Guia del Programador de PostgreSQL

El equipo de desarrollo de PostgreSQL

Editado por

Thomas Lockhart

Guia del Programador de PostgreSQL

por El equipo de desarrollo de PostgreSQL

Editado por Thomas Lockhart

Tabla de contenidos

Lista de tablas

Tabla de figuras

Tabla de ejemplos

Sumario

Postgres, desarrollada originalmente en el Departamento de Ciencias de la Computación de la Universidad de California en Berkeley, fué pionera en muchos de los conceptos de bases de datos relacionales orientadas a objetos que ahora empiezan a estar disponiblesen algunas bases de datos comerciales. Ofrece suporte al lenguage SQL92/SQL3, integridad de transacciones, y extensibilidad de tipos de datos. PostgreSQL es un descendiente de dominio público y código abierto del código original de Berkeley.

Capítulo 1. Introducción

Este documento es el manual del programador para el gestor de bases de datos PostgreSQL (http://postgresql.org/), desarrollado inicialmente en la Universidad de California en Berkeley. PostgreSQL se basa en Postgres versión 4.2 (http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html). El proyecto Postgres, liderado por el Profesor Michael Stonebraker, ha sido financiado por la Agencia de Proyectos de Investigación de Defensa Avanzados (DARPA), la Oficina de Investigación del Ejército (ARO), la Fundación Nacional de Ciencia (NSF), y ESL, Inc.

La primera parte de este manual explica el enfoque extensible de Postgres y describe como Postgres puede ser ampliado por los usuarios añadiendo sus propios tipos, operadores, agregados y funciones, tanto en lenguaje de programación como en lenguaje de bases de datos. Después de una discusión del sistema de reglas de Postgres, discutimos las interfaces de disparo (trigger) y SPI. El manual concluye con una descripción detallada de la interfaz de programación y de las librerías de apoyo para varios lenguajes.

Asumimos que el lector tiene buenos conocimientos de Unix y de programación en C.

1.1. Recursos

Este manual está organizado en diferentes partes:

Tutorial

Introduccion para nuevos usuarios. No cubre características avanzadas.

Guia del Usuario

Informacion 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, libreria de interfaces y lo referido al diseño de aplicaciones.

Guia del Administrador

Información sobre instalación y administración. Lista de equipo soportado.

Guia del Desarrollador

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

Manual de Referencia

Información detallada sobre los comandos. Actualmente incluído en la *Guia del Usuario*.

Ademas de éste manual, hay otros recursos que le servirán de ayuda para la instalacion y el uso de Postgres:

man pages

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

FAQs(Preguntas Frequentes)

La sección de Preguntas Frequentes(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) estan disponibles para algunas contribuciones.

Web Site

El sitio web de Postgres (postgresql.org) 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 (mailto:pgsql-general@postgresql.org) (archive (http://www.PostgreSQL.ORG/mhonarc/pgsql-general/)) 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 empieze 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 (mailto:pgsql-docs@postgresql.org) (archivo (http://www.PostgreSQL.ORG/mhonarc/pgsql-docs/)) de la lista de correos es un buen lugar para comenzar sus pesquisas.

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

1.3. 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) (mailto:docs@postgresql.org).

1.4. Y2K Statement (Informe sobre el efecto 2000)

> **Autor:** Escrito por Thomas Lockhart (mailto:lockhart@alumni.caltech.edu) 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 Usuario (http://www.postgresql.org/docs/user/datatype.htm) 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 Project (http://www.gnu.org/software/year2000.html) y a The Perl Institute (http://language.perl.com/news/y2k.html) 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.

1.5. 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. .

Capítulo 2. Arquitectura

2.1. 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 2-1. 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 3. Extensor SQL: Preludio

En la seccion que sigue, trataremos como añadir extensiones al Postgres SQL usando peticiones del lenguaje:

- funciones
- tipos
- operadores
- añadidos

3.1. 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 bilioteca. 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 sitemas de base de datos pueden ser extendidos solamante 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 incoporar 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 habilitad para modificar la operacion ’al vuelo’ hace de Postgres la unica suite para prototipos rapidos de nuevas aplicaciones y estructuras de almacenamiento.

3.2. 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.

3.3. Acerca de los Sistema de Catalogo de Postgres

Para introducirnos en los conceptos basicos de la extensibilidad, hemos de estudiar como se dise¤an los catalogos. Puede saltarse esta seccion ahora, pero algunas secciones mas tarde no seran comprendidas sin la informacion dada aqui, asi que marque esta paágina 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 3-1. Sistema de Catalogos de Postgres

Nombre del Catalogo	Descriccion
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 3-1. 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 caracteristicas 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 complrender, y seran descritas en profundidad (en la seccion sobre tipos de interfase y operadores de indice) antes de que estudiemos las extensiones basicas.

Capítulo 4. 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 funcón 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`.

4.1. Funciones de Lenguaje de Consultas (SQL)

Las funciones SQL ejecutan una lista arbitraria de consultas SQL, devolviendo los resultados de la última consulta de la lista. Las funciones SQL en general devuelven conjuntos. Si su tipo de retorno no se especifica como un `setof`, entonces un elemento arbitrario del resultado de la última consulta será devuelto.

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.

4.1.1. 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’;
```

4.1.2. 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 sobreescribió la lista objetivo de la función. Por esto, el resultado se etiqueta answer en vez de one.

Es casi tan fácil definir funciones SQL que tomen tipos base como argumentos. En el ejemplo de abajo, note cómo nos referimos a los argumentos dentro de la función como $1 y $2:

```
CREATE FUNCTION add_em(int4, int4) RETURNS int4
 AS 'SELECT $1 + $2;' LANGUAGE 'sql';

SELECT add_em(1, 2) AS answer;

     +-----+
     |answer |
     +-----+
```

```
|3      |
+-----+
```

4.1.3. 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 |
     +-+
```

4.2. Funciones de Lenguaje Procedural

Los lenguajes procedurales no están construidos 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.

4.3. 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.

4.4. 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.)

4.4.1. 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 4-1. 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

Built-In Type	C Type	Defined In
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 funciones 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 *) palloc(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 *) palloc(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 *) palloc(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).

4.4.2. 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 GetAttributeByNa-
me() */

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 discutirlas en este manual.

4.4.3. 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.

4.5. 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.

4.5.1. 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.

4.5.1.1. 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 este restricción: mientras las rutinas de carga dinámicas en la mayoría de los sistemas operativos están mas 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 státicamente y declararlas como funciones INTERNAL. Entonces, las funciones deben tener todas 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 envolvente) 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 5. 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`.

5.1. Tipos Definidos por el Usuario

5.1.1. 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.

5.1.2. 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 6. Extenediendo 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 "azucar 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 azucar sintáctico, porque carga información adicional que ayuda al planeador de consultas a obtimizar 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 tragajo; 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 procedimietno y las sentencias del argumneto 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.

6.1. 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 proveidas 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 descríptas aquí son todas aquellas que la versión 6.5 entinede.

6.1.1. COMMUTATOR (conmutador)

La sentencia COMMUTATOR ,si es proveida, 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 A y) es igual (y 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 proveido 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 qeu 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.)

6.1.2. NEGATOR(negador)

La sentencia NEGATOR, si es proveida, nombra a un operador que es el negador del operador que está siendo definido. Nosotros decimos que un operador A es el negdor de un operador B si ambos retornan resultados bouleanos y (x A y) no es igual (x 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 CONMMUTATOR, un par de operadores unarios, pueden ser validamente marcados como negadores entre si; eso significaría (A x) no es igual (B 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.

6.1.3. RESTRICT (Restringir)

La sentencia RESTRICT, si es proveida, 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 bouleanos. La idea detrás de un estimador selectivo de restricción es suponer que fracción de las filas en una tabla satisfacerán 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 particualr. Esto asiste al optimizador dandole 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 CONMMUTATOR...)

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 <>
scalarltsel for < or <=
scalargtsel for > or >=
```

Puede parecer un poco curioso que estas son las categorias, pero ellas tienen sentido si tu piensas acerca de ellas. ’=’ tipicamente aceptará solamente una fracción pequeña de las filas en la tabla; ’<>’ tipicamente rechazará solamente una fracción pequeña. ’<’

aceptara una fracción que depende de donde las constantes dadas quedan en el rango de valores para es 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 mas laraga que ’<’ para las mismas constantes comparadas, pero son lo suficientemente cerradas para no ser costosas, especialmente desde que nosotros no estamos probalemente 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 ejemplok, la expresión regular operadores emparejados (~,~*, etc.) usa eqsel sobre la suposición que ellos usualmente solo emparejen una pequeña fracciónde entradas en una tabla.

Tu puedes usar scalarltsel y scalargtsel para comparaciones sobre tipos de datos que tienen cierto significado sencible 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 travéz de una columna de la tabla pg_type; pero eso no ha pasado todavía.)Si tu no haces esto, las cosas seguiran funcionando, pero la estimación del optimizador no estarán tan bien como podrian.

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 mejormejorarlas).

6.1.4. JOIN (unir)

La sentencia JOIN, si es proveida, 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 bouleanos. La idea detrás de un estimador selectivo join es suponer que fracción de las filas de un par de tablas satisfacerán 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 permitiendole 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 estandars si alguna es aplicable:

```
eqjoinsel for =
neqjoinsel for <>
scalarltjoinsel for < or <=
scalargtjoinsel for > or >=
areajoinsel for 2D area-based comparisons
positionjoinsel for 2D position-based comparisons
contjoinsel for 2D containment-based comparisons
```

6.1.5. HASHES(desmenusamiento)

La sentencia HASHES, si está presente, le dice al sistema que está bien usar un metodo 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 selamente 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, implicitamente asumiendo que el resultado del operadior 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 sera 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, todoslos pares que el operador de igualdad dija 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 interes, es inseguro marcar el operador de iguladad HASHES. (al menos, quizas, tu escribes tu otro operador para asegurarte que los bits sin uso son iguales a zero). Otro ejemplo es que los tipo 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 mas 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 mnos cero y un mas cero probablemente no serían igualados por hash join, pero ellos serían igualados por cualquier otro proceso join.

La última linea es para la cual tu probablemente deberías usar unicamente HASEHES para igualdad de operadores que son (o podría ser) implementada por memcmp().

6.1.6. SORT1 and SORT2 (orden1 y orden2)

La sentencia SORT, si está presente, le dice al sistema que esta 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 tipo 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 ídea de ordenar las tablas izquierdas y derechas en un orden y luego inspecionarlas 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 busqueda de orden. En práctica esto significa que el operador join debe comportarse como iguldad. Pero distinto de hashjoin, cuando los tipos de datos izquierdos y derechos tuvieron qe ser mejor el mismo. (o al menos iquivalentes bit a bit), es posible unir dos tipos de datos distintos tanto como sean ellos compatibles logicamente. Por ejemplo, el operador de igualdad int2-versus-int4 es unible. Solo necesitamos operadores de oprden que traigna 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 bouleanos; el operador SORT1 debe tener ambos tipo 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 tu definiste el operador de igualda antes que los otros.

En práctica tu debes solo escribier sentencias SORT para un operador ’=’, y los dos operadores referenciados deben ser siempre nombrados ’<’. Tratando de usar merge join con operadores nombrados nada mas resultará en confusiones inesperadas, por razones que veremos en un momento.

Hay restricciones adicionales sobre operadores que tu marcas mergegoinables. 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 operados mergejinable en si mismo. Estos operadores *deben*ser nombrados ’<’ and ’>’; tu no tienes opcion en este problema, desde que no hay provición para especificarlos explicitamente. Nota que si los tipo de datos izquierdo y derechos son diferentes, ninguno de estos operadors es el mismo que cualquier operador SORT. pero ellos tuvieron mejores ordenados la compativilidad de los valores de dato con los operadores SORT, o o mergejoin fallará al funcionar.

Capítulo 7. 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 funcion 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 8. 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*].

8.1. ¿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 situa 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 repesentable 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 catalgo 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.

8.1.1. 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 arbol 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 identica 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 clausula VALUES y las de la clausula 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 expresion 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 arbol 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 e 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 clausula WHERE de una instrucción SQL.

the others

Las otras partes de un arbol de query, como la clausula ORDER BY, no tienen interés aquí. El sistema de reglas sustituye las entradas aquí presentes mientras está aplicando las reglas, pero aquellas no tiene 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.

8.2. Las vistas y el sistema de reglas.

8.2.1. 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.

8.2.2. Cómo trabajan las reglas de SELECT

Las reglas ON SELECT se aplican a todas las queries como el último paso, incluso si el comando dado es INSERT, UPDATE o DELETE. Y tienen diferentes semanticas de las otras en las que modifican el arbol de traducción en lugar de crear uno nuevo. Por ello, las reglas SELECT se describen las primeras.

Actualmente, debe haber sólo una acción y debe ser una acción SELECT que es una INSTEAD. Esta restricción se requería para hacer las reglas seguras contra la apertura por usuarios ordinarios, y restringe las reglas ON SELECT a reglas para vistas reales.

El ejemplo para este documento son dos vistas unidas que hacen algunos cálculos y algunas otras vistas utilizadas para ello. Una de estas dos primeras vistas se personaliza más tarde añadiendo reglas para operaciones de INSERT, UPDATE y DELETE de modo que el resultado final será una vista que se comporta como una tabla real con algunas funcionalidades mágicas. No es un ejemplo fácil para empezar, y quizá sea demasiado duro. Pero es mejor tener un ejemplo que cubra todos los puntos discutidos paso a paso que tener muchos ejemplos diferentes que tener que mezclar después.

La base de datos necesitada para ejecutar los ejemplos se llama al_bundy. Verá pronto el porqué de este nombre. Y necesita tener instalado el lenguaje procedural PL/pgSQL, ya que necesitaremos una pequeña función min() que devuelva el menor de dos valores enteros. Creamos esta función como:

```
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 facil 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 interes para las reglas de SELECT.

Ahora publicamos `unit`, `shoe_data` y `shoelace_data` y Al (el propietario de al_bundy) teclea su primera SELECT en esta vida.

```
al_bundy=> INSERT INTO unit VALUES (’cm’, 1.0);
al_bundy=> INSERT INTO unit VALUES (’m’, 100.0);
al_bundy=> INSERT INTO unit VALUES (’inch’, 2.54);
al_bundy=>
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     (’sh1’, 2, ’black’, 70.0, 90.0, ’cm’);
```

```
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     (’sh2’, 0, ’black’, 30.0, 40.0, ’inch’);
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     (’sh3’, 4, ’brown’, 50.0, 65.0, ’cm’);
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     (’sh4’, 3, ’brown’, 40.0, 50.0, ’inch’);
al_bundy=>
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     (’sl1’, 5, ’black’, 80.0, ’cm’);
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     (’sl2’, 6, ’black’, 100.0, ’cm’);
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     (’sl3’, 0, ’black’, 35.0 , ’inch’);
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     (’sl4’, 8, ’black’, 40.0 , ’inch’);
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     (’sl5’, 4, ’brown’, 1.0 , ’m’);
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     (’sl6’, 0, ’brown’, 0.9 , ’m’);
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     (’sl7’, 7, ’brown’, 60 , ’cm’);
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     (’sl8’, 1, ’brown’, 40 , ’inch’);
al_bundy=>
al_bundy=> SELECT * FROM shoelace;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
-------+-----+-------+----+-----+------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       7|brown     |    60|cm      |       60
sl3       |       0|black     |    35|inch    |     88.9
sl4       |       8|black     |    40|inch    |    101.6
sl8       |       1|brown     |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
sl6       |       0|brown     |   0.9|m       |       90
(8 rows)
```

Esta es la SELECT más sencilla que Al puede hacer en sus vistas, de modo que nosotros la tomaremos para explicar la base de las reglas de las vistas. ’SELECT * FROM shoelace’ fue interpretado por el traductor y produjo un árbol de traducción.

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace;
```

y este se le dá al sistema de reglas. El sistema de reglas viaja a través de la tabla de rango, y comprueba si hay reglas en `pg_rewrite` para alguna relación. Cuando se procesa las entradas en la tabla de rango para `shoelace` (el único hasta ahora) encuentra la regla ’_RETshoelace’ con el árbol de traducción

```
SELECT s.sl_name, s.sl_avail,
       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 arbol de traducción, que se refieren a entradas de la tabla de rango (la única que se está procesando en este momento para `shoelace`) por las correspondientes expresiones de la lista objetivo correspondiente a la acción de las reglas. El resultado es la query final:

```
 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 el número 2 (shoelace *OLD*). La Relación `shoelace` tiene una regla, pero su entrada en la tabla de rangos no está referenciada en ninguna de las variables del árbol de traducción, de modo que se ingnora. Puesto que todas las entradas restantes en la tabla de rango, o bien no tienen reglas en `pg_rewrite` o bien no han sido referenciadas, se alcanza el final de la tabla de rango.

La reescritura está completa y el resultado final dado se pasa al optimizador. El optimizador ignora las entradas extra en la tabla de rango que no están referenciadas por variables en el árbol de traducción, y el plan producido por el planificador/optimizador debería ser exactamente el mismo que si Al hubiese tecleado la SELECT anterior en lugar de la selección de la vista.

Ahora enfrentamos a Al al problema de que los Blues Brothers aparecen en su tienda y quieren comprarse zapatos nuevos, y como son los Blues Brothers, quieren llevar los mismos zapatos. Y los quieren llevar inmediatamente, de modo que necesitan también cordones.

Al necesita conocer los zapatos para los que tiene en el almacén cordones en este momento (en color y en tamaño), y además para los que tenga un número igual o superior a 2. Nosotros le enseñamos a realizar la consulta a su base de datos:

```
al_bundy=> SELECT * FROM shoe_ready WHERE total_avail >= 2;
shoename  |sh_avail|sl_name   |sl_avail|total_avail
-------+-----+-------+-----+-------
sh1       |       2|sl1       |       5|          2
sh3       |       4|sl7       |       7|          4
(2 rows)
```

Al es un guru de los zapatos, y sabe que sólo los zapatos de tipo sh1 le sirven (los cordones sl7 son marrones, y los zapatos que necesitan cordones marrones no son los más adecuados para los Blues Brothers).

La salida del traductor es esta vez el arbol de traducción.

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM shoe_ready shoe_ready
 WHERE int4ge(shoe_ready.total_avail, 2);
```

Esa será la primera regla aplicada para la relación `shoe_ready` y da como resultado el árbol de traducción

```
SELECT rsh.shoename,
```

```
        rsh.sh_avail,
        rsl.sl_name,
rsl.sl_avail,
        min(rsh.sh_avail, rsl.sl_avail) AS
        total_avail
   FROM shoe_ready shoe_ready, shoe_ready *OLD*,
        shoe_ready *NEW*,
shoe rsh,
        shoelace rsl
  WHERE int4ge(min(rsh.sh_avail, rsl.sl_avail), 2)
    AND (bpchareq(rsl.sl_color, rsh.slcolor)
         AND float8ge(rsl.sl_len_cm, rsh.slminlen_cm)
         AND float8le(rsl.sl_len_cm, rsh.slmaxlen_cm)
        );
```

