Manual del usuario de PostgreSQL

El equipo de desarrollo de PostgreSQL

Editado por

Thomas Lockhart

Manual del usuario de PostgreSQL

por El equipo de desarrollo de PostgreSQL

Editado por Thomas Lockhart

Tabla de contenidos

Lista de tablas

Tabla de ejemplos

Sumario

Postgres, desarrollada originalmente en el Departamento de Ciencias de la Computación de la Universidad de California en Berkeley, fue pionera en muchos de los conceptos de bases de datos relacionales orientadas a objetos que ahora empiezan a estar disponibles en algunas bases de datos comerciales. Ofrece suporte al lenguaje 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. Introduction

Este documento es el manual de usuario del sistema de mantenimiento de bases de datos PostgreSQL (http://postgresql.org/) , originariamente desarrollado en la Universidad de California en Berkeley. PostgreSQL está basada en Postgres release 4.2 (http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html). El proyecto Postgres , liderado por el Porfesor Michael Stonebraker, fue esponsorizado por diversos organismos oficiales u oficiosos de los EEUU: la Agencia de Proyectos de Investigación Avanzada de la Defensa de los EEUU (DARPA), la Oficina de Investigación de la Armada (ARO), la Fundación Nacional para la Ciencia (NSF), y ESL, Inc.

1.1. ¿Qué es Postgres?

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

clases
herencia
tipos
funciones

Otras características aportan potencia y flexibilidad adicional:

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

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

1.2. Breve historia de Postgres

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

1.2.1. El proyecto Postgres de Berkeley

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

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

Postgres forma parte de la implementación de muchas aplicaciones de investigación y producción. Entre ellas: un sistema de análisis de datos financieros, un paquete de monitorización de rendimiento de motores a reacción, una base de datos de seguimiento de asteroides y varios sistemas de información geográfica. También se ha utilizado como una herramienta educativa en varias universidades. Finalmente, Illustra Information Technologies (http://www.illustra.com/) (posteriormente absorbida por Informix (http://www.informix.com/)) tomó el código y lo comercializó. Postgres llegó a ser el principal gestor de datos para el proyecto científico de computación Sequoia 2000 (http://www.sdsc.edu/0/Parts_Collabs/S2K/s2k_home.html) a finales de 1992.

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

1.2.2. Postgres95

En 1994, Andrew Yu (mailto:ayu@informix.com) y Jolly Chen (http://http.cs.berkeley.edu/~jolly/) añadieron un intérprete de lenguage SQL a Postgres. Postgres95 fue publicado a continuación en la Web para que encontrara su propio hueco en el mundo como un descendiente de dominio público y código abierto del código original Postgres de Berkeley.

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

- El lenguage de consultas Postquel fue reemplazado con SQL (implementado en el servidor). Las subconsultas no fueron soportadas hasta PostgreSQL (ver más abajo), pero podían ser emuladas en Postgres95 con funciones SQL definidas por el usuario. Las funciones agregadas fueron reimplementadas. También se añadió una implementación de la cláusula GROUP BY. La interfaz `libpq` permaneció disponible para programas escritos en C.
- Además del programa de monitorización, se incluyó un nuevo programa (psql) para realizar consultas SQL interactivas usando la librería GNU `readline`.
- Una nueva librería de interfaz, `libpgtcl`, soportaba clientes basados en Tcl. Un shell de ejemplo, pgtclsh, aportaba nuevas órdenes Tcl para interactuar con el motor Postgres95 desde programas tcl
- Se revisó la interfaz con objetos grandes. Los objetos grandes de Inversion fueron el único mecanismo para almacenar objetos grandes (el sistema de archivos de Inversion fue eliminado).
- Se eliminó también el sistema de reglas a nivel de instancia, si bien las reglas siguieron disponibles como reglas de reescritura.
- Se distribuyó con el código fuente un breve tutorial introduciendo las características comunes de SQL y de Postgres95.
- Se utilizó GNU make (en vez de BSD make) para la compilación. Postgres95 también podía ser compilado con un gcc sin parches (al haberse corregido el problema de alineación de variables de longitud doble).

