezaron a cambiarse al calendario Gregoriano. En el calendario Juliano, el año tropical es aproximadamente como $365 \frac{1}{4}$ días = 365.25 días. Esto da un error de un día en aproximadamente 128 días. El error acumulado del calendario movió al Papa Gregorio XIII a reformar el calendario acorde con las instrucciones del Concilio de Trento.

En el calendario Gregoriano, el año tropical es aproximadamente $365 + 97 / 400$ días = 365.2425 días. Así coge aproximadamente 3300 años para el año tropical se desplace un día con respecto al calendario Gregoriano.

La aproximación $365+97/400$ esta lograda mediante 97 años bisiestos cada 400 años, usando las siguientes reglas:

Cada año divisible por 4 es un año bisiesto.

Sin embargo, cada año divisible por 100 no es un año bisiesto.

Sin embargo, cada año divisible por 400 es un año bisiesto después de todo.

De este modo, 1700, 1800, 1900, 2100, y 2200 no son años bisiestos. pero 1600, 2000, y 2400 son años bisiestos. Por el contrario, en el viejo calendario Juliano sólo los años divisibles por 4 son años bisiestos.

La bula papal de Febrero del 1582 decretó que se debía quitar 10 días a Octubre de 1582, así que el 15 de Octubre debe seguir inmediatamente después del 4 de Octubre. Esto se observó en Italia, Polonia, Portugal, y España. Los otros países Católicos lo siguieron poco después, pero los países Protestantes se resistieron al cambio, y los países ortodoxos griegos no cambiaron hasta que no empezó este siglo. La reforma fue observada por Gran Bretaña y sus Colonias (incluido lo que ahora es USA) en 1752. Así el 2 Septiembre 1752 fue seguido por el 14 Septiembre 1752. Esto es lo que tiene cal de los sistemas UNIX produciendo lo siguiente:

```
% cal 9 1752
      Septiembre 1752
  S  M Tu  W Th  F  S
                1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

Nota: SQL92 dista que “Dentro de la definición del fecha/tiempo literal , los valores fecha/tiempo están restringidos por las reglas naturales para las fechas y los tiempos acorde con el calendario Gregoriano “. Las Fechas entre 1752-09-03 y 1752-09-13, aunque se han eliminado en algunos países por el "fiat" Papal, conforme a las “reglas naturales” y son por lo tanto fechas validas.

Diferentes calendarios han sido desarrollados en varios lugares del mundo, muchos proceden del sistema Gregoriano. Por ejemplo, Los principios del calendario Chino pueden remontarse hasta el siglo 14 AC. La leyenda dice que el Emperador Huangdi inventó el calendario en 2637 AC. La gente de la República de China usa el calendario Gregoriano para uso civil. El calendario Chino es utilizado para las fiestas.

Notas

1. jose@sferacarta.com

Apéndice DG1. El Repositorio del CVS

El código fuente de Postgres se almacena y administra utilizando el sistema de gestión de código CVS.

Hay al menos dos métodos, CVS anónimo y CVSup, utilizables para copiar el árbol del código de CVS desde el servidor de Postgres a su máquina local.

Organización del árbol de CVS

Author: Escrito por Marc G. Fournier¹ el 1998-11-05.

Traductor: Traducido por Equipo de traducción de PostgreSQL² el 2001-03-14.

(N. del T: Ismael Olea ha escrito un estupendo documento llamado "*Micro-cómo empezar a trabajar con cvs*", muy fácil de entender y de utilizar, y que puede resultar muy interesante para los que sólo deseen utilizar un cliente de CVS de modo genérico. Como él también colabora en la traducción, no puedo por menos de recomendarlo.

Lo pueden conseguir en su página personal³ y desde luego pidiendoselo directamente a él olea@hispa Fuentes.com⁴. Fin de la N. del T.)

El comando **cvs checkout** tiene un indicador (flag), **-r**, que le permite comprobar una cierta revisión de un módulo. Este indicador facilita también, por ejemplo, recuperar las fuentes que formaban la release 1.0 del módulo 'tc' en cualquier momento futuro:

```
$ cvs checkout -r REL6_4 tc
```

Esto es utilizable, por ejemplo, si alguien asegura que hay un error (un bug) en esa release, y usted no es capaz de encontrarlo en la copia de trabajo actual.

Sugerencia: También puede usted comprobar un módulo conforme era en cualquier momento dado utilizando la opción **-D**.

Cuando etiquete usted más de un fichero con la misma etiqueta, puede usted pensar en las etiquetas como "una línea curva que recorre una matriz de nombres de ficheros contra número de revisión". Digamos que tenemos 5 ficheros con las siguientes revisiones:

fich1	fich2	fich3	fich4	fich5	
1.1	1.1	1.1	1.1	/-1.1*	<--* TAG (etiqueta)
1.2*-	1.2	1.2	-1.2*-		
1.3 \-	1.3*-	1.3	/ 1.3		
1.4		1.4	/ 1.4		
		\-1.5*-	1.5		
		1.6			

donde la etiqueta “TAG” hará referencia a fich1-1.2, fich2-1.3, etc.

Nota: Para crear la rama de una nueva release, se emplea de nuevo el comando -b, del mismo modo anterior.

De este modo, para crear la release v6.4, hice lo siguiente:

```
$ cd pgsql
$ cvs tag -b REL6_4
```

lo cual creará la etiqueta y la rama para el árbol RELEASE.

Ahora, para aquellos con acceso CVS, también es sencillo. Primero, cree dos subdirectorios, RELEASE y CURRENT, de forma que no mezcle usted los dos. A continuación haga:

```
cd RELEASE
cvs checkout -P -r REL6_4 pgsql
cd ../CURRENT
cvs checkout -P pgsql
```

lo que dará lugar a dos árboles de directorios, RELEASE/pgsql y CURRENT/pgsql. A partir de este momento, CVS tomará el control de qué rama del repositorio se encuentra en cada árbol de directorios, y permitirá actualizaciones independientes de cada árbol.

Si usted *sólo* está trabajando en el árbol fuente CURRENT hágalo todo tal como empezamos antes etiquetando las ramas de la release. If you are *only* working on the CURRENT source tree, you just do everything as before we started tagging release branches.

Una vez que usted realiza el checkout (igualado, comprobación, descarga) inicial en una rama,

```
$ cvs checkout -r REL6_4
```

todo lo que usted haga dentro de esa estructura de directorios se restringe a esa rama. Si usted aplica un patch a esa estructura de directorios y hace un