En realidad, la clausula AND en la cualificación será un nodo de operadores de tipo AND, con una expresión a la izquierda y otra a la derecha. Pero eso la hace menos legible de lo que ya es, y hay más reglas para aplicar. De modo que sólo las mostramos entre paréntesis para agruparlos en unidades lógicas en el orden en que se añaden, y continuamos con las reglas para la relación shoe como está en la entrada de la tabla de rango a la que se refiere, y tiene una regla. El resultado de aplicarlo es

```
 SELECT sh.shoename,
        sh.sh_avail,
        rsl.sl_name, rsl.sl_avail,
        min(sh.sh_avail, rsl.sl_avail)
        AS total_avail,
   FROM shoe_ready shoe_ready, shoe_ready *OLD*,
        shoe_ready *NEW*, shoe rsh,
        shoelace rsl, shoe *OLD*,
        shoe *NEW*,
shoe_data sh,
        unit un
  WHERE (int4ge(min(sh.sh_avail, rsl.sl_avail), 2)
         AND (bpchareq(rsl.sl_color, sh.slcolor)
              AND float8ge(rsl.sl_len_cm,
                 float8mul(sh.slminlen, un.un_fact))
```

```
                AND float8le(rsl.sl_len_cm,
                   float8mul(sh.slmaxlen, un.un_fact))
               )
          )
      AND bpchareq(sh.slunit, un.un_name);
```

Y finalmente aplicamos la regla para `shoelace` que ya conocemos bien (esta vez en un arbol de traducción que es un poco más complicado) y obtenemos

```
    SELECT sh.shoename, sh.sh_avail,
           s.sl_name, s.sl_avail,
           min(sh.sh_avail, s.sl_avail) AS total_avail
      FROM shoe_ready shoe_ready, shoe_ready *OLD*,
           shoe_ready *NEW*, shoe rsh,
           shoelace rsl, shoe *OLD*,
           shoe *NEW*, shoe_data sh,
           unit un, shoelace *OLD*,
           shoelace *NEW*,
   shoelace_data s,
           unit u
     WHERE (  (int4ge(min(sh.sh_avail, s.sl_avail), 2)
       AND (bpchareq(s.sl_color, sh.slcolor)
           AND float8ge(float8mul(s.sl_len, u.un_fact),
                                float8mul(sh.slminlen, un.un_fact))
           AND float8le(float8mul(s.sl_len, u.un_fact),
                                float8mul(sh.slmaxlen, un.un_fact))
                     )
                )
            AND bpchareq(sh.slunit, un.un_name)
           )
      AND bpchareq(s.sl_unit, u.un_name);
```

Lo reducimos otra vez a una instrucción SQL real que sea equivalente en la salida final del sistema de reglas:

```
    SELECT sh.shoename, sh.sh_avail,
           s.sl_name, s.sl_avail,
           min(sh.sh_avail, s.sl_avail) AS total_avail
```

```
  FROM shoe_data sh, shoelace_data s, unit u, unit un
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 limite de memoria) es crear tablas y luego crearles reglas a mano con CREATE RULE de forma que una lea a la otra y la otra a la una. Esto no puede ocurrir con el comando CREATE VIEW, porque en la primera creación de una vista la segunda aún no existe, de modo que la primera vista no puede seleccionar desde la segunda.

8.2.3. Reglas de vistas en instrucciones diferentes a SELECT

Dos detalles del arbol de traducción no se han tocado en la descripción de las reglas de vistas hasta ahora. Estos son el tipo de comando (commandtype) y la relación resultado (resultrelation). De hecho, las reglas de vistas no necesitan estas informaciones.

Hay sólo unas pocas diferencias entre un árbol de traducción para una SELECT y uno para cualquier otro comando. Obviamente, tienen otros tipos de comandos, y esta vez la relación resultado apunta a la entrada de la tabla de rango donde irá el resultado.

Cualquier otra cosa es absolutamente igual. Por ello, teniendo dos tablas t1 y t2, con atributos a y b, los árboles de traducción para las dos instrucciones:

```
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 continen una variable que apunta al atributo b de la entrada de la tabla rango para la tabla t2.
- Las expresiones de cualificación comparan los atributos a de ambos rangos para la igualdad.

La consecuencia es que ambos árboles de traducción dan lugar a planes de ejecución similares. En ambas hay joins entre las dos tablas. Para la UPDATE, las columnas que no aparecen de la tabla t1 son añadidas a la lista objetivo por el optimizador, y el árbol de traducción final se lee como:

```
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 transaccion actual (ctid). De este modo, la fila anterior se oculta tras el commit de la transacción, y el limpiador vacuum puede realmente eliminarla.

Conociendo todo eso, podemos simplemente aplicar las reglas de las vistas exactamente en la misma forma en cualquier comando. No hay diferencia.

8.2.4. 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).

8.2.4.1. Beneficios

Los beneficios de implementar las vistas con el sistema de reglas están en que el

optimizados tiene toda la información sobre qué tablas tienen que ser revisadas, más las relaciones entre estas tablas, más las cualificaciones restrictivas a partir de la definición de las vistas, más las cualificaciones de la query original, todo en un único árbol de traducción. Y esta es también la situación cuando la query original es ya una join entre vistas. Ahora el optimizador debe decidir cuál es la mejor ruta para ejecutar la query. Cuanta más información tenga el optimizador, mejor será la decisión. Y la forma en que se implementa el sistema de reglas en Postgres asegura que toda la información sobre la query está utilizable.

8.2.4.2. Puntos delicados a considerar

Hubo un tiempo en el que el sistema de reglas de Postgres se consideraba agotado. El uso de reglas no se recomendaba, y el único lugar en el que trabajaban era las reglas de las vistas. E incluso estas reglas de las vistas daban problemas porque el sistema de reglas no era capaz de aplicarse adecuadamente en más instrucciones que en SELECT (por ejemplo, no trabajaría en una UPDATE que utilice datos de una vista).

Durante ese tiempo, el desarrollo se dirigió hacia muchas características añadidas al traductor y al optimizador. El sistema de reglas fué quedando cada vez más desactualizado en sus capacidades, y se volvió cada vez más dificil de actualizar. Y por ello, nadie lo hizo.

En 6.4, alguien cerró la puerta, respiró hondo, y se puso manos a la obra. El resultado fué el sistema de reglas cuyas capacidades se han descrito en este documento. Sin embargo, hay todavía algunas construcciones no manejadas, y algunas fallan debido a cosas que no son soportadas por el optimizador de queries de Postgres.

- Las vistas con columnas agregadas tienen malos problemas. Las expresiones agregadas en las cualificaciones deben utilizarse en subselects. Actualmente no es posible hacer una join de dos vistas en las que cada una de ellas tenga una columna agregada, y comparar los dos valores agregados en a cualificación. Mientras tanto, es posible colocar estas expresiones agregadas en funciones con los argumentos apropiados y utilizarlas en la definición de las vistas.

- Las vistas de uniones no son soportadas. Ciertamente es sencillo reescribir una SELECT simple en una unión, pero es un poco más dificil si la vista es parte de una join que hace una UPDATE.
- Las clausulas ORDER BY en las definiciones de las vistas no están soportadas.
- DISTINCT no está soportada en las definiciones de vistas.

No hay una buena razon por la que el optimizador no debiera manipular construcciones de árboles de traducción que el traductor nunca podría producir debido a las limitaciones de la sintaxis de SQL. El autor se alegrará de que estas limitaciones desaparezcan en el futuro.

8.2.5. Efectos colaterales de la implementación

La utilización del sistema de reglas descrito para implementar las vistas tiene algunos efectos colaterales divertidos. Lo siguiente no parece trabajar:

```
al_bundy=> INSERT INTO shoe (shoename, sh_avail, slcolor)
al_bundy->     VALUES (’sh5’, 0, ’black’);
INSERT 20128 1
al_bundy=> SELECT shoename, sh_avail, slcolor FROM shoe_data;
shoename  |sh_avail|slcolor
-------+-----+-------
sh1       |       2|black
sh3       |       4|brown
sh2       |       0|black
sh4       |       3|brown
(4 rows)
```

Lo interesante es que el código de retorno para la INSERT nos dió una identificación de objeto, y nos dijo que se ha insertado una fila. Sin embargo no aparece en `shoe_data`. Mirando en el directorio de la base de datos, podemos ver que el fichero de la base de datos para la relación de la vista `shoe` parece tener ahora un bloque de datos. Y efectivamente es así.

Podemos también intentar una DELETE, y si no tiene una cualificación, nos dirá que las filas se han borrado y la siguiente ejecución de vacuum limpiará el fichero hasta tamaño cero.

La razon para este comportamiento es que el árbol de la traducción para la INSERT no hace referencia a la relación shoe en ninguna variable. La lista objetivo contiene sólo valores constantes. Por ello no hay reglas que aplicar y se mantiene sin cambiar hasta la ejecución, insertandose la fila. Del mismo modo para la DELETE.

Para cambiar esto, podemos definir reglas que modifiquen el comportamiento de las queries no-SELECT. Este es el tema de la siguiente sección.

8.3. Reglas sobre INSERT, UPDATE y DELETE

8.3.1. Diferencias con las reglas de las vistas.

Las reglas que se definien 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.

8.3.2. 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 cualficación 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 busando 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.

8.3.2.1. Una primera regla paso a paso.

Queremos tracear 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 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 siutación 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 tio 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 multiples 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 absolútamente 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.

8.3.3. 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 tiene 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 absolútamente 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 exactametne 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 recien 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, showlace_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 showlace_data
 WHERE bpchareq(shoelace.sl_name, showlace_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 showlace_data,
        shoelace *OLD*, shoelace *NEW*,
        shoelace_data s, unit u
 WHERE bpchareq(s.sl_name, showlace_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 showlace_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,  showlace_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, descrubrimos que la relación `shoelace_data` aparece dos veces en la tabla de rango, lo que se debería reducir a una sóla. 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 reglás ha probocado un barrido extra de la relación `shoelace_data` absolútamente 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 absolútamente 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->     (’sl9’, 0, ’pink’, 35.0, ’inch’, 0.0);
    al_bundy=> INSERT INTO shoelace VALUES
    al_bundy->     (’sl10’, 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
-------+-----+-------+----+-----+------
sl9       |       0|pink      |    35|inch    |     88.9
sl10      |    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 dificil para Postgres, no la borramos directamente. En su lugar, crearemos una nueva vista