1.2.3. PostgreSQL

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

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

Las principales mejoras en PostgreSQL incluyen:

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

1.3. Acerca de esta versión

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

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

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

1.4. 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.

1.5. 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.

1.6. 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).

1.7. 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.

1.8. 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. .

documentación, incluso si la Universidad ha sido notificada de la posibilidad de tales daños.

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

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

Capítulo 2. Sintaxis SQL

Una descripción de la sintaxis general de SQL.

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

2.1. Palabras Clave

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

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

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

2.1.1. Palabras clave reservadas

SQL92 y SQL3 tienen *Palabras clave reservadas* las cuales no están permitidas ni como identificador ni para cualquier uso distinto de señales fundamentales en declaraciones SQL . Postgres tiene palabras clave adicionales con las mismas restricciones. En particular, estas palabras clave no están permitidas para nombre de

tablas o campos, aunque en algunos casos están permitidas para ser etiquetas de columna (pe. en la cláusula AS).

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

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

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

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

```
CASE COALESCE CROSS CURRENT CURRENT_USER
DEC DECIMAL
ELSE END
FALSE FOREIGN
GLOBAL GROUP
```

```
LOCAL
NULLIF NUMERIC
ORDER
POSITION PRECISION
SESSION_USER
TABLE THEN TRANSACTION TRUE
USER
WHEN
```

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

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

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

```
CHAR_LENGTH CHARACTER_LENGTH
```

Las siguientes palabras reservadas de SQL92 o SQL3no son palabras clave reservadas dePostgres pero si se usan como nombre de tipo se traducen siempre en un tipo alternativo/nativo:

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

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

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

```
ALLOCATE ARE ASSERTION AT AUTHORIZATION AVG
BIT BIT_LENGTH
CASCADED CATALOG COLLATION CONNECT CONNECTION
 CONTINUE CONVERT CORRESPONDING COUNT
DATE DEALLOCATE DEC DESCRIBE DESCRIPTOR
 DIAGNOSTICS DISCONNECT DOMAIN
ESCAPE EXCEPT EXCEPTION EXEC EXTERNAL
FIRST FOUND
GET GO GOTO
```

```
IDENTITY INDICATOR INPUT INTERSECT
LAST LOWER
MAX MIN MODULE
OCTET_LENGTH OPEN OUTPUT OVERLAPS
PREPARE PRESERVE
ROWS
SCHEMA SECTION SESSION SIZE SOME
 SQL SQLCODE SQLERROR SQLSTATE SUM SYSTEM_USER
TEMPORARY TRANSLATE TRANSLATION
UNKNOWN UPPER USAGE
VALUE
WHENEVER WRITE
```

2.1.2. Palabras clave no-reservadas

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

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

```
ACCESS AFTER AGGREGATE
BACKWARD BEFORE
CACHE CREATEDB CREATEUSER CYCLE
DATABASE DELIMITERS
EACH ENCODING EXCLUSIVE
FORWARD FUNCTION
HANDLER
INCREMENT INDEX INHERITS INSENSITIVE INSTEAD ISNULL
```

```
LANCOMPILER LOCATION
MAXVALUE MINVALUE MODE
NOCREATEDB NOCREATEUSER NOTHING NOTNULL
OIDS OPERATOR
PASSWORD PROCEDURAL
RECIPE RENAME RETURNS ROW RULE
SEQUENCE SERIAL SHARE START STATEMENT STDIN STDOUT
TRUSTED
VALID VERSION
```

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

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

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

```
COMMITTED SERIALIZABLE TYPE
```