```
cvs commit
```

mientras usted se encuentra dentro de ella, el patch se aplica a esa rama y *sólo* a esa rama.

Tomando Las Fuentes Vía CVS Anónimo

Si quisiera usted mantenerse proximo a las fuentes actuales de una forma regular, podría usted ir a buscarlos a nuestro propio servidor CVS y utilizar entonces CVS para recuperar las actualizaciones de tiempo en tiempo.

CVS Anónimo

1. Necesitará usted una copia local de CVS (Concurrent Version Control System, Sistema de Control de Versiones Concurrentes -simultáneas-), que puede usted tomar de <http://www.cyclic.com/>⁵ o cualquier otra dirección que archive software GNU. Actualmente recomendamos la versión 1.10 (la más reciente en el momento de escribir). Muchos sistemas tienen una versión reciente de cvs instalada por defecto.

2. Haga una conexión (login) inicial al servidor CVS:

```
$ cvs -d :pserver:anoncvs@postgresql.org:/usr/local/cvsroot login
```

Se le preguntará su password; introduzca 'postgresql'. Sólo necesitará hacer esto una vez, pues el password se almacenará en .cvspass, en su directorio de defecto (your home directory).

3. Descargue las fuentes de Postgres:

```
cvs -z3 -d :pserver:anoncvs@postgresql.org:/usr/local/cvsroot co -P pgsql
```

lo cual instala las fuentes de Postgres en un subdirectorio pgsql del directorio en el que usted se encuentra.

Nota: Si tiene usted una conexión rápida con Internet, puede que no necesite `-z3`, que instruye a CVS para utilizar compresión gzip para la transferencia de datos. Pero en una conexión a velocidad de modem, proporciona una ventaja muy sustancial.

Esta descarga inicial es un poco más lenta que simplemente descargar un fichero `tar.gz`; con un modem de 28.8K, puede tomarse alrededor de 40 minutos. La ventaja de CVS no se muestra hasta que intenta usted actualizar nuevamente el fichero.

4. Siempre que quiera usted actualizar las últimas fuentes del CVS, **cd** al subdirectorio `pgsql`, y ejecute

```
$ cvs -z3 update -d -P
```

Esto descargará sólo los cambios producidos desde la última actualización realizada. Puede usted actualizar en apenas unos minutos, típicamente, incluso con una línea de velocidad de modem.

5. Puede usted mismo ahorrarse algo de tecleo, creando un fichero `.cvsrc` en su directorio de defecto que contenga:

```
cvs -z3
update -d -P
```

Esto suministra la opción `-z3` a todos los comandos al cvs, y las opciones `-d` y `-P` al comando `cvs update`. Ahora, simplemente tiene que teclear

```
$ cvs update
```

para actualizar sus ficheros.

Atención

Algunas versiones anteriores de CVS tenían un error que llevaba a que todos los ficheros comprobados se almacenasen con permisos de escritura para todo el mundo (777) en su directorio. Si le ha pasado esto, puede usted hacer algo como

```
$ chmod -R go-w pgsq1
```

para colocar los permisos adecuadamente. Este error se fijó a partir de la versión 1.9.28 de CVS.

CVS puede hacer un montón de otras cosas, del tipo de recuperar revisiones previas de los fuentes de Postgres en lugar de la última versión de desarrollo. Para más información, consulte el manual que viene con CVS, o mire la documentación en línea en <http://www.cyclic.com/>⁶.

Tomando Los Fuentes Vía CVSup

Una alternativa al uso de CVS anónimo para recuperar el árbol fuente de Postgres es CVSup. CVSup fué desarrollado por John Polstra⁷ para distribuir repositorios CVS y otro árboles de ficheros para El proyecto FreeBSD⁸.

Una ventaja importante de utilizar CVSup es que puede replicar de forma eficaz el repositorio *entero* en su sistema local, permitiendo un acceso local rápido a las operaciones de cvs como `log` y `diff`. Otras ventajas incluyen sincronización rápida al servidor de Postgres debido a un eficiente protocolo de transferencia de cadenas que sólo envía los cambios desde la última actualización.

Preparando un Sistema Cliente CVSup

Se requieren dos áreas para que CVSup pueda hacer su trabajo: un repositorio local de CVS (o simplemente un área de directorios si usted está tomando una foto fija (snapshot) en lugar de un repositorio; vea más abajo) y área local de anotaciones de CVSup. Estas dos áreas pueden coexistir en el mismo árbol de directorios.

Decida donde quiere usted conservar su copia local del repositorio CVS. En uno de nuestros sistemas, recientemente hemos instalado un repositorio en `/home/cvs/`, pero anteriormente lo teníamos bajo un árbol de desarrollo de Postgres en `/opt/postgres/cvs/`. Si desea usted mantener su repositorio en `/home/cvs/`, incluya

```
setenv CVSROOT /home/cvs
```

en su fichero `.cshrc`, o una línea similar en su fichero `.bashrc` o `.profile`, dependiendo de su shell.

Se debe inicializar el área del repositorio de cvs. Una vez que se fija CVSROOT, se puede hacer esto con un único comando:

```
$ cvs init
```

tras lo cual, debería usted ver al menos un directorio llamado `CVSROOT` cuando liste el directorio `CVSROOT`:

```
$ ls $CVSROOT
CVSROOT/
```

Ejecutando un Cliente CVSup

Verifique que `cvsup` se encuentra en su path; en la mayoría de los sistemas, puede usted hacer esto tecleando

```
which cvsup
```

Entonces, simplemente ejecute `cvsup` utilizando:

```
$ cvsup -L 2 postgres.cvsup
```

donde `-L 2` activa algunos mensajes de status para que pueda usted monitorizar el progreso de la actualización, y `postgres.cvsup` es la ruta y el nombre que usted ha dado a su fichero de configuración de CVSup.

Aquí le mostramos un fichero de configuración de CVSup modificado para una instalación específica, y que mantiene un repositorio CVS local completo: (N. del T: voy a traducir los comentarios a modo de documentación del fichero. Obviamente, no traduciré los comandos, lo que puede dar una imagen algo complicada, pero me parece que puede valer la pena. Agradeceremos sus comentarios a doc-postgresql-es@hispalinux.es)