```
CREATE VIEW shoelace_candelete AS
    SELECT * FROM shoelace_obsolete WHERE sl_avail = 0;
```

Y lo haremos de esta forma:

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_candelete
             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
-------+-----+-------+----+-----+------
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
sl10      |    1000|magenta   |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
sl6       |      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.

8.4. 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 contralos 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_numbre. El sistema de reglas reescribirá la SELECT de phone_numbre 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_numbre, 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_numbre 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_numbre, 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 Al (suponiendo que no fuera el mismo Al) podría haber autorizado (GRANT) SELECT, INSERT, UPDATE o DELETE a la vista shoelace a Al. Pero sólo SELECT en shoelace_log. La acción de la regla de escribir entradas del log deberá ser ejecutada con exito, y Al 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.

8.5. 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, poruqe 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 manupulan 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 se 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 memoría caché.

La última query que veremos es

```
DELETE FROM computer WHERE manufacurer = 'bim';
```

De nuevo esto debería dar como resultado muchoas 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.manufacurer = '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 acontencimientos 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 cualficación 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 hecjo de que su accoión 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 mirándo 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 9. 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 médodo 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 9-1. Esquema de un Índice

Atributo	Descripción
amname	nombre del método de acceso
amowner	identificador de objeto del propietario de esta instancia en pg_user
amstrategies	número de estrategias para este método de acceso (véase más abajo)
amsupport	número de rutinas de soporte para este método de acceso (véase más abajo)

Atributo	Descripción
amorderstrategy	cero si el índice no ofrece secuencia de ordenamiento, sino el número de estrategia del operador de estrategia que describe la secuencia de ordenamiento
amgettuple	
aminsert	
...	indicadores de procedimiento para las rutinas de interfaz con el método de acceso. Por ejemplo, aquí aparecen identificadores regproc 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 similaridad 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 9-2. 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 typname = '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 9-3. Esquema de `pg_amproc`

Atributo	Descripción
amopid	el `oid` de la instancia de `pg_am` para B-tree (== 403, véase arriba)
amopclaid	el `oid` de la instancia de `pg_opclass` para `complex_abs_ops`(== lo que obtuvo en vez de `17314`, véase arriba)
amopopr	los `oid`s de los operadores para la clase de operador (opclass) (que obtendremos dentro de un minuto)

Entonces necesitamos los `oid`s 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 `oid`s 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 typname = 'complex_abs'

UPDATE pg_operator
    SET oprrest = 'neqsel'::regproc, oprjoin = 'neqjoinsel'
    WHERE oprname = '' AND
       oprleft = oprright AND
       oprleft = (SELECT oid FROM pg_type WHERE typname = 'complex_abs');

UPDATE pg_operator
    SET oprrest = 'neqsel'::regproc, oprjoin = 'neqjoinsel'
    WHERE oprname = '' AND
       oprleft = oprright AND
       oprleft = (SELECT oid FROM pg_type WHERE typname = 'complex_abs');

UPDATE pg_operator
   SET oprrest = 'scalarltsel'::regproc, oprjoin = 'scalarltjoinsel'
   WHERE oprname = '<' AND
```

```
        oprleft = oprright AND
        oprleft = (SELECT oid FROM pg_type WHERE typname = 'complex_abs');

UPDATE pg_operator
    SET oprrest = 'scalarltsel'::regproc, oprjoin = 'scalarltjoinsel'
    WHERE oprname = '<=' AND
        oprleft = oprright AND
        oprleft = (SELECT oid FROM pg_type WHERE typname = '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');

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 10. GiST Indices

La información sobre GIST está en http://GiST.CS.Berkeley.EDU:8000/gist/ con más sobre diferentes esquemas de ordenación e indexado en http://s2k-ftp.CS.Berkeley.EDU:8000/personal/jmh/ También existe más lectura interesante en el sitio de la base de datos de Berkely en http://epoch.cs.berkeley.edu:8000/.

> **Autor:** Esta extracción de un e-mail enviado por Eugene Selkov Jr. (mailto:selkovjr@mcs.anl.gov) 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 Berkeley (ftp://s2k-ftp.cs.berkeley.edu/pub/gist/pggist/pggist.tgz) vienen con una introducción de los métodos y demuestran mecanismos de índices espaciales para cajas 2D, polígonos, intervalos enteros y testo come with an overview of the methods and demonstrate spatial index mechanisms for 2D boxes, polygons, integer intervals and text (véase también GiST at Berkeley (http://gist.cs.berkeley.edu:8000/gist/)). 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 mensage 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 mi, 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 commentario 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.

Capítulo 11. 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.

11.1. 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
LANCOMPILER '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
    LANCOMPILER '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.

11.2. 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.

11.2.1. 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.

11.2.2. Descripcion

11.2.2.1. Estructura de PL/pgSQL

El lenguaje PL/pgSQL no es sensible a las mayúsculas. Todas las palabras clave e identificadores pueden usarse en cualquier mexcla 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 subloques 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.

11.2.2.2. Comments

Hay dos tipos de comentarios en PL/pgSQl. Un par de guiones '–' comienza un comentario que se extiende hasta el fin de la linea. 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.

11.2.2.3. Declaraciones

Todas las variables, filas y columnas que se usen en un bloque o subloque 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 timepo 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.

11.2.2.4. 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.

11.2.2.5. 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 ’datatime’), 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.

11.2.2.6. 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 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 son 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 explicita, 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 limite 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.

11.2.2.7. 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.

11.2.2.8. 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 abría 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.

11.2.3. 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 de 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.

11.2.3.1. 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’;
```

11.2.3.2. 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’;
```

11.2.3.3. 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.empna
        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();
```

11.3. 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.

11.3.1. 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.

11.3.2. Descripción

11.3.2.1. Funciones de Postgres y nombres de procedimientos Tcl

En Postgres, un mismo nombre de función puede usarse para diferentes funciones, siempre que el numero 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 numero de argumentos) de una misma función de Postgres pueden ser diferentes también para Tcl.

11.3.2.2. 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';
```

11.3.2.3. 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 interprete 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.

11.3.2.4. 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 numero positivo, comenzando por 1, en el que los campos están numerados en el catalogo 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');
```

11.3.2.5. 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_prepare’. 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/ejecutos de la consulta. El valor opcional -count la dice a ’spi_exec’ el máximo numero 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 numero 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 numero 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*? ?-nulls*esquvalue*? ?*loop-body*?

Ejecuta una consulta preparada en ’spi_prepare’ con sustitución de variables. El valor opcional ’-count’ le dice a ’spi_execp’ el máximo numero 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 numero 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 ANI
                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 12. Enlazando funciones de carga dinámica

Después de crear y registrar una función definida por el usuario, el trabajo está prácticamente terminado.Postgres, sim embargo debe cargar el fichero de código objeto (e.g., a `.o`, o una biblioteca compartida) que implemente esa funcuín. 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 como 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-releer) 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 estwe proceso. Si comprende lo que realizan estas pruebas, no debería tener ningún problema.

La siguiente terminología se usará más adelante:

- *Carga dinámica (Dynamic loading)* el 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 ésto usando el mecanismo de carga dinámica proporcionado por el sistema operativo.
- *Configuracion 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 tambien funcionan, pero hay que terner 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 aplicacion 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 el el directorio de nivel superior no leible y/o no ejecutable para el usuario "postgres" es un error extremadamente común.)
- Los nombre de simbolos definidos en los fichero objetos no deben estar en conflicto entre sí, ni con los simbolos definidos en Postgres .
- El compilador de C GNU normalmente no dispone de las opciones especiales necesarias para usar la interfase del cargador dinámico del systema. En caso de que esto ocurra, ha de usarse el compilador de C que venga con el sistema operativo.

12.1. ULTRIX

Es muy facil 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 unica restriccion es que debe producir cada

fichero objeto con la opcion -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.

12.2. 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 (termiando 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 asegura de que se envia 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 dificil 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!

12.3. SunOS 4.x, Solaris 2.x y HP-UX

Bajo SunOS 4.x, Solaris 2.x y HP-UX, los ficheros objetos pueden crearse complando 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 opcion -b. Todo esto suena complicado, pero en realidad es mu 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 (normalemente bibliotecas de sistema, como las bibliotecas de C y matemáticas) en la linea de ordenes de ld

Capítulo 13. 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.

13.1. 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 utilizado 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 funcions").

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 explicita 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...

13.2. 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 dis-
paró antes;
   TRIGGER_FIRED_AFTER(event) devuelve TRUE si se disparó después;
   TRIGGER_FIRED_FOR_ROW(event) devuelve TRUE si el trigger se dis-
paró 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 INSE
```

```
   TRIGGER_FIRED_BY_DELETE(event) devuelve TRUE si fue disparado por un DELE
   TRIGGER_FIRED_BY_UPDATE(event) devuelve TRUE si fue disparado por un UPDA

tg_relation
   es un puntero a una estructura que describe la relación dispa-
radora. Mirar
   en src/include/utils/rel.h para ver detalles sobre esta estruc-
tura. Lo más
   interesante es tg_relation->rd_att (descriptor de las tuplas de la relaci
   y tg_relation->rd_rel->relname (nombre de la relación. No es un char*, si
   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, es-
to 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 Eje-
cutor 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 pa-
ra un INSERT
   o un DELETE. Esto es lo que debe devolverse al Ejecutor en el ca-
so 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 argu-
mentos en
   tgargs,
   tgargs es un array de punteros a los argumentos especificados en el CREAT
   TRIGGER. Otros miembros son exclusivamente para uso interno.
```

13.3. 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 con-
sultas 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!

13.4. 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 rettuple;
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))
rettuple = CurrentTriggerData->tg_newtuple;
else
rettuple = 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 quizo 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 pgsql 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 pgsql 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 pgsql 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 pgsql no lo tiene).

Finalmente este es un buen ejemplo de que por que pgsql dice que "implementa un subconjunto extendido de SQL92", no hay trigger en SQL92, son del SQL3.

Capítulo 14. 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.

14.1. 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)
```

Inputs

void **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 arround, 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.