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

```
ADA
C CATALOG_NAME CHARACTER_SET_CATALOG CHARACTER_SET_NAME
 CHARACTER_SET_SCHEMA CLASS_ORIGIN COBOL COLLATION_CATALOG
 COLLATION_NAME COLLATION_SCHEMA COLUMN_NAME
 COMMAND_FUNCTION CONDITION_NUMBER
 CONNECTION_NAME CONSTRAINT_CATALOG CONSTRAINT_NAME
 CONSTRAINT_SCHEMA CURSOR_NAME
DATA DATE_TIME_INTERVAL_CODE DATE_TIME_INTERVAL_PRECISION
 DYNAMIC_FUNCTION
FORTRAN
LENGTH
MESSAGE_LENGTH MESSAGE_OCTET_LENGTH MORE MUMPS
NAME NULLABLE NUMBER
PAD PASCAL PLI
REPEATABLE RETURNED_LENGTH RETURNED_OCTET_LENGTH
 RETURNED_SQLSTATE ROW_COUNT
SCALE SCHEMA_NAME SERVER_NAME SPACE SUBCLASS_ORIGIN
TABLE_NAME
UNCOMMITTED UNNAMED
```

2.2. Comentarios

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

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

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

```
/* multi
   line
   comment
 */
```

2.3. Nombres

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

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

2.4. Constantes

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

2.4.1. Constantes tipo Cadenas

Las cadenas son secuencias arbitrarias de caracteres ASCII limitadas por comillas simples (" ’ ", pe. `’Esto es una cadena’`)SQL92 permite que las comillas simples puedan estar incluidos en una cadena tecleando dos comillas simples adyacentes (pe. `’Dianne”s horse’`). En Postgres las comillas simples deben estar precedidas por una contra barra ("\", pe.. `’Dianne\’s horse’`). para incluir una contra barra en una constante de tipo cadena, teclear dos contra barras. Los caracteres no imprimibles también deben incluir en la cadena precedidos de una contra barra (pe`’\tab’`).

2.4.2. Constantes tipo Entero

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

Destacar que el entero más largo puede ser especificado para int8 utilizando una notación de cadenaSQL92 o una notación del tipo Postgres:

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

2.4.3. Constantes tipo Punto Flotante

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

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

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

Una constante de tipo punto flotante es del tipo float8. Parafloat4 se puede especificar explícitamente usando la notación de cadena de SQL92o notación de tipo Postgres:

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

2.4.4. Constantes Postgres de tipos definido por el usuario

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

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

El valor de dentro de la cadena se pasa como entrada a rutina de conversión para el tipo llamado *type*. El resultado es una constante del tipo indicado. La tipología puede

omitirse si no hay ambigüedad sobre el tipo de constate que debe ser, en este caso este está automáticamente forzado.

2.4.5. Constantes de tipo Array

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

`{`*`val1delimval2delim`*`}`

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

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

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

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

2.5. Campos y Columnas

2.5.1. Campos

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

oid

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

xmin

El identificador de la transacción insertada.

xmax

El identificador de la transacción borrada.

cmin

El identificador del comando dentro de la transacción.

cmax

El identificador del comando borrado.

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

2.5.2. Columnas

Una *columna* se construye de esta forma:

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

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

2.6. Operadores

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

2.7. Expresiones

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

más información).

Una expresión es una de las siguientes:

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

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

2.7.1. Parámetros

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

$*number*

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

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

2.7.2. Expresiones Funcionales

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

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

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

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

2.7.3. Expresiones de Agregación

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

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

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

La primera forma de expresión de agregación llama a la agregación a través de todas

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

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

2.7.4. Lista Objetivo

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

```
a_expr [ AS result_attname ]
```

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

2.7.5. Calificadores

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

NOT
AND
OR

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

2.7.6. Lista From

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

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

Donde *class_reference* es de la forma

```
class_name [ * ]
```

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

Capítulo 3. Data Types

Describes the built-in data types available in Postgres.

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

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

Tabla 3-1. Postgres Data Types

Postgres Type	SQL92 or SQL3 Type	Description
bool	boolean	logical boolean (true/false)
box		rectangular box in 2D plane
char(n)	character(n)	fixed-length character string
cidr		IP version 4 network or host address
circle		circle in 2D plane
date	date	calendar date without time of day
decimal	decimal(p,s)	exact numeric for p <= 9, s = 0
float4/8	float(p)	floating-point number with precision p