```
# Este fichero representa el fichero de distribución estandar de CVSup
# para el proyecto de ORDBMS PostgreSQL.
# Modificado por lockhart@alumni.caltech.edu 1997-08-28
# - Apunta a mi foto fija local del árbol fuente.
# - Recupera el repositorio CVS completo, y no sólo la última actualización.
#
# Valores de defecto que se aplican a todas las recolecciones.
*default host=postgresql.org
*default compress
*default release=cvs
*default delete use-rel-suffix
# activar la línea siguiente para tomar la última actualización.
*#default tag=.
# activar la línea siguiente para tomar todo lo que se ha especifica-
do antes
# o por defecto en la fecha especificada a continuación.
*#default date=97.08.29.00.00.00

# el directorio base apunta a donde CVSup almacenará sus ficheros de marcas.
# creará un subdirectorio sup/
*#default base=/opt/postgres # /usr/local/pgsql
*default base=/home/cvs

# el directorio prefijo apunta a donde CVSup almacenará la/s distribu-
ción/es actuales.
*default prefix=/home/cvs

# la distribución completa, incluyendo todo lo siguiente.
pgsql
```

```
# distribuciones individuales contra 'el paquete completo'
# pgsql-doc
# pgsql-perl5
# pgsql-src
```

El siguiente fichero de configuración de CVSup se sugiere en el servidor ftp de Postgres⁹ y descargará únicamente la foto fija actual:

```
# Este fichero representa el fichero de distribución estandar de CVSup
# para el proyecto de ORDBMS PostgreSQL.
#
# Valores de defecto que se aplican a todas las recolecciones, a todas las descargas.
*default host=postgresql.org
*default compress
*default release=cvs
*default delete use-rel-suffix
*default tag=.

# el directorio base apunta a donde CVSup almacenará sus ficheros de marcas.
*default base=/usr/local/pgsql

# el directorio prefijo apunta a dnde CVSup almacenará las distribucio-
# nes actuales.
*default prefix=/usr/local/pgsql

# distribución completa, incluyendo todo lo siguiente.
pgsql

# distribuciones individuales contra 'el paquete completo'
# pgsql-doc
# pgsql-perl5
# pgsql-src
```

Instalando CVSup

CVSup se puede adquirir como ficheros fuentes, binarios preconstruidos o RPM,s de Linux. Es mucho más facil utilizar un binario que construirlo a partir de los fuentes, principalmente porque el compilador Modula-3, muy capaz pero también muy voluminoso, se necesita para la construcción.

Instalación a partir de Binarios de CVSup

Puede usted utilizar los binarios si tiene una plataforma para la que los binarios se hayan remitido al servidor ftp de Postgres¹⁰, o si está usted utilizando FreeBSD, para el que CVSup está disponible como una adaptación (porting).

Nota: CVSup fue desarrollado originariamente como una herramienta para la distribución del árbol fuente de FreeBSD. Está disponible como una adaptación, y para aquellos que utilizan FreeBSD, si esto no es suficiente para decirles como obtenerlo e instalarlo, les agradeceremos que nos aporten un procedimiento eficaz.

En el momento de escribir, se disponen binarios para Alpha/Tru64, ix86/xBSD, HPPA/HPUX-10.20, MIPS/irix, ix86/linux-libc5, ix86/linux-glibc, Sparc/Solaris, and Sparc/SunOS.

1. Adquiera el fichero tar con los binarios para cvsup (cvsupd no se requiere para ser un cliente) adecuado para su plataforma.

- a. Si utiliza usted FreeBSD, instale la adaptación de CVSup.
- b. Si tiene usted otra plataforma, localice y descargue los binarios apropiados desde el servidor ftp de Postgres¹¹.

2. Compruebe el fichero tar para verificar el contenido y la estructura de directorios, si la hay. Al menos para el fichero tar de linux, los binarios estáticos y las páginas man se incluyen sin ningún empaquetado de directorios.

- a. Si el binario se encuentra en el nivel superior del fichero tar, simplemente desempaquete el fichero tar en su directorio elegido:

```
$ cd /usr/local/bin
$ tar zxvf /usr/local/src/cvsup-16.0-linux-i386.tar.gz
$ mv cvsup.1 ../doc/man/man1/
```

- b. Si hay una estructura de directorios en el fichero tar, desempaquete el fichero tar en /usr/local/src, y mueva los binarios a la dirección adecuada como antes.

3. Asegúrese de que los nuevos binarios se encuentran en su path.

```
$ rehash
$ which cvsup
$ set path=(path a cvsup $path)
$ which cvsup
/usr/local/bin/cvsup
```

Instalación a partir de los Fuentes.

La instalación a partir de los fuentes de CVSup no es totalmente trivial, principalmente porque la mayoría de sistemas necesitarán antes el compilador Modula-3. Este compilador se encuentra disponible como RPM de Linux, como paquete FreeBSD, o como código fuente.

Nota: Una instalación limpia de Modula-3 se lleva aproximadamente 200 MB de espacio en disco, de los que se pueden recuperar unos 50 MB cuando se borren los fuentes.

Instalación en Linux

1. Instale Modula-3.
 - a. Tome la distribución de Modula-3 desde Polytechnique Montréal¹², quien mantiene activamente el código base originalmente desarrollado por the

DEC Systems Research Center¹³. La distribución RPM “PM3” está comprimida aproximadamente unos 30 MB. En el momento de escribir, la versión 1.1.10-1 se instalaba limpiamente en RH-5.2, mientras que la 1.1.11-1 estaba construída aparentemente para otra versión (¿RH-6.0?) y no corría en RH-5.2.

Sugerencia: Este empaquetado rpm particular tiene *muchos* ficheros RPM, de modo que seguramente quiera usted situarlos en un directorio aparte.

- b. Instale los rpms de Modula-3:

```
# rpm -Uvh pm3*.rpm
```

2. Desempaquete la distribución de cvsup:

```
# cd /usr/local/src
# tar xzf cvsup-16.0.tar.gz
```

3. Construya la distribución de cvsup, suprimiendo la interface gráfica para evitar la necesidad de las librerías X11:

```
# make M3FLAGS="-DNOGUI"
```

Y si quiere construir un binario estático para trasladarlo a sistemas en los cuales no pueda tener instalado Modula-3, intente:

```
# make M3FLAGS="-DNOGUI -DSTATIC"
```

4. Instale el binario construido:

```
# make M3FLAGS="-DNOGUI -DSTATIC" install
```

Notas

1. <mailto:scrappy@hub.org>
2. <mailto:doc-postgresql-es@listas.hispalinux.org>
3. <http://slug.HispaLinux.ES/~olea/micro-como-empezar-con-cvs.html>
4. <mailto:olea@hispafuentes.com>
5. <http://www.cyclic.com/>
6. <http://www.cyclic.com/>
7. <mailto:jdp@polstra.com>
8. <http://www.freebsd.org>
9. <ftp://ftp.postgresql.org/pub/CVSup/README.cvsup>
10. <ftp://postgresql.org/pub>

11. <ftp://postgresql.org/pub>
12. <http://m3.polymtl.ca/m3>
13. <http://www.research.digital.com/SRC/modula-3/html/home.html>

Apéndice DG2. Documentación

El propósito de la documentación es hacer de Postgres más fácil de aprender y desarrollar. El conjunto de esta documentación describe el sistema Postgres, su lenguaje y su interfaz. Esta documentación debe ser capaz de responder a las cuestiones más frecuentes y permitir al usuario encontrar respuestas por sí mismo, sin tener que recurrir al soporte de listas de correo.

Mapa de la documentación

Postgres tiene cuatro formatos básicos de documentos:

- Texto plano con información acerca de la pre-instalación.
- HTML, que se usa para navegación on-line y como referencia.
- Documentación impresa para lecturas más detenidas y también como referencia.
- páginas man como referencia rápida.

Tabla DG2-1. Documentación de Postgres

Fichero	Descripción
./COPYRIGHT	Apuntes de Copyright
./INSTALL	Instrucciones de instalación (textos sgml->rtf->text)
./README	Información introductoria
./register.txt	Mensajes de registro durante make
./doc/bug.template	Plantilla para informes de depuración
./doc/postgres.tar.gz	Documentos integrados (HTML)
./doc/programmer.ps.gz	Guía del programador (Postscript)
./doc/programmer.tar.gz	Guía del programador (HTML)
./doc/reference.ps.gz	Manual de referencia (Postscript)
./doc/reference.tar.gz	Manual de referencia (HTML)
./doc/tutorial.ps.gz	Introducción (Postscript)
./doc/tutorial.tar.gz	Introducción (HTML)
./doc/user.ps.gz	Guía del usuario (Postscript)
./doc/user.tar.gz	Guía del usuario (HTML)

Se disponen de páginas man, así como como un gran número de ficheros de texto del tipo README en todas las fuentes de Postgres.

El proyecto de documentación

Puede disponer de documentación tanto en formato HTML como *Postscript*, ambos

formatos disponibles como parte de la instalación estándar de Postgres . Aquí se discute acerca de cómo trabajar con las fuentes de la documentación y sobre cómo generar paquetes de documentación.

Las fuentes de la documentación se han escrito usando ficheros de texto plano en formato SGML . El propósito del SGML DocBook es el de permitir a un autor especificar la estructura y el contenido de un documento técnico (usando el DTD DocBook) y también el de tener un documento de estilo que defina cómo ese contenido se verá finalmente (por ejemplo, utilizando Modular Style Sheets, Hojas de Estilo Modular, de Norm Walsh).

Lea *Introduction to DocBook*¹ para tener un buen y rápido resumen de las características de DocBook. *DocBook Elements*² le da una poderosa referencia de las características de DocBook.

El conjunto de esta documentación se ha construido usando varias herramientas, incluyendo *jade*³ de James Clark y *Modular DocBook Stylesheets*⁴ de Norm Walsh.

Normalmente las copias impresas se producen importando ficheros *Rich Text Format* (RTF) desde *jade* a *ApplicxWare* para eliminar algún error de formato y a partir de aquí exportarlo como fichero *Postscript*.

*TeX*⁵ es un formato soportado como salida por *jade*, pero no se usa en estos momentos, principalmente por su incapacidad para hacer formateos pequeños del documento antes de la copia impresa y por el inadecuado soporte de tablas en las hojas de estilo *TeX*.

Fuentes de la documentación

Las fuentes de la documentación pueden venir en formato de texto plano, página de man y html. Sin embargo, la mayoría de la documentación de Postgres se escribirá usando *Standard Generalized Markup Language* (SGML) *DocBook*⁶ *Document Type Definition* (DTD). La mayoría de la documentación ha sido convertida o será convertida a SGML.

El propósito de SGML es el de permitir a un autor especificar la estructura y el contenido de un documento (por ejemplo, usando el DTD DocBook) y permitir que el estilo del documento defina cómo se verá el resultado final (por ejemplo, utilizando las hojas de estilo de Norm Walsh).

La documentación se ha reunido desde varias fuentes. Debido a que integramos y asimilamos la documentación existente y la convertimos en un conjunto coherente, las versiones antiguas pasarán a ser obsoletas y serán borradas de la distribución . Sin embargo, esto no ocurrirá inmediatamente y tampoco ocurrirá con todos los documentos al mismo tiempo. Para facilitar la transición y ayudar a los desarrolladores y escritores de guías, hemos definido un esquema de transición.

Estructura del documento

Actualmente hay cuatro documentos escritos con DocBook. Cada uno de ellos tiene un documento fuente que lo contiene y que define el entorno de Docbook y otros ficheros fuente del documento. Estos ficheros fuente se encuentran en `doc/src/sgml/` junto con la mayoría de otros ficheros fuente usados para la documentación. La fuente primera de ficheros fuente son:

`postgres.sgml`

Este es el documento integrado, incluyendo todos los otros documentos como partes. La salida es generada en HTML, ya que la interfaz del navegador hace

fácil moverse por toda la documentación sólo con pulsaciones del ratón. Los otros documentos están disponibles tanto en formato HTML como en copias impresas.

tutorial.sgml

Es el tutorial introductorio, con ejemplos. No incluye elementos de programación y su intención es la de ayudar al lector no familiarizado con SQL. Este es el documento "getting started", cómo empezar .

user.sgml

Es la Guía del usuario. Incluye información sobre tipos de datos e interfaces a nivel de usuario. Este es el sitio donde emplazar información de "porqués".

reference.sgml

El Manual de referencia. Incluye sintaxis de Postgres SQL . Este es el lugar donde recoger información de los "cómo".

programming.sgml

Es la Guía del programador. Incluye información sobre la extensibilidad de Postgres y sobre las interfaces de programación.

admin.sgml

La Guía del administrador. Abarca la instalación y notas de la versión.

Estilos y convenciones

DocBook tiene un rico conjunto de etiquetas y conceptos y un gran número de ellos son muy útiles para documentación bien formada. Sólo recientemente el conjunto de la documentación de Postgres ha sido adaptada a SGML y en un futuro próximo varias partes de ese conjunto serán seleccionadas y mantenidas como ejemplos del uso de DocBook . También se incluirá abajo un pequeño sumario de etiquetas de DocBook.

Herramientas de autor para SGML

Actualmente la documentación de Postgres se ha escrito usando un editor de textos normal (o emacs/psgml, véase más abajo) con el marcado del contenido utilizando etiquetas SGML de DocBook.

SGML y DocBook no se ven afectados debido a las numerosas herramientas de autor de código abierto. El conjunto de herramientas más usado es el paquete de edición emacs/xemacs con la extensión psgml. En algunos sistemas (por ejemplo, RedHat Linux) estas herramientas se encuentran en la instalación de la distribución.

emacs/psgml

emacs (y xemacs) tienen un modo de trabajo (*major mode*) SGML. Si se configura correctamente le permitirá utilizar emacs para insertar etiquetas y controlar la consistencia de las marcas.

Introduzca las siguientes líneas en su fichero ~/.emacs (ajustando el path si fuera necesario):