14.2. 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
SPI_ERROR_NOATTRIBUTE if there is an invalid attribute number in attnum (attnum ≤ 0 or >

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_repalloc` 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

14.3. 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!

14.4. 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.

14.5. Examples

This example of SPI usage demonstrates the visibility rule. There are more complex examples in 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 invo-
cations
```

Capítulo 15. 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.

15.1. 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.)

15.2. 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.

15.3. 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 aparece, 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.

15.3.1. 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);
```

15.3.2. 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.

15.3.3. 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.

15.3.4. 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`.

15.3.5. 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.

15.3.6. 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.

15.3.7. 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 descripto por *fd* a la nueva ubicación especificada por el *desplazamiento*. Los valores válidos para *desde_donde* son SEEK_SET, SEEK_CUR, y SEEK_END.

15.3.8. 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.

15.4. 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';
```

15.5. 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.

15.6. 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/
 *
 *-----------------------------------------
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE         1024

/*
 * importFile *    importar el archivo "filename" en la ba-
se de datos 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", filen
    }

    /*
     * 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 ar-
chivo 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
                  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 suc-
cessfully made */
            if (PQstatus(conn) == CONNECTION_BAD) {
             fprintf(stderr,"Falló la conexión con la base de da-
tos '%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 ob-
jeto grande con X's\n");
            overwrite(conn, lobjOid, 1000, 1000);
        */
```

```
    printf("exportando el objeto grande al archivo %s\n", out_filen
/*    exportFile(conn, lobjOid, out_filename); */
    lo_export(conn, lobjOid,out_filename);

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    exit(0);
}
```

Capítulo 16. 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++), `libpgtcl` (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`.

16.1. Funciones de Conexión a la Base de Datos

Las siguientes rutinas le permitirán realzar 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 traves 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 PQconnetStart / PQconnectPoll resulta preferible para la programación de las aplicaciones. La cadena pasada puede estar varía 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 espación, 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 de 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 antigüos.

- `PQconnectStart` `PQconnectPoll` Realizan una conexión al servidor de base de datos de forma no bloqueante.

  ```
  PGconn *PQconnectStart(const char *conninfo)
  ```

  ```
  PostgresPollingStatusType *PQconnectPoll(PQconn *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 lo 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 restricciónes:

  - 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, asegurese de que el objeto de la secuencia en la cual realiza usted un rastreo no bloquea.
  - Asegurese usted mismo de que el socket se encuentra en el estado apropiado antes de llamar a PQconnetPoll, 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 valido (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), considerelo ’activo’. Si ya ha decido que el 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 ántes. 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 particual, o en todos, 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.

- `PQconndefaults` Devuelve la opciones de conexión de defecto.

```
PQconninfoOption *PQconndefaults(void)

struct PQconninfoOption
{
    char   *keyword;   /* Palabra clave de la opción */
    char   *envvar;    /* Nombre de la variable de entorno que re-
coge su valor
                          si no se da expresamente */
    char   *compiled;  /* Valor de defecto en el código fuente si tam-
poco se asigna
                          variable de entorno */
    char   *val;       /* Valor de la opción */
    char   *label;     /* Etiqueta para el campo en el diálogo de co-
nexión */
    char   *dispchar;  /* Carácter a mostrar para este campo en un diá-
logo 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 ca
po por defecto */
    int     dispsize;  /* Tamaño del campo en carácteres para dia-
logo */
}
```

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. Prohiba 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 probocar 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 situaciónes 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 negocación 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 carácteres 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 PGsetenvHandle. En otro vaso, se devuelve una estructura handle valida. (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 procecimiento utilizando PQsetenvPoll, exactamente del mismo modo en que hubiese creado la conexión utilizando PQconnectPoll.

  El procedimiento se puede abortar en cualquier momento llamando a PQsetevnAbort (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.

16.2. 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 resulado 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 antíguo 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 siguentes 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 al 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 númerod 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 indice 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 explicitamente 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. Todas los resultados de las consultas deberían liberarse con PQclear cuando ya no son necesarios.

  ```
  void PQclear(PQresult *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 sta-
  tus);
  ```

  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.

16.3. Procesamiento Asíncrono de Consultas

La función `PQexec` es adecuada para emitir consultas en aplicaciones síncronas

sencillas. Sin embargo, tiene una porción de definciencias 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 dificil 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 comands 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 antíguas pueden rechazar el uso de `PQsetnonblocking` y mantener el comportamiento anterior potencialmente bloquante. 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 leido 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 probocando 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 s hay algún tipo de problema (en cuyo caso se fija `PQerrorMessage`). Tengase 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, probocando 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 resultado 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 sera >= 0; un resultado de 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 conexiónes 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 terminos 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`. Observese que `PQrequestCancel` no tendrá efecto si la conexión no está abierta en ese momento, o si el servidor no está procesando una consulta.

16.4. 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 usuario no necesitarán esta carácterística.

- `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 llamade 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.

16.5. 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. Subsiguientemenet, la llegada de mensajes NOTIFY se puede detectar llamando a PQnotifies().

- `PQnotifies` Devuelve la siguiente nofication 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 proce-
so servidor */
} PGnotify;
```

Tras procesar un objeto PGnotify devuelto por `PQnotifies`, asegurese 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 absorvidos 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 símplemente `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.

16.6. 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 caracter "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 carácteres 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 leido la línea entera, y 1 si se ha llenado la zona de almacenamiento, pero aún no se ha leido el fin de línea.

  Observese que la aplicación deberá comprobar si la nueva línea consiste en los dos carácteres "\.", 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 carácteres 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 responabilidad 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, asegurese 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 envíar 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 `PGgetline`. 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\thello 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.

16.7. 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)
  ```

16.8. 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 *messa-
ge);

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 sobreescribir 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` inmediamente 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.

16.9. 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 serviodor 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.

16.10. Programas de Ejemplo

16.10.1. 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 servi-
dor. Si los
     * parámetros son nulos, el sistema probará a utilizar valores de defect
     * razonables para buscar en las variables de entorno, y, si es-
to falla,
     * utilizará constantes incluídas directamente en el código.
     */
    pghost = NULL;              /* host name of the backend server */
                                /* nombre del ordenador 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 arran-
car el servidor */
    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 catá-
logo de sistema de
     * bases de datos
     */
    res = PQexec(conn, "DECLARE mycursor CURSOR FOR select * from pg_databas
    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); */
    }
```

16.10.2. 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 venta-
na 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 servi-
dor. Si los
     * parámetros son nulos, el sistema probará a utilizar valores de defect
     * razonables para buscar en las variables de entorno, y, si es-
to falla,
     * utilizará constantes incluídas directamente en el código.
     */
    pghost = NULL;              /* host name of the backend server */
                                /* nombre del ordenador del servi-
dor */
    pgport = NULL;              /* port of the backend server */
                                /* puerto asignado al servidor */
    pgoptions = NULL;           /* special options to start up the backend
                                 * server */
```

```
                                /* opciones especiales para arran-
car el servidor */
    pgtty = NULL;               /* 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 nece-
sario, 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 comprobaciónes; esperar con select
         *     sería más eficiente.
         */
        sleep(1);
        /* collect any asynchronous backend messages */
        /*       Se recoge asíncronamente cualquier mensaje del ser-
vidor */
        PQconsumeInput(conn);
        /* check for asynchronous notify messages */
        /*       Se comprueban los mensajes de notificación asín-
crona */
        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 límpia */
    PQfinish(conn);
```

```
}
```

16.10.3. 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   boundbox = (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   boundbox =
 * (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 servi-
dor. Si los
     * parámetros son nulos, el sistema probará a utilizar valores de defect
     * razonables para buscar en las variables de entorno, y, si es-
to falla,
     * utilizará constantes incluídas directamente en el código.
     */
    pghost = NULL;              /* host name of the backend server */
                                /* nombre de ordenador del servi-
dor */
    pgport = NULL;              /* port of the backend server */
                                /* puerto asignado al servidor. */
    pgoptions = NULL;           /* special options to start up the backend
                                 * server */
                                /* opciones especiales para arran-
car el servidor */
    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 ba-
se de datos 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 ne-
cesario, 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 catálo-
go de sistema de
     *       bases de datos.
     */
    res = PQexec(conn, "DECLARE mycursor BINARY CURSOR FOR select * from tes
    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 necesitamo
             *       incrementar con VARHDSZ
             */
            pval = (POLYGON *) malloc(plen + VARHDRSZ);
            pval->size = plen;
            memmove((char *) &pval->npts, PQgetvalue(res, i, p_fnum), plen);
            printf("tuple %d: got\n", i);
            printf(" i = (%d bytes) %d,\n",
                   PQgetlength(res, i, i_fnum), *ival);
            printf(" d = (%d bytes) %f,\n",
                   PQgetlength(res, i, d_fnum), *dval);
            printf(" p = (%d bytes) %d points \tboundbox = (hi=%f/%f, lo = %f,%f
                   PQgetlength(res, i, d_fnum),
                   pval->npts,
                   pval->boundbox.xh,
                   pval->boundbox.yh,
                   pval->boundbox.xl,
                   pval->boundbox.yl);
        }
        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);

    }
```

Capítulo 17. 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.