Postgres Type	SQL92 or SQL3 Type	Description
float8	real, double precision	double-precision floating-point number
inet		IP version 4 network or host address
int2	smallint	signed two-byte integer
int4	int, integer	signed 4-byte integer
int8		signed 8-byte integer
line		infinite line in 2D plane
lseg		line segment in 2D plane
money	decimal(9,2)	US-style currency
numeric	numeric(p,s)	exact numeric for p == 9, s = 0
path		open and closed geometric path in 2D plane
point		geometric point in 2D plane
polygon		closed geometric path in 2D plane
serial		unique id for indexing and cross-reference
time	time	time of day
timespan	interval	general-use time span
timestamp	timestamp with time zone	date/time
varchar(n)	character varying(n)	variable-length character string

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

Tabla 3-2. Postgres Function Constants

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

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

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

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

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

3.1. Numeric Types

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

Tabla 3-3. Postgres Numeric Types

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

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

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

3.1.1. The Serial Type

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

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

is equivalent to specifying:

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

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

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

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

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

3.2. Monetary Type

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

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

Tabla 3-4. Postgres Monetary Types

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

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

3.3. Character Types

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

Tabla 3-5. Postgres Character Types

Character Type	Storage	Recommendation	Description
char	1 byte	SQL92-compatible	Single character
char(n)	(4+n) bytes	SQL92-compatible	Fixed-length blank padded

Character Type	Storage	Recommendation	Description
text	(4+x) bytes	Best choice	Variable-length
varchar(n)	(4+n) bytes	SQL92-compatible	Variable-length with limit

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

Tabla 3-6. Postgres Specialty Character Type

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

3.4. Date/Time Types

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

Tabla 3-7. PostgreSQL Date/Time Types

Type	Description	Storage	Earliest	Latest
timestamp	for data containing both date and time	8 bytes	4713 BC	AD 1465001
interval	for time intervals	12 bytes	-178000000 years	178000000 ye

Type	Description	Storage	Earliest	Latest
date	for data containing only dates	4 bytes	4713 BC	32767 AD
time	for data containing only times of the day	4 bytes	00:00:00.00	23:59:59.99

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

3.4.1. Date/Time Input

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

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

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

3.4.1.1. date

The following are possible inputs for the date type.

Tabla 3-8. PostgreSQL Date Input

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

Tabla 3-9. PostgreSQL Month Abbreviations

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

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

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

Tabla 3-10. PostgreSQL Day of Week Abbreviations

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

3.4.1.2. time

The following are valid time inputs.

Tabla 3-11. PostgreSQL Time Input

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

3.4.1.3. timestamp

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

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

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

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

is supported.

Tabla 3-12. PostgreSQL Time Zone Input

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

Time Zone	Description
-8	ISO-8601 offset for PST

3.4.1.4. interval

intervals can be specified with the following syntax:

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

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

3.4.1.5. Special values

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

PostgreSQL also supports several special constants for convenience.

Tabla 3-13. PostgresSQL Special Date/Time Constants

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

Constant	Description
today	Midnight today
tomorrow	Midnight tomorrow
yesterday	Midnight yesterday

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

3.4.2. Date/Time Output

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

Tabla 3-14. PostgreSQL Date/Time Output Styles

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

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

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

setting affects interpretation of input values.)

Tabla 3-15. PostgreSQL Date Order Conventions

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

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

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

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

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

3.4.3. Time Zones

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

Two obvious problems are:

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

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

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

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

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

There are several ways to affect the time zone behavior:

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

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

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

3.4.4. Internals

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

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

3.5. Boolean Type

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

bool uses 1 byte of storage.

Tabla 3-16. Postgres Boolean Type

State	Output	Input

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

3.6. Geometric Types

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

Tabla 3-17. Postgres Geometric Types

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

A rich set of functions and operators is available to perform various geometric

operations such as scaling, translation, rotation, and determining intersections.

3.6.1. Point

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

point is specified using the following syntax:

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

3.6.2. Line Segment

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

lseg is specified using the following syntax:

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

3.6.3. Box

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

box is specified using the following syntax:

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

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

3.6.4. Path

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

path is specified using the following syntax:

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

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

3.6.5. Polygon

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

polygon is specified using the following syntax:

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

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

3.6.6. Circle

Circles are represented by a center point and a radius.

circle is specified using the following syntax:

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

Circles are output using the first syntax.