```
; ***** for SGML mode (psgml)

(setq sgml-catalog-files "/usr/lib/sgml/CATALOG")
(setq sgml-local-catalogs "/usr/lib/sgml/CATALOG")

(autoload 'sgml-mode "psgml" "Major mode to edit SGML files." t )
```

y añade una entrada en el mismo fichero para SGML en la definición existente para auto-mode-alist:

```
(setq
  auto-mode-alist
  '(("\\.sgml$" . sgml-mode)
  ))
```

Cada fichero fuente SGML tiene el siguiente bloque al final del fichero:

```
!- Mantenga este comentario al final del fichero
Local variables:
mode: sgml
sgml-omittag:t
sgml-shorttag:t
sgml-minimize-attributes:nil
sgml-always-quote-attributes:t
sgml-indent-step:1
sgml-indent-data:t
sgml-parent-document:nil
sgml-default-dtd-file:"./reference.ced"
sgml-exposed-tags:nil
sgml-local-catalogs:("/usr/lib/sgml/catalog")
sgml-local-ecat-files:nil
End:
-
```

La distribución de Postgres incluye un fichero DTD de definiciones, `reference.ced`.

Cuando esté usando `emacs/psgml`, una manera cómoda de trabajar con los ficheros separados de partes del libro es insertar una declaración DOCTYPE mientras los está editando. Si, por ejemplo, usted estuviera trabajando sobre este fichero fuente, que es un apéndice, usted especificaría que es una instancia "appendix" de un documento DocBook haciendo que la primera línea sea parecida a esto:

```
!doctype appendix PUBLIC "-//Davenport//DTD DocBook V3.0//EN"
```

Esa línea significa que cualquier cosa que sea capaz de leer SGML lo tomará correctamente y se puede verificar el documento con `"nsgmls -s docguide.sgml"`.

Haciendo documentaciones

El make de GNU se utiliza para compilar documentación desde fuentes DocBook. Hay algunas definiciones de entorno que quizás sea necesario fijar o modificar en su instalación. Makefile busca `doc/./src/Makefile` e implícitamente `doc/./src/Makefile.custom` para obtener información del entorno. En mi sistema el fichero `src/Makefile.custom` tiene este aspecto:

```
# Makefile.custom
# Thomas Lockhart 1998-03-01

PGRESDIR= /opt/postgres/current
CFLAGS+= -m486
YFLAGS+= -v

# documentation

HSTYLE= /home/lockhart/SGML/db143.d/docbook/html
PSTYLE= /home/lockhart/SGML/db143.d/docbook/print
```

donde HSTYLE y PSTYLE determinan el path a `docbook.dsl` para hojas de estilo HTML y copias impresas respectivamente. Estos ficheros de hojas de estilo son para el Modular Style Sheets, de Norm Walsh. Si se usan otras hojas de estilo, entonces se pueden definir HDSL y PDSL como el path completo y como nombre del fichero de la hoja de estilo como se hizo más arriba con HSTYLE y PSTYLE. En muchos sistemas estas hojas de estilo se encontrarán en en paquetes instalados en `/usr/lib/sgml/`, `/usr/share/lib/sgml/`, o `/usr/local/lib/sgml/`.

Los paquetes de documentación HTML pueden ser generados desde la fuente SGML escribiendo lo siguiente:

```
% cd doc/src
% make tutorial.tar.gz
% make user.tar.gz
% make admin.tar.gz
% make programmer.tar.gz
% make postgres.tar.gz
% make install
```

Estos paquetes pueden ser instalados desde el directorio principal de la documentación escribiendo:

```
% cd doc
% make install
```

Páginas man

Usamos la utilidad `docbook2man` para convertir las páginas REFENTRY de DocBook a salidas `*roff` adecuadas para páginas man. Hasta el momento de escribir esto, la utilidad requería un parche para ejecutar correctamente las marcas de Postgres y hemos añadido una pequeña cantidad de nuevas funciones que permiten configurar la sección de páginas man en el fichero de salida.

`docbook2man` está escrita en `perl` y requiere el paquete CPAN `SGMLSpm` para funcionar. También requiere `nsgmls`, que se incluye en la distribución de `jade`. Después de instalar estos paquetes, simplemente ejecute:

```
$ cd doc/src
$ make man
```

que generará un fichero `tar` en el directorio `doc/src`.

Proceso de instalación de `docbook2man`

1. Instale el paquete `docbook2man`, que se encuentra disponible en <http://shell.ipoline.com/~elmert/comp>
2. Instale el módulo `perl SGMLSpm`, disponible en las diferentes réplicas de CPAN.
3. Instale `nsgmls`, si todavía no lo tiene de instalación de `jade`.

Generación de copias impresas para v6.5

La documentación Postscript para copia impresa se genera convirtiendo la fuente SGML a RTF, para después importarla a ApplixWare-4.4.1. Después de pequeños cambios (vea la sección que viene) la salida se "imprime" a un fichero postscript.

Texto de copia impresa

`INSTALL` e `HISTORY` se actualizan en cada versión nueva. Por razones histórica estos ficheros están en un formato de texto plano, pero provienen de ficheros fuente SGML.

Generación de texto plano

Tanto `INSTALL` como `HISTORY` se generan desde fuentes SGML que ya existen. Se extraen desde el fichero RTF intermedio.

1. Para generar RTF escriba:


```
% cd doc/src/sgml
% make installation.rtf
```
2. Importe `installation.rtf` a Applix Words.
3. Fije el ancho y los márgenes de la página.
 - a. Ajuste el ancho de página en `File.PageSetup` a 10 pulgadas.
 - b. Seleccione todo el texto. Ajuste el margen derecho a 9,5 pulgadas utilizando la regla. Esto dará un ancho de columna máximo de 79 caracteres, que esta dentro del límite superior de las 80.
4. Elimine las partes del documento que no sean necesarias.

Para `INSTALL`, elimine todas las notas de la versión desde el final del texto a excepción de aquellas que tengan que ver con la versión actual. Para `HISTORY`, elimine todo el texto que esté por encima de las notas de versión, preservando y modificando el título y el ToC.
5. Exporte el resultado como "ASCII Layout".

6. Usando emacs o vi, elimine la información de tablas en `INSTALL`. Borre los “mailto” URLs de los que han contribuido al porte para comprimir la altura de las columnas.

Copia impresa postscript

Hay que abordar varias cosas mientras se genera en Postscript hardcopy.

Arreglos en Applixware RTF

No parece que Applixware haga bien el trabajo de importar RTF generado por jade/MSS. En particular, a todo el texto se le da la etiqueta de atributo “Header1”, aunque por lo demás el formateo del texto es aceptable. También se da que los números de página de la Tabla de Contenidos no referencian correctamente a la sección listada en la tabla, sino que se refieren directamente a la página de la Tabla de Contenidos.

1. Por ejemplo, para generar la entrada RTF escriba:

```
% cd doc/src/sgml
% make tutorial.rtf
```

2. Abra un nuevo documento en Applix Words y después importe el fichero RTF.
3. Imprima la Tabla de Contenidos que ya existe.
4. Inserte figuras en el documento y centre cada figura en la página utilizando el botón de centrado de márgenes.
No todos los documentos tienen figuras. Puede buscar en los ficheros fuente SGML la cadena “graphic” para identificar aquellas partes de la documentación que puedan tener figuras. Unas pocas figuras se repiten en varias partes de la documentación.
5. Trabaje sobre el documento ajustando saltos de página y anchos de columnas en las tablas.
6. Si existe bibliografía, Applix Words aparentemente marca todo el texto restante después del primer título como si tuviera un atributo de subrayado. Seleccione todo el texto restante y desactive el subrayado usando el botón de subrayado y a partir de aquí subraye explícitamente cada documento y el título del libro.
7. Siga tratando el documento marcando la Tabla de Contenidos con el correspondiente número de página para cada entrada.
8. Reemplace los valores de número de página de la derecha en la Tabla de Contenidos con los valores correctos. Esto sólo toma unos minutos por documento.
9. Guarde el documento en el formato nativo de Applix Words para permitir una edición rápida más adelante si fuera necesario.
10. “Imprima” el documento a un fichero en formato Postscript.
11. Comprima el fichero Postscript utilizando `gzip`. Coloque el fichero comprimido en el directorio `doc`.

Herramientas

Hemos documentado nuestra experiencia con tres métodos de instalación para las diferentes herramientas que son necesarias para procesar la documentación. Una es la instalación desde RPMs en Linux, la segunda es la instalación desde el *porte* a FreeBSD y la última es una instalación general desde distribuciones originales de las herramientas. Estas se describirán más abajo.

Pueden existir otras distribuciones empaquetadas de estas herramientas. Por favor remita información sobre estado de los paquetes a las listas de correo y la incluiremos aquí.

Instalación de RPM Linux

La instalación más sencilla para sistemas Linux compatibles con RedHat utiliza RPM, desarrollado por Mark Galassi de Cygnus. También es posible instalar desde las fuentes, como se describe en secciones posteriores.

Instalando RPMs

1. Instale los RPM ⁸ para Jade y los paquetes relacionados.
2. Instale las últimas hojas de estilo de Norm Walsh. Dependiendo de la antigüedad de los RPM, las últimas hojas de estilo pueden haber sido muy mejoradas respecto a aquellas que aparecen con los RPM.
3. Actualice su `src/Makefile.custom` para que incluyan las definiciones de HSTYLE y de PSTYLE que apuntan a las hojas de estilo.

Instalación en FreeBSD

Hay un gran conjunto de *portes* de la documentación de las herramientas disponibles para FreeBSD. De hecho, postgresql.org, en el que la documentación se actualiza automáticamente cada tarde, es una máquina con FreeBSD.

Instalando los "portes" de FreeBSD

1. Para compilar la documentación sobre FreeBSD se necesita instalar unos cuantos "portes".

```
% cd /usr/ports/devel/gmake && make install
% cd /usr/ports/textproc/docproj && make install
% cd /usr/ports/textproc/docbook && make install
% cd /usr/ports/textproc/dsssl-docbook-modular && make install
```

2. Fijar las variables de entorno para acceder al conjunto de herramientas de jade.

Nota: Esto no era requerido para la máquina FreeBSD de postgresql.org, así que puede que esto no sea necesario.

```
export SMGL_ROOT=/usr/local/share/sgml
SGML_CATALOG_FILES=/usr/local/share/sgml/jade/catalog
SGML_CATALOG_FILES=/usr/local/share/sgml/html/catalog:$SGML_CATALOG_FILES
```

```
SGML_CATALOG_FILES=/usr/local/share/sgml/iso8879/catalog:$SGML_CATALOG_FILES
SGML_CATALOG_FILES=/usr/local/share/sgml/transpec/catalog:$SGML_CATALOG_FILES
SGML_CATALOG_FILES=/usr/local/share/sgml/docbook/catalog:$SGML_CATALOG_FILES
export SGML_CATALOG_FILES
```

(esto es para sintaxis sh/bash. Ajústelo para csh/tcsh).

3. Make necesita algunos argumentos especiales o estos han de ser añadidos a su Makefile.custom:

```
HSTYLE=/usr/local/share/sgml/docbook/dsssl/modular/html/
PSTYLE=/usr/local/share/sgml/docbook/dsssl/modular/print/
```

Por descontado que necesitará usar gmake, no sólo 'make', para compilar.

Instalación en Debian

Hay un juego completo de paquetes de la documentación de las herramientas disponible para Debian.

Instalando los paquetes Debian

1. Instale jade, docbook y unzip:

```
apt-get install jade
apt-get install docbook
apt-get install docbook-stylesheets
```

2. Instale las últimas hojas de estilo.

- a. Verifique que unzip está instalado o instale el paquete:

```
apt-get install unzip
```

- b. Consiga el fichero comprimido con las últimas hojas de estilo en <http://www.nwalsh.com/docbook> y descomprímalo (quizás en /usr/share).

3. Edite src/Makefile.custom y añádale las definiciones de HSTYLE y PSTYLE adecuadas:

```
HSTYLE= /usr/share/docbook/html
PSTYLE= /usr/share/docbook/print
```

Instalación manual de las herramientas

Esta es una breve descripción del proceso de obtener e instalar el software que necesitará para editar la fuente DocBook con Emacs y tratarla con las hojas de estilo DSSSL de Norman Walsh par crear ficheros HTML y RTF.

La manera más fácil de obtener las herramientas SGML y DocBook quizás sea tomar sgmltools desde sgmltools¹⁰. sgmltools necesita la versión GNU de m4. Para confirmar que tiene la versión correcta de m4 pruebe

```
gnum4 -version
```

Si instala GNU m4, instálelo con el nombre gnum4 y sgmltools lo encontrará. Después de la instalación usted tendrá sgmltools, jade y las hojas de estilo DocBook de Norman Walsh. Las instrucciones de abajo son para instalar estas herramientas de modo separado.

Requisitos previos

Lo que usted necesita es:

- Una instalación funcionando de GCC 2.7.2
- Una instalación trabajando de Emacs 19.19 o posterior
- La utilidad unzip para descomprimir ficheros

Debe conseguir:

- Jade de James Clark¹¹ (la versión 1.1 en el fichero jade1_1.zip en el momento de escribir esto)
- DocBook versión 3.0¹²
- Modular Stylesheets¹³ de Norman Walsh (la versión 1.19 fue originalmente usada para producir estos documentos)
- PSGML¹⁴ de Lennar Staflin (la versión 1.0.1 en psgml-1.0.1.tar.gz era la que estaba disponible en el momento de escribir esto)

URLs importantes:

- La web de Jade¹⁵
- La página web de DocBook¹⁶
- La web de Modular Stylesheets¹⁷
- Web de PSGML¹⁸
- La guía de Steve Pepper¹⁹
- Base de datos de Robin Cover sobre software SGML²⁰

Instalación de Jade

Instalación de Jade

1. Lea las instrucciones de instalación en la URL mostrada arriba.
2. Descomprima la distribución con unzip en el lugar adecuado . El comando para para hacer esto puede ser como este:

```
unzip -aU jade1_1.zip
```

3. Jade no ha sido construido usando GNU autoconf, de modo que tendrá que editar un `Makefile` por su cuenta. Ya que James Clark ha sido lo suficientemente bueno como para preparar su kit para ello, es una buena idea crear un directorio (con un nombre como la arquitectura de su máquina, por ejemplo) bajo el directorio principal de Jade, copiar desde él el fichero `Makefile` al directorio recién creado, editarlo y desde ahí mismo ejecutar **make**.

`Makefile` necesitar ser. Hay un fichero llamado `Makefile.jade` en el directorio principal cuyo cometido es ser usado con **make -f Makefile.jade** cuando se construye Jade (a diferencia de SP, el parser SGML sobre el que está construido Jade). Aconsejamos que no se haga esto, ya que deberá cambiar más cosas que lo que hay en `Makefile.jade`.

Recorra el fichero `Makefile`, leyendo las instrucciones de Jame y haciendo los cambios necesarios. Hay algunas variables que necesitan ser fijadas. Aquí se muestra un sumario de las más representativas con sus valores más típicos:

```
prefix = /usr/local
XDEFINES = -DSGML_CATALOG_FILES_DEFAULT=\"/usr/local/share/sgml/catalog\"
XLIBS = -lm
RANLIB = ranlib
srcdir = ..
XLIBDIRS = grove spgrove style
XPROGDIRS = jade
```

Observe la especificación de dónde encontrar el catálogo SGML por defecto de los ficheros de soporte (quizás tenga que cambiarlo a algo más adecuado para su instalación). Si su sistema no necesita los ajustes mencionados arriba para la librería de funciones matemáticas y para el comando **ranlib**, déjelos tal y como están en `Makefile`.

4. Escriba **make** para compilar Jade y las herramientas de SP.
5. Una vez que esté compilado, **make install** hará la instalación.

Instalación del DTD de DocBook

Instalación del DTD de DocBook

1. Es conveniente que emplace los ficheros que constituyen el DTD de DocBook en el directorio en el que compiló Jade, `/usr/local/share/sgml/` si ha seguido nuestra recomendación. Además de los ficheros de DocBook, necesitará el fichero `catalog` en su sitio para el mapa de las especificaciones del tipo de documento y las referencias externas de las entidades a los ficheros actuales en ese directorio. También necesitará el mapa de caracteres ISO y posiblemente una o más versiones de HTML.

Una manera para instalar las diferentes DTD y ficheros de soporte y para ajustar el fichero `catalog` es juntarlos todos en el directorio mencionado más arriba, usar un único fichero llamado `CATALOG` que los describa a todos y entonces crear el fichero `catalog` como un puntero a `catalog` añadiendo la línea:

```
CATALOG /usr/local/share/sgml/CATALOG
```

2. El fichero CATALOG contendría tres tipos de líneas. La primera (opcional) la declaración SGML :

```
SGMLDECL docbook.dcl
```

Después, las diferentes referencias a DTD y ficheros de las entidades. Para los ficheros DocBook las líneas serían como estas:

```
PUBLIC "-//Davenport//DTD DocBook V3.0//EN" docbook.dtd
PUBLIC "-//USA-DOD//DTD Table Model 951010//EN" cals-tbl.dtd
PUBLIC "-//Davenport//ELEMENTS DocBook Information Pool V3.0//EN" dbpool.mod
PUBLIC "-//Davenport//ELEMENTS DocBook Document Hierarchy V3.0//EN" dbhier.mod
PUBLIC "-//Davenport//ENTITIES DocBook Additional General Entities V3.0//EN" dbgenent
```

3. Por supuesto que en el kit de DocBook hay un fichero que contiene todo esto. Observe que el último elemento en cada una de esas líneas es un nombre de fichero, que aquí se da sin el path. Puede poner los ficheros en subdirectorios de su directorio SGML si quiere y modificar la referencia en el fichero CATALOG. DocBook también referencia el conjunto de caracteres ISO de las entidades, por lo que necesitará traerlos e instalarlos (están disponibles desde varias fuentes y se pueden encontrar fácilmente en las URLs mostradas más arriba), además de su entradas:

```
PUBLIC "ISO 8879-1986//ENTITIES Added Latin 1//EN" ISO/ISOlat1
```

Observe que el nombre de fichero contiene un directorio, diciéndonos que hemos puesto los ficheros de entidades ISO en un subdirectorio ISO. Nuevamente las entradas oportunas en el catálogo deben acompañar a la entidad que se haya traído.

Instalación de las hojas de estilo DSSSL de Norman Walsh

Instalación de las hojas de estilo DSSSL de Norman Walsh

1. Lea las instrucciones de instalación en la URL mostrada más arriba.
2. Para instalar las hojas de estilo de Norman, simplemente descomprima los ficheros de la distribución en el lugar adecuado. Un buen lugar para hacerlo sería `/usr/local/share`, que emplaza los los ficheros en un directorio bajo `/usr/local/share/docbook`. El comando sería algo parecido a esto:

```
unzip -aU db119.zip
```

3. Una manera de probar la instalación es compilar los formularios HTML y RTF de la Guía de usuarios de *PostgreSQL*.
 - a. Para compilar los ficheros HTML vaya al directorio fuente de SGML , `doc/src/sgml`, y escriba

```
jade -t sgml -d /usr/local/share/docbook/html/docbook.dsl -D ../graphics postgres.sgml
```

`book1.htm` es el nodo más alto de la salida...

- b. Para generar el RTF preparado ser importado a su procesador de textos favorito, escriba:

```
jade -t rtf -d /usr/local/share/docbook/print/docbook.dsl -D ../graphics postgres.sgml
```

Instalación de PSGML

Instalación de PSGML

1. Lea las instrucciones de instalación en la URL mostrada más arriba.
2. Desempaque el fichero de la distribución, ejecute configure, make y make install para colocar en su sitio los ficheros compilados y las librerías.
3. Después añada las líneas siguientes al fichero `/usr/local/share/emacs/site-lisp/site-start.el` de modo que Emacs pueda cargar correctamente PSGML cuando lo necesite:

```
(setq load-path
      (cons "/usr/local/share/emacs/site-lisp/psgml" load-path))
(auto-load 'sgml-mode "psgml" "Major mode to edit SGML files." t)
```

4. Si necesita usar PSGML cuando también esté editando HTML añada esto:

```
(setq auto-mode-alist
      (cons '("\\.s?html?\\'" . sgml-mode) auto-mode-alist))
```

5. Hay una cosa importante que debe tener en cuenta con PSGML: su autor asume que su directorio principal para el DTD SGML es `/usr/local/lib/sgml`. Si, como en los ejemplos de capítulo, utiliza `/usr/local/share/sgml`, debe corregirlo adecuadamente.
 - a. Puede fijar la variable de entorno `SGML_CATALOG_FILES`.
 - b. Puede personalizar su instalación de PSGML (el manual le dirá cómo).
 - c. Puede incluso editar el fichero fuente `psgml.el` antes de compilar e instalar PSGML, de modo que cambie los path para que se adecuen a los suyos por defecto.

Instalación de JadeTeX

Si quiere, también puede instalar JadeTeX para usar TeX como utilidad para formatear Jade. Tenga en cuenta que es todavía un software sin depurar y generará salidas impresas inferiores a las obtenidas desde RTF. A pesar de todo, funciona bien, especialmente para los documentos más simples que no usan tablas y además, como JadeTeX y las hojas de estilo, está en un proceso continuo de mejora a medida que pasa el tiempo.

Para instalar y utilizar JadeTeX necesitará que TeX y LaTeX2e estén funcionando correctamente, incluyendo los paquetes `tools` y `graphics`, Babel, AMS fonts y AMS-LaTeX, PSNFSS y el kit de las 35 fuentes, dvips para generar PostScript, los paquetes de macros `fancyhdr`, `hyperref`, `minitoc`, `url` y `ot2enc` y por supuesto JadeTeX. Todos ellos se pueden encontrar en el site CTAN más próximo.

JadeTeX, en el momento de escribir esto, no viene con una guía de instalación, pero hay fichero `makefile` que muestra qué es necesario. También incluye un directorio llamado `cooked` donde encontrará algunos de los paquetes de macro que necesita (aunque no todos y tampoco completos).

Antes de compilar el fichero de formato `jadetex.fmt`, es posible que quiera editar el fichero `jadetex.ltx` para cambiar la configuración de Babel para ajustarla a su instalación. La línea a cambiar se parece a esta:

```
\RequirePackage[german,french,english]{babel}[1997/01/23]
```

y obviamente debe poner sólo los idiomas que necesite y configurar Babel para ello.

Con JadeTeX en funcionamiento, debería poder generar y formatear las salidas de TeX para los manuales de PostgreSQL pasando los comandos (como más arriba, en el directorio `doc/src/sgml`)

```
jade -t tex -d /usr/local/share/docbook/print/docbook.dsl -D ../graphics postgres.sgml
jadetex postgres.tex
jadetex postgres.tex
dvips postgres.dvi
```

Por supuesto, cuando haga esto TeX parará durante la segunda ejecución diciendo que su capacidad se ha sobrepasado. Esto es debido al modo en que JadeTeX genera información de referencias cruzadas. TeX puede ser compilado de manera que utilice estructuras de datos mayores. Los detalles de esto variarán de acuerdo a su instalación.

Otras herramientas

`sgml-tools v2.x` diseñada para `jade` y DocBook.

Notas

1. <http://nis-www.lanl.gov/~rosalia/mydocs/docbook-intro.html>
2. <http://www.ora.com/homepages/dtdparse/docbook/3.0/>
3. <http://www.jclark.com/jade/>
4. <http://www.nwalsh.com/docbook/dsssl/>
5. <http://sunsite.unc.edu/pub/packages/TeX/systems/unix/>
6. <http://www.ora.com/davenport/>
7. <http://shell.ipoline.com/~elmer/comp/docbook2X/>
8. <ftp://ftp.cygnum.com/pub/home/rosalia/>
9. <http://www.nwalsh.com/docbook/dsssl>

10. <http://www.sgmltools.org/>
11. <ftp://ftp.jclark.com/pub/jade/>
12. <http://www.ora.com/davenport/docbook/current/docbk30.zip>
13. <http://nwalsh.com/docbook/dsssl/>
14. <ftp://ftp.lysator.liu.se/pub/sgml/>
15. <http://www.jclark.com/jade/>
16. <http://www.ora.com/davenport/>
17. <http://nwalsh.com/docbook/dsssl/>
18. http://www.lysator.liu.se/projects/about_psgml.html
19. <http://www.infotek.no/sgmltool/guide.htm>
20. <http://www.sil.org/sgml/publicSW.html>

Bibliografía

Selección de referencias y lecturas sobre SQL y Postgres.

Libros de referencia sobre SQL

Documentación específica sobre PostgreSQL

Procedimientos y Artículos