17.1. Control e Inicialización

17.1.1. 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 nomber 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 sesion 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.

17.2. Clases de libpq++

17.2.1. 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.

17.2.2. 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++.

17.3. 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 facilmente 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.

17.4. Funciones de Ejecución de las Consultas

- `Exec` Envia una consulta al servidor. Probáblemente 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 exito.

- `ExecTuplesOk` Envía una consulta de tuplas al servidor.

  ```
  int PgConnection::ExecTuplesOk(const char *query)
  ```

  Devuelve VERDADERO si la consulta se realizó con exito.

- `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()
  ```

17.5. Notificación Asíncrona

Postgres soporta notificación asíncrona a través de los comandos **LISTEN** y **NOTIFY**. Un servidor registra su interes 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.

17.6. 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 razon, 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 linea 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 caracter null.

`PgDatabase::GetLine` Devuelve EOF al final de un fichero, 0 si se ha leido 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 18. pgtcl

`pgtcl` 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.

18.1. Comandos

Tabla 18-1. Comandos `pgtcl`

Comando	Descripción
pg_connect	abre una conexión al servidor backend
pg_disconnect	cierra una conexión
pg_conndefaults	obtiene las opciones de conexión y sus valores por defecto
pg_exec	envía una consulta al backend
pg_result	manipula los resultados de una consulta
pg_select	hace un bucle sobre el resultado de una declaración SELECT
pg_listen	establece una rellamada mensajes NOTIFY
pg_lo_creat	crea un objeto grande
pg_lo_open	abre un objeto grande
pg_lo_close	cierra un objeto grande
pg_lo_read	lee un objeto grande
pg_lo_write	escribe un objeto grande
pg_lo_lseek	busca y se coloca sobre una posición en un objeto grande

Comando	Descripción
pg_lo_tell	devuelve la posición de un objeto grande sobre la que se está
pg_lo_unlink	borra un objeto grande
pg_lo_import	importa un fichero Unix a un objeto grande
pg_lo_export	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 descripor 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.

18.2. 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
}
```

18.3. 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 *pqtty*]

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`

Especifiva un handle válido para una base de datos.

`stringConsulta`

Especifica una consulta SQL válida.

Outputs

`handleResultado`

Se devolverá un error Tcl si Pgtcl 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 Pgtcl 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 válidad 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 objto 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 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.

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 RedHat (http://www.redhat.com) 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 verique 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 recientementente. Hasta ahora, RedHat mantiene documentación útil ligeramente anticuada pero inmóvil a http://www.redhat.com/apps/support/.

Nombre

—

```
( );
```

Capítulo 19. Interfaz ODBC

Nota: Información de fondo realizada originalmente por Tim Goeke (mailto:tgoeke@xpressway.com)

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.

19.1. 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 servidoresMS 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 Distributors (http://www.insightdist.com/) dan soporte continuo y actual a la distribución psqlODBC. Suministran un FAQ (http://www.insightdist.com/psqlodbc/), sobre el desarrollo actual del código base, y participan activamente en la lista de correo de interfaces (mailto:interfaces@postgresql.org).

19.2. 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.

19.2.1. 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.

19.3. 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.

19.3.1. 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 AS200 (http://www.as220.org/FreeODBC/iodbc-2.12.shar.Z). 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 driverpsqlODBC 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 (POSTGRESDIR). 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 desempaquételo en un directorio vacío.
2. Desde el directorio donde se encuentran los fuentes, teclee:

   ```
   % ./configure
   % make
   % make POSTGRESDIR=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=instfil
   stall
   ```

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` exists) 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 Distributors (http://www.insightdist.com/psqlodbc), 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. Desempaquete 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/iodbc`, 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 en la línea de comandos de la orden ’./configure’ con –with-odbc.

8. Instala el código fuente:

   ```
   % make POSTGRESDIR=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 POSTGRESDIR 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 POSTGRESDIR.

Aquí es donde usted podrían especificar varios destinos explícitamente:

```
% make BINDIR=bindir LIBDIR=libdir HEADERDIR=headerdir install
```

Por ejemplo, tecleando

```
% make POSTGRESDIR=/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 POSTGRESDIR=/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.

19.4. 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 = POSTGRESDIR/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
```

19.5. ApplixWare

19.5.1. 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 haba 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 inzquierda 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.

19.5.2. 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 Campbell (mailto:scampbell@lear.com).

Author: Contribuido por Steve Campbell (mailto:scampbell@lear.com) 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.

19.5.3. 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 -aucx | 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, despues 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.

19.5.4. 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.

19.5.5. 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.

19.5.6. 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.

Capítulo 20. JDBC Interface

Author: Written by Peter T. Mount (peter@retep.org.uk), 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.

20.1. Building the JDBC Interface

20.1.1. 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.

20.1.2. Installing the Driver

To use the driver, the jar archive postgresql.jar needs to be included in the CLASSPATH.

20.1.2.1. 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.

20.2. 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.

20.3. 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.

20.4. 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.

20.5. 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 it's 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.

20.6. 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://*host*/*database*
- jdbc:postgresql://*host*:*port*/*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);
```

20.7. 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.

20.7.1. 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.

20.7.2. 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();
```

20.8. 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)");
```

20.9. Closing the Connection

To close the database connection, simply call the close() method to the Connection:

```
db.close();
```

20.10. 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 va-
lues (?,?)");
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 ima-
ges 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.

20.11. 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 ti-
me 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 imbeded 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:
                null if no data, Integer if an integer result, or
byte[]
                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
prefered 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:
                null if no data, Integer if an integer result, or
byte[]
                otherwise

        Throws: SQLException
                if name is unknown or if a database-access error
occurs.
```

```
        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
overidden, 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 overidden 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 overidden 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 li-
ne 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 overidden 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.PGlseg

java.lang.Object
   |
   +---postgresql.util.PGobject
            |
            +---postgresql.geometric.PGlseg

   public class PGlseg 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 PGlseg(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 PGlseg(PGpoint p1,
               PGpoint p2)

        Parameters:
                p1 - first point
                p2 - second point

 public PGlseg(String s) throws SQLException

        Parameters:
                s - definition of the circle in PostgreSQL’s syntax.

        Throws: SQLException
                on conversion failure

 public PGlseg()

          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 overidden 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 overidden 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 imbeded 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 overidden 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 overidden 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 profides to Java the libpq C interface’s large object API. It consists of two classes, LargeObjectManager, which deals with creating, opening and deleting large obejects, 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 begining 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 begining

        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 ordeserialize 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 overidden 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
overidden.

        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 overidden, 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 overidden 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 overidden to allow the object to be cloned

        Overrides:
                clone in class Object

 public String toString()

          This is defined here, so user code need not overide 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 aquired).

        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.
```

20.12. 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 site (http://www.javasoft.com).

My own web site (http://www.retep.org.uk) contains updated information not included in this document, and also includes precompiled drivers for v6.4, and earlier.

Capítulo 21. Interfaz de Programación Lisp

`pg.el` es una interfaz a nivel de socket a Postgres para emacs.

Autor: Escrito por Eric Marsden (mailto:emarsden@mail.dotcom.fr) 21 Jul 1999.

`pg.el` es una interfaz a nivel de socket a Postgres para emacs (extraordinario editor de texto). El módulo es capaz de asignar tipos de SQL al tipo equivalente de Emacs Lisp. Actualmente no soporta ni encriptación ni autentificación Kerberos, ni objetos grandes (large objects).

El código (version 0.2) está disponible bajo la licencia GNU GPL en
http://www.chez.com/emarsden/downloads/pg.el
(http://www.chez.com/emarsden/downloads/pg.el)

Cambios desde la última versión:

- ahora funciona con XEmacs (probado con Emacs 19.34 y 20.2, y XEmacs 20.4)
- añadidas funciones para proveer metainformación (lista de bases de datos, de tablas, de columnas)
- los argumentos de ‘pg:result‘ son ahora :keywords
- Resistente a MULE
- 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)))
```

Capítulo 22. Codigo Fuente Postgres

22.1. Formateo

El formateo del codigo fuente utiliza un espacio a 4 columnas tabuladas, actualmente con tabulaciones protegidas (i.e. las tabulacines no son expandidas a espacios).

Para emacs, añade lo siguiente (o algo similar) a tu archivo de inicializacion `~/.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 codigo C de Postgres
  (interactive)
  (c-mode)
  (setq-default tab-width 4)
  (c-set-style "bsd")             ; configura c-basic-offset a 4, mas otros
  (c-set-offset 'case-label '+)   ; vuelve la indexacion de la ca-
jas para que se empareje con el cliente PG
  (setq indent-tabs-mode t))      ; nos aseguramos de que mantie-
ne las tabulaciones 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 23. 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.

23.1. 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.

23.2. 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 `libpg`. 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.

23.3. 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.

23.3.1. 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 cambio en los ficheros C generados, pues serán sobreescritos 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 23-1. 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 sencillaEste 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 recogidas 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 clausula WHERE de la consulta de SQL dada en el ejemplo *Una SELECT sencillaEste 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; 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.

23.3.2. 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.

23.4. 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*. Está técnica continúa implementada.

Para información sobre la sintaxis y la creación de reglas en sistema Postgres Diríjase a la *Guía del Usuario de PostgreSQL*.

23.4.1. 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.