3.7. IP Version 4 Networks and Host Addresses

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

Tabla 3-18. PostgresIP Version 4 Types

IPV4 Type	Storage	Description	Range
cidr	variable	CIDR networks	Valid IPV4 CIDR blocks
inet	variable	nets and hosts	Valid IPV4 CIDR blocks

3.7.1. CIDR

The cidr type holds a CIDR network. The format for specifying classless networks is `x.x.x.x/y` where `x.x.x.x` is the network and `/y` is the number of bits in the

netmask. If `/y` omitted, it is calculated using assumptions from the older classfull naming system except that it is extended to include at least all of the octets in the input.

Here are some examples:

Tabla 3-19. PostgresIP Types Examples

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

3.7.2. inet

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

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

Capítulo 4. Operadores

Describe los operadores propios disponibles en Postgres.

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

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

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

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

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

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

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

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

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

4.1. Lexical Precedence

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

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

Elemento	Precedencia	Descripción
UNION	izquierda	constructor SQ
::		conversor de ti
[ ]	izquierda	delimitadores d
.	izquierda	delimitadores d
-	derecha	menos unario
;	izquierda	terminación de
:	derecha	exponenciación
|	izquierda	comienzo de in
* / %	izquierda	multiplicación,
+ -	izquierda	adición, substra
IS		test para TRUE
ISNULL		test para NULL
NOTNULL		test para NOT
(todos los demás operadores)		nativos y defin
IN		fijar miembro
BETWEEN		continente
LIKE		concordancia d

Elemento	Precedencia	Descripción
< >		desigualdad bo
=	derecha	igualdad
NOT	derecha	negación
AND	izquierda	intersección ló
OR	izquierda	unión lógica

4.2. Operadores generales

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

Tabla 4-2. Postgres Operators

Operador	Descripción	Utilización
<	Menor que?	1 < 2
<=	Menor o igual que?	1 <= 2
<>	No igual?	1 <> 2
=	Igual?	1 = 1
>	Mayor que?	2 > 1
>=	Mayor o igual que?	2 >= 1
||	Concatena strings	'Postgre' || 'SQL'
!!=	NOT IN	3 !!= i
~~	Como	'scrappy,marc,hermit' ~~ '%scrappy%'
!~~	No como	'bruce' !~~ '%al%'
~	Concordancia (regex), sensible a mayusc/minusc	'thomas' ~ '.*thomas.*'

Operador	Descripción	Utilización
~*	Concordancia (regex), sensible a mayusc/minusc	’thomas’ ~* ’.*Thomas.*’
!~	No concuerda (regex), sensible a mayusc/minusc	’thomas’ !~ ’.*Thomas.*’
!~*	No concuerda (regex), sensible a mayusc/minusc	’thomas’ !~* ’.*vadim.*’

4.3. Operadores numéricos

Tabla 4-3. Postgres Operadores numéricos

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

Operador	Descripción	Utilización
||/	Raíz cúbica	||/ 27.0

4.4. Operadores geométricos

Tabla 4-4. Postgres Operadores geométricos

Operador	Descripción	Utilización
+	Translación	’((0,0),(1,1))’::box + ’(2.0,0)’::punto
-	Translación	’((0,0),(1,1))’::box - ’(2.0,0)’::punto
*	Escalado/rotación	’((0,0),(1,1))’::box * ’(2.0,0)’::punto
/	Escalado/rotación	’((0,0),(2,2))’::box / ’(2.0,0)’::punto
#	Intersección	’((1,-1),(-1,1))’ # ’((1,1),(-1,-1))’
#	Número de puntos en polígono	# ’((1,0),(0,1),(-1,0))’
##	Punto más próximo	’(0,0)’::punto ## ’((2,0),(0,2))’::lseg
&&	Se superpone a?	’((0,0),(1,1))’::caja && ’((0,0),(2,2))’::caja
&<	Se superpone por la izquierda?	’((0,0),(1,1))’::caja &< ’((0,0),(2,2))’::caja

Operador	Descripción	Utilización
&>	Se superpone por la derecha?	’((0,0),(3,3))’::caja &> ’((0,0),(2,2))’::caja
<->	Distancia entre	’((0,0),1)’::círculo <-> ’((5,0),1)’::círculo
«	A la izquierda de?	’((0,0),1)’::círculo « ’((5,0),1)’::círculo
<^	Está debajo de?	’((0,0),1)’::círculo <^ ’((0,5),1)’::círculo
»	A la derecha de?	’((5,0),1)’::círculo » ’((0,0),1)’::círculo
>^	Esta encima de?	’((0,5),1)’::círculo >^ ’((0,0),1)’::círculo
?#	Interseca o se superpone	’((-1,0),(1,0))’::lseg ?# ’((-2,-2),(2,2))’::caja;
?-	Es horizontal?	’(1,0)’::punto ?- ’(0,0)’::punto
?-|	Es perpendicular?	’((0,0),(0,1))’::lseg ?-| ’((0,0),(1,0))’::lseg
@-@	Longitud de circunferencia	@-@ ’((0,0),(1,0))’::path
?|	Es vertical?	’(0,1)’::punto ?| ’(0,0)’::punto
?||	Es paralelo?	’((-1,0),(1,0))’::lseg ?|| ’((-1,2),(1,2))’::lseg
@	Contenido en	’(1,1)’::punto @ ’((0,0),2)’::círculo
@@	Centro de	@@ ’((0,0),10)’::círculo
~=	Parecido a	’((0,0),(1,1))’::poligono ~= ’((1,1),(0,0))’::poligono

4.5. Operadores de intervalos de tiempo

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

Tabla 4-5. Postgres Operadores de intervalos de tiempo

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

4.6. Operadores IP V4 CIDR

Tabla 4-6. PostgresOperadores IP V4 CIDR

Operador	Descripción	Utilizacióne
<	Menor que	’192.168.1.5’::cidr < ’192.168.1.6’::cidr
<=	Menor o igual que	’192.168.1.5’::cidr <= ’192.168.1.5’::cidr
=	Igual que	’192.168.1.5’::cidr = ’192.168.1.5’::cidr
>=	Mayor o igual que	’192.168.1.5’::cidr >= ’192.168.1.5’::cidr
>	Mayor que	’192.168.1.5’::cidr > ’192.168.1.4’::cidr
<>	No igual que	’192.168.1.5’::cidr <> ’192.168.1.4’::cidr
«	Está contenido en	’192.168.1.5’::cidr « ’192.168.1/24’::cidr
«=	Está contenido en o es igual a	’192.168.1/24’::cidr «= ’192.168.1/24’::cidr
»	Contiene	’192.168.1/24’::cidr » ’192.168.1.5’::cidr
»=	Contiene o es igual que	’192.168.1/24’::cidr »= ’192.168.1/24’::cidr

4.7. Operdores IP V4 INET

Tabla 4-7. PostgresOperdores IP V4 INET

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

Capítulo 5. Funciones

Describe las funciones built-in disponibles en Postgres.

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

5.1. Funciones SQL

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

Tabla 5-1. Funciones SQL

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

5.2. Funciones Matemáticas

Tabla 5-2. Funciones Matemáticas

Funciones	Retorna	Descripcion	Ejemplo
dexp(float8)	float8	redimensiona al exponente especificado	dexp(2.0)
dpow(float8,float8)	float8	redimensiona un numero al exponente especificado	dpow(2.0, 16.0)
float(int)	float8	convierte un entero a punto flotante	float(2)
float4(int)	float4	convierte un entero a punto flotante	float4(2)
integer(float)	int	convierte un punto flotante a entero	integer(2.0)

5.3. String Functions

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

Tabla 5-3. SQL92 String Functions

Funciones	Retorna	Descripcion	Ejemplo
char_length(string)	int4	longitud del texto	char_length(’jose’)