23.4.1.1. 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';
```

23.5. Planificador/optimizador

La tarea del *planificador/optimizador* es crear un plan de ejecución óptimo. Primero combina todas las posibles vías de *barrer* (scannear) 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.

23.5.1. 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 tes 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 pro 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.

23.5.2. 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 `qpqual` 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.

23.6. 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`. (Posteriores 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 `qpqual` 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 24. pg_options

Nota: Aportado por Massimo Dal Zotto (mailto:dz@cs.unitn.it)

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-leido por un servidor qeu recibe una señal SIGHUP, haciendo así posible cambiar opciones de tiempo de ejecución sobre la marcha sin necesidad de rearrancar 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 exáctamente 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 leidas 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 enviandole 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 estadisticas 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 leidos 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 codigo 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.

Capítulo 25. Optimización Genética de Consulta en Sistemas de Base de Datos

Author: Escrito por Martin Utesch (utesch@aut.tu-freiberg.de) del Instituto de Control Automático de la Universidad de Minería y Tecnología en Freiberg, Alemania.

25.1. 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 vias 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, árbor-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.

25.2. 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:
-----------------
```

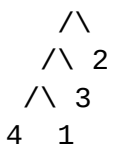
```
P(t)    generación de antecesores en un tiempo t
P”(t)  generación de descendientes en un tiempo t

+=========================================+
|»»»»»»>  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                           |
+===+======================================+
```

25.3. 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



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 especificas de la implementación de OGEC en Postgres son:

- El uso de un AG en *estado constante* (remplazo 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 perdidas 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).

25.4. Futuras Tareas de Implementación para el OGEC de Postgres

25.4.1. Mejoras Básicas

25.4.1.1. 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.

25.4.1.2. Mejora de las configuraciones de los parámetros del algoritmo genético

En el archivo `backend/optimizer/geqo/geqo_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

25.4.1.3. Busqueda de una mejor solución para el desbordamiento de entero

En el archivo `backend/optimizer/geqo/geqo_eval.c`, rutina `geqo_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.

25.4.1.4. 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/geqo/geqo_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.

The Hitch-Hiker's Guide to Evolutionary Computation, Jörg Heitkötter y David Beasley, Recurso de InterNet, *The Design and Implementation of the Postgres Query Optimizer*, Z. Fong, University of California, Berkeley Computer Science Department, *Fundamentals of Database Systems*, R. Elmasri y S. Navathe, The Benjamin/Cummings Pub., Inc..

FAQ en comp.ai.genetic (news://comp.ai.genetic) esta disponible en Encore (ftp://ftp.Germany.EU.net/pub/research/softcomp/EC/Welcome.html).

Archivo `planner/Report.ps` en la documentación de postgres en la distribución.

Capítulo 26. Protocolo Frontend/Backend

Nota: Escrito por Phil Thompson (mailto:phil@river-bank.demon.co.uk). Actualizaciones del protocolo por Tom Lane (mailto:tgl@sss.pgh.pa.us).

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 introdució 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 despues de que la comunicación es establecida), son tratadas en otros lugares.

26.1. Introducción

Los tres principales componentes son el frontend (ejecutandose en el clicente) y el postmaster y backend (ejecutandose 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 autentificación necesita del frontend (si existe) y responde al frontend en concordancia.

El frontend envía entonces cualquier información de autentificación requerida. Una vez que el postmaster valida esta información responde al frontend que está autentificado 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 cosultas/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.

26.2. 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 tambien provisiones especiales para notificación de respuestas y cancelación de comandos, que pueden ocurrir en cualquier instante despues de la fase de inicio.

26.2.1. Inicio

El inicio se divide en fase de autentificación 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 autentificación 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 una diálogo de autentificación 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 autentificación 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 esel password correcto, el postmaster responde con un AuthenticationOk, en caso contrario responde con un ErrorResponse.

Si el frontend no soporta el método de autentificación requerido por el postmaster, debería cerrar inmediatamente la conexión.

Despues 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 despues de un inicio correcto del backend. Proporciona una clave secreta que el frontend debe guardar is 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 llamandas a función.

ErrorResponse

: El arranque del backend no tuvo éxito. La conexión es cerrada despues 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 despues de cada ciclo de consulta. Dependiendo de las necesiades 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.

26.2.2. 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 cadea 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 respuesta 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 envíada 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 seperado 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 frontned 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.

26.2.3. 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 respueste 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 provinientes 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 procesado de la llamada a función se completó. ReadyForQuery se enviará siempre, aunque el procesado 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.

26.2.4. Respuestas de notificación

Si un frontend envía un comando listen(l), entonces el backend enviará un mensaje NotificationResponse (no se confunca 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 (despues 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 notifiaciones 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.

26.2.5. 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 peticion 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 fín 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.

26.2.6. 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 resursos y finalizar. El frontend tiene la opción de lanzar un nuevo backen recontactando el postmaster, si lo desea.

26.3. 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 especificade es el valor literal. P.e. Byte, Byte1(’\n’).

26.4. Formatos de Mensajes

Esta seccion describe el formato detallado de cada mensaje. Cana 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.

Byte*n*

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 autentificación.

Int32(0)

Especifica que la autentificación tuvo éxito.

AuthenticationKerberosV4 (B)

Byte1('R')

Identifica el mensaje como una petición de autentificación.

Int32(1)

Especifica que se requiere autentificación Kerberos V4.

AuthenticationKerberosV5 (B)

Byte1('R')

Identifica el mensaje como una petición de autentificación.

Int32(2)

Especifica que se requiere autentificación Kerberos V5.

AuthenticationUnencryptedPassword (B)

Byte1('R')

Identifica el mensaje como una petición de autentificación.

Int32(3)

Especifica que se requiere una contraseña no encriptada.

AuthenticationEncryptedPassword (B)

Byte1('R')

Identifica el mensaje como una petición de autentificación.

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 valore se 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 fial 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.

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" el los 16 bits más significativos, y "5678" en los 16 bits menos significativos. Para evitar confisión, 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 objectivo.

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.

Byte*n*

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.

Byte*n*

Especifia el valor del resultado en formato binario. *n* Es el tamaño anterior.

Byte1('0')

Sin utilizar. (Hablando propiamente, FunctionResultResponse y FunctionVoidResponse 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 mansaje como una repuesta 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 numero 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 linea 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.

Capítulo 27. Señales de Postgres

Nota: Contribución de Massimo Dal Zotto (mailto:dz@cs.unitn.it)

Postgres usa las siguientes señales para la comunicación entre el postmaster y los backends:

Tabla 27-1. Señales Postgres

Signal	Acción postmaster	Acción del s
SIGHUP	kill(*,sighup)	read_pg_optio
SIGINT	die	cancela la cons
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 as
SIGCHLD	reaper	ignorado (prue
SIGTTIN	ignorado	
SIGTTOU	ignorado	
SIGCONT	dumpstatus	
SIGFPE		FloatExceptio

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
```

Capítulo 28. gcc Default Optimizations

Nota: Contributed by Brian Gallew (mailto:geek+@cmu.edu)

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 lineas de longitud. La primera es "*`*section_name`*:" (e.g. "*asm:"). La segunda es una linea de opciones, y la tercera es una linea 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 enun 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 lineas más abajo se encuentra la siguiente sección:

```
- -------SECTION-------
*cc1:


- -------SECTION-------
```

Como puede verse, no hay ninguna opción por defecto. Si siempre compila codigo C usando "-m486 -fomit-frame-pointer", tendria que cambiarlo de este modo:

```
- -------SECTION-------
*cc1:
- -m486 -fomit-frame-pointer

- -------SECTION-------
```

Si queiero poder generar codigo 386 para otro equipo Linux más antiguo que tenga por ahí, tendríamos que hacer algo así:

```
- -------SECTION-------
*cc1:
%{!m386:-m486} -fomit-frame-pointer

- -------SECTION-------
```

Esto omite siempre los punteros de marco; se construirá codigo optimizado para 486 a menos que se especifique -m386 en la linea 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.

Capítulo 29. 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 todavia 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 "librerias" 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**.

29.1. 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.

29.2. 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 metodo 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. Deberia ser cambiado para que tenga sentido. - Thomas 1998-08-04

29.3. 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`*.

29.4. 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 especifico 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*.

29.5. 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 30. 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.

30.1. 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 30-1. 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)	

Campo	Descripción
Datos de Cabecera de Página (PageHeaderData)	

Los primeros 8 bytes de cada página consiten en la cabecera de la página (PpageHeaderData). 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 situan nuevos identificadores de campo a partir de los primeros cuatro bytes de espacio libre. Debido a que un identificador de campo nonca se mueve hasta que se elimina, este índice se puede utilizar para indicar la situación de un campo en l apágina. 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 situa 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.

30.2. Ficheros

`data/`

Localización de los ficheros de base de datos compartidos (globales).

`data/base/`

Localización de los ficheros de base de datos locales.

30.3. 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.

Apéndice DG1. El Repositorio del CVS

El código fuente de Postgres se almacena y administra utiliando 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.

DG1.1. Organización del árbol de CVS

Author: Escrito por Marc G. Fournier (mailto:scrappy@hub.org) el 1998-11-05.

Traductor: Traducido por Equipo de traducción de PostgreSQL (mailto:doc-postgresql-es@listas.hispalinux.org) el 2001-03-14.

(N. del T: Ismael Olea ha escrito un estupendo documento llamado *"Micro-cómo empezar a trabajar con cvs"*, muy facil 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 personal (http://slug.HispaLinux.ES/~olea/micro-como-empezar-con-cvs.html) y desde luego pidiendoselo directamente a él olea@hispafuentes.com (mailto:olea@hispafuentes.com). 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.

DG1.2. 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/ 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 lo-
   gin
   ```

   Se le preguntará us 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 realizda. 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 pgsql
> ```
>
> 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/.

DG1.3. 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 Polstra (mailto:jdp@polstra.com) para distribuir repositorios CVS y otro árboles de ficheros para El proyecto FreeBSD (http://www.freebsd.org).

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.

DG1.3.1. 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/
```

DG1.3.2. 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 ficheros 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 CV-
Sup
# 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 especi-
ficado 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 dis-
tribución/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 Postgres (ftp://ftp.postgresql.org/pub/CVSup/README.cvsup) y descargará únicamente la foto fija actual:

```
# Este fichero representa el fichero de distribución estandar de CV-
Sup
# para el proyecto de ORDBMS PostgreSQL.
#
# Valores de defecto que se aplican a todas las recolecciones, a to-
das 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 distri-
buciones 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
```

DG1.3.3. 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 Postgres (ftp://postgresql.org/pub), 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 Postgres (ftp://postgresql.org/pub).

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
```

DG1.3.4. 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 Montréal (http://m3.polymtl.ca/m3), quien mantiene activamente el código base

originalmente desarrollado por the DEC Systems Research Center (http://www.research.digital.com/SRC/modula-3/html/home.html). La distribución RPM “PM3” está comprimida aproximadamente unos 30 MB. En el momento de escribir, la versión 1.1.10-1 se instalaba límpiamente 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 zxf cvsup-16.0.tar.gz
```

3. Construya la distribución de cvsup, suprimiento 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
```

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.

DG2.1. 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)

Fichero	Descripción
./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 .

DG2.2. 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 (http://nis-www.lanl.gov/~rosalia/mydocs/docbook-intro.html) para tener un buen y rápido sumario de las características de DocBook. DocBook Elements

(http://www.ora.com/homepages/dtdparse/docbook/3.0/) le da una poderosa referencia de las características de DocBook.

El conjunto de esta documentación se ha construido usando varias herramientas, incluyendo jade (http://www.jclark.com/jade/) de James Clark y Modular DocBook Stylesheets (http://www.nwalsh.com/docbook/dsssl/) de Norm Walsh.

Normalmente las copias impresas se producen importando ficheros *Rich Text Format* (RTF) desde jade a ApplixWare para eliminar algún error de formato y a partir de aquí exportarlo como fichero Postscript.

TeX (http://sunsite.unc.edu/pub/packages/TeX/systems/unix/) 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.

DG2.3. 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 (http://www.ora.com/davenport/) *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.

DG2.3.1. 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.

DG2.3.2. Estilos y convenciones

DocBook tiene un 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 y mantenidas como ejemplos del uso de DocBook . También se incluirá abajo un pequeño sumario de etiquetas de DocBook.

DG2.3.3. 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.

DG2.3.3.1. 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 añada 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".

DG2.4. 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

POSTGRESDIR= /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
```

DG2.5. 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/docbook2X/
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.

DG2.6. 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.

DG2.6.1. 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.

DG2.6.2. 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 titulo 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`.

DG2.7. 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 incluirimos aquí.

DG2.7.1. 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 (ftp://ftp.cygnus.com/pub/home/rosalia/) para Jade y los paquetes relacionados.
2. Instale las últimas hojas de estilo de Norm Walsh. Dependiendo de la antiguedad 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.

DG2.7.2. 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_FI
SGML_CATALOG_FILES=/usr/local/share/sgml/transpec/catalog:$SGML_CATALOG_F
SGML_CATALOG_FILES=/usr/local/share/sgml/docbook/catalog:$SGML_CATALOG_FI
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.

DG2.7.3. 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/dsssl 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
```

DG2.7.4. 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 sgmltools (http://www.sgmltools.org/). 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.

DG2.7.4.1. 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 Clark (ftp://ftp.jclark.com/pub/jade/) (la versión 1.1 en el fichero `jade1_1.zip` en el momento de escribir esto)
- DocBook versión 3.0 (http://www.ora.com/davenport/docbook/current/docbk30.zip)
- Modular Stylesheets (http://nwalsh.com/docbook/dsssl/) de Norman Walsh (la versión 1.19 fue originalmente usada para producir estos documentos)
- PSGML (ftp://ftp.lysator.liu.se/pub/sgml/) 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 Jade (http://www.jclark.com/jade/)
- La página web de DocBook (http://www.ora.com/davenport/)
- La web de Modular Stylesheets (http://nwalsh.com/docbook/dsssl/)
- Web de PSGML (http://www.lysator.liu.se/projects/about_psgml.html)
- La guía de Steve Pepper (http://www.infotek.no/sgmltool/guide.htm)
- Base de datos de Robin Cover sobre software SGML (http://www.sil.org/sgml/publicSW.html)

DG2.7.4.2. 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 aquitectura 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.

DG2.7.4.3. 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 declaracion 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.
   PUBLIC "-//Davenport//ELEMENTS DocBook Document Hierarchy V3.0//EN" dbhie
   PUBLIC "-//Davenport//ENTITIES DocBook Additional General Enti-
   ties V3.0//EN" dbgenent.mod
   ```

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.

DG2.7.4.4. 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
      ```

DG2.7.4.5. Instalación de PSGML

Instalación de PSGML

1. Lea las instrucciones de instalación en la URL mostrada más arriba.
2. Desempaquete 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))
(autoload '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.

DG2.7.4.6. 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 LaTeX2eestén 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 ../grap-
hics 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.

DG2.8. Otras herramientas

sgml-tools v2.x diseñada para jade y DocBook.

Bibliografía

Selección de referencias y lecturas sobre SQL y Postgres.

Libros de referencia sobre SQL

The Practical SQL Handbook , Bowman et al, 1993 , *Using Structured Query Language* , 3, Judity Bowman, Sandra Emerson, y Marcy Damovsky, 0-201-44787-8, 1996, Addison-Wesley, 1997.

A Guide to the SQL Standard , Date and Darwen, 1997 , *A user's guide to the standard database language SQL* , 4, C. J. Date y Hugh Darwen, 0-201-96426-0, 1997, Addison-Wesley, 1997.

An Introduction to Database Systems , Date, 1994 , 6, C. J. Date, 1, 1994, Addison-Wesley, 1994.

Understanding the New SQL , Melton and Simon, 1993 , *A complete guide*, Jim Melton y Alan R. Simon, 1-55860-245-3, 1993, Morgan Kaufmann, 1993.

Abstract

Accessible reference for SQL features.

Principles of Database and Knowledge : Base Systems , Ullman, 1988 , Jeffrey D. Ullman, 1, Computer Science Press , 1988 .

Documentación específica sobre PostgreSQL

The PostgreSQL Administrator's Guide , The Administrator's Guide , Editado por Thomas Lockhart, 1998-10-01, The PostgreSQL Global Development Group.

The PostgreSQL Developer's Guide , The Developer's Guide , Editado por Thomas Lockhart, 1998-10-01, The PostgreSQL Global Development Group.

The PostgreSQL Programmer's Guide , The Programmer's Guide , Editado por Thomas Lockhart, 1998-10-01, The PostgreSQL Global Development Group.

The PostgreSQL Tutorial Introduction , The Tutorial , Editado por Thomas Lockhart, 1998-10-01, The PostgreSQL Global Development Group.

The PostgreSQL User's Guide , The User's Guide , Editado por Thomas Lockhart, 1998-10-01, The PostgreSQL Global Development Group.

Enhancement of the ANSI SQL Implementation of PostgreSQL , Simkovics, 1998 , Stefan Simkovics, O.Univ.Prof.Dr.. Georg Gottlob, November 29, 1998, Department of Information Systems, Vienna University of Technology .

Discurre sobre la historia y la sintaxis de SQL y describe la adición de las construcciones INTERSECT y EXCEPT en Postgres. Preparado como "Master's Thesis" con ayuda de O.Univ.Prof.Dr. Georg Gottlob y Univ.Ass. Mag. Katrin Seyr en Vienna University of Technology.

The Postgres95 User Manual , Yu and Chen, 1995 , A. Yu y J. Chen, The POSTGRES Group , Sept. 5, 1995, University of California, Berkeley CA.

Procedimientos y Articulos

Partial indexing in POSTGRES: research project , Olson, 1993 , Nels Olson, 1993, UCB Engin T7.49.1993 O676, University of California, Berkeley CA.

A Unified Framework for Version Modeling Using Production Rules in a Database System , Ong and Goh, 1990 , L. Ong y J. Goh, April, 1990, ERL Technical Memorandum M90/33, University of California, Berkeley CA.

The Postgres Data Model , Rowe and Stonebraker, 1987 , L. Rowe y M. Stonebraker, Sept. 1987, VLDB Conference, Brighton, England, 1987.

Generalized partial indexes (http://simon.cs.cornell.edu/home/praveen/papers/partindex.de95.ps.Z) , , P. Seshadri y A. Swami, March 1995, Eleventh International Conference on Data Engineering, 1995, Cat. No.95CH35724, IEEE Computer Society Press.

The Design of Postgres , Stonebraker and Rowe, 1986 , M. Stonebraker y L. Rowe, May 1986, Conference on Management of Data, Washington DC, ACM-SIGMOD, 1986.

The Design of the Postgres Rules System, Stonebraker, Hanson, Hong, 1987 , M. Stonebraker, E. Hanson, y C. H. Hong, Feb. 1987, Conference on Data Engineering, Los Angeles, CA, IEEE, 1987.

The Postgres Storage System , Stonebraker, 1987 , M. Stonebraker, Sept. 1987, VLDB Conference, Brighton, England, 1987.

A Commentary on the Postgres Rules System , Stonebraker et al, 1989, M. Stonebraker, M. Hearst, y S. Potamianos, Sept. 1989, Record 18(3), SIGMOD, 1989.

The case for partial indexes (DBMS) (http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-17.pdf) , Stonebraker, M, 1989b, M. Stonebraker, Dec. 1989, Record 18(no.4):4-11, SIGMOD, 1989.

The Implementation of Postgres , Stonebraker, Rowe, Hirohama, 1990 , M. Stonebraker, L. A. Rowe, y M. Hirohama, March 1990, Transactions on Knowledge and Data Engineering 2(1), IEEE.

On Rules, Procedures, Caching and Views in Database Systems , Stonebraker et al, ACM, 1990 , M. Stonebraker y et al, June 1990, Conference on Management of Data, ACM-SIGMOD.