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

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

Tabla 5-4. Funciones de Texto

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

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

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

5.4. Funciones de Fecha/Hora

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

Tabla 5-5. Date/Time Functions

Funciones	Retorna	Descripcion	Ejemplo
abstime(datetime)	abstime	convierte a abstime	abstime(’now’::datetime)
age(datetime,datetime)	timespan	preserva meses y años	age(’now’,’1957-06-13’::datetime)
datetime(abstime)	datetime	convierte abstime a datetime	datetime(’now’::abstime)
datetime(date)	datetime	convierte date a datetime	datetime(’today’::date)
datetime(date,time)	datetime	convierte adatetime	datetime(’1998-02-24’::datetime, ’23:07’::time);
date_part(text,datetime)	float8	porción de fecha	date_part(’dow’,’now’::datetime)

Funciones	Retorna	Descripcion	Ejemplo
date_part(text,timespan)	float8	porción de hora	date_part('hour','4 hrs 3 mins'::timespan)
date_trunc(text,datetime)	datetime	fecha truncada	date_trunc('month','now'::abstime)
isfinite(abstime)	bool	un tiempo finito ?	isfinite('now'::abstime)
isfinite(datetime)	bool	una hora finita ?	isfinite('now'::datetime)
isfinite(timespan)	bool	una hora finita ?	isfinite('4 hrs'::timespan)
reltime(timespan)	reltime	convierte a reltime	reltime('4 hrs'::timespan)
timespan(reltime)	timespan	convierte a timespan	timespan('4 hours'::reltime)

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

5.5. Funciones de Formato

Author: Written by Karel Zak (mailto:zakkr@zf.jcu.cz) on 2000-01-24.

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

Tabla 5-6. Funciones de Formato

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

Funciones	Retorna	Descripcion	Ejemplo
to_number(text, text)	numeric	convierte string a numeric	to_number(’12,454.8-’, ’99G999D9S’)

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

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

Format-picture	Descripción
HH	hora del día(01-12)
HH12	hora del día(01-12)
MI	minuto (00-59)
SS	segundos (00-59)
SSSS	segundos pasados la medianoche(0-86399)
Y,YYY	año(4 o mas dígitos) con coma
YYYY	año(4 o mas dígitos)
YYY	últimos 3 dígitos del año
YY	últimos 2 dígitos del año
Y	último dígito del año
MONTH	nombre completo del mes(9-letras) - todos los caracteres en mayúsculas
Month	nombre completo del mes(9-letras) - el primer carácter en mayúsculas
month	nombre completo del mes(9-letras) - todos los caracteres en minúsculas
MON	nombre abreviado del mes(3-letras) -todos los caracteres en mayúsculas

Format-picture	Descripción
Mon	nombre abreviado del mes(3-letras) - el primer carácter en mayúsculas
mon	nombre abreviado del mes(3-letras) - todos los caracteres en minúsculas
MM	mes (01-12)
DAY	nombre completo del día(9-letters) - todos los caracteres en mayúsculas
Day	nombre completo del día(9-letters) - el primer carácter en mayúsculas
day	nombre completo del día(9-letters) - todos los caracteres en minúsculas
DY	nombre abreviado del día(3-letters) - todos los caracteres en mayúsculas
Dy	nombre abreviado del día(3-letters) - el primer carácter en mayúsculas
dy	nombre abreviado del día(3-letters) - todos los caracteres en minúsculas
DDD	día del año(001-366)
DD	día del mes(01-31)
D	día de la semana(1-7; SUN=1)
W	semana del mes
WW	número de la semana en el año
CC	centuria (2-digits)
J	día juliano(dias desde Enero 1, 4712 BC)
Q	quarter
RM	mes en númeral romano(I-XII; I=ENE)

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

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

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

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

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

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

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

Format-picture	Descripción

Format-picture	**Descripción**
9	valor retornado con el número especificado de dígitos y si no estan disponibles usa espacios en blanco
0	como 9, pero en lugar de espacios en blanco usa ceros
. (period)	punto decimal
, (comma)	separador de grupo (miles)
PR	retorna el valor negativo en angle brackets
S	retorna el valor negativo con el signo menos (usa locales)
L	símbolo monetario (usa locales)
D	punto decimal (usa locales)
G	separador de grupos (usa locales)
MI	retorna el signo menos en la posición especificada (si número < 0)
PL	retorna el signo mas en la posición especificada (si número > 0) - PostgreSQL extension
SG	retorna el signo mas/menos en la posición especificada - PostgreSQL extension
RN	retorna el número como número romano(número debe ser entre1 y 3999)
TH or th	convierte el número a número ordinal (no convertir números menores que cero y números decimales) - PostgreSQL extension

Format-picture	Descripción
V	arg1 * (10 ^ n); - retorna un valor multiplicado por 10^n (donde ’n’ es número de ’9’s despues de ’V’). El to_char() no soporta el uso de ’V’ y punto decimal juntos, ejemplo "99.9V99".
EEEE	numeros cientificos . ahora no soportados.

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

```
’  -12’
```

, but to_char(-12, ’MI9999’) produce:

```
’-  12’
```

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

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

Input	Output
to_char(now(), ’Day, HH12:MI:SS’)	’Tuesday , 05:39:18’
to_char(now(), ’FMDay, HH12:MI:SS’)	’Tuesday, 05:39:18’
to_char(-0.1, ’99.99’)	’ -.10’
to_char(-0.1, ’FM9.99’)	’-.1’
to_char(0.1, ’0.9’)	’ 0.1’
to_char(12, ’9990999.9’)	’ 0012.0’
to_char(12, ’FM9990999.9’)	’0012’
to_char(485, ’999’)	’ 485’

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

5.6. Funciones Geométricas

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

Tabla 5-11. Funciones Geométricas

Funciones	Retorna	Descripcion	Ejemplo
area(box)	float8	área del rectangulo	area('((0,0),(1,1))'::box)
area(circle)	float8	área del circulo	area('((0,0),2.0)'::circle)
box(box,box)	box	rectangulo de intersección de rectangulos	box('((0,0),(1,1))','((0.5,0.5),(2,2))')
center(box)	point	centro del objeto	center('((0,0),(1,2))'::box)
center(circle)	point	centro del objeto	center('((0,0),2.0)'::circle)
diameter(circle)	float8	diametro del circulo	diameter('((0,0),2.0)'::circle)
height(box)	float8	tamaño vertical del rectangulo	height('((0,0),(1,1))'::box)
isclosed(path)	bool	ruta cerrada ?	isclosed('((0,0),(1,1),(2,0))'::path)

Funciones	Retorna	Descripcion	Ejemplo
isopen(path)	bool	ruta abierta ?	isopen('[(0,0),(1,1),(2,0)]'::path)
length(lseg)	float8	longitud de la linea segmento	length('((-1,0),(1,0))'::lseg)
length(path)	float8	longitud de la ruta	length('((0,0),(1,1),(2,0))'::path)
pclose(path)	path	convierte path a closed	popen('[(0,0),(1,1),(2,0)]'::path)
point(lseg,lseg)	point	intersección	point('((-1,0),(1,0))'::lseg,'((-2,-2),(2,2))'::lseg)
points(path)	int4	número de puntos	points('[(0,0),(1,1),(2,0)]'::path)
popen(path)	path	convierte path a open	popen('((0,0),(1,1),(2,0))'::path)
radius(circle)	float8	radio del círculo	radius('((0,0),2.0)'::circle)
width(box)	float8	tamaño horizontal	width('((0,0),(1,1))'::box)

Tabla 5-12. Funciones de conversión de tipos Geométricos

Funciones	Retorna	Descripcion	Ejemplo
box(circle)	box	convierte circulo a rectangulo	box('((0,0),2.0)'::circle)
box(point,point)	box	convierte puntos a rectangulo	box('(0,0)'::point,'(1,1)'::point)
box(polygon)	box	convierte poligono a rectangulo	box('((0,0),(1,1),(2,0))'::polygon)
circle(box)	circle	convierte a circulo	circle('((0,0),(1,1))'::box)
circle(point,float8)	circle	convierte a circulo	circle('(0,0)'::point,2.0)
lseg(box)	lseg	convierte diagonal a lseg	lseg('((-1,0),(1,0))'::box)
lseg(point,point)	lseg	convierte a lseg	lseg('(-1,0)'::point,'(1,0)'::point)
path(polygon)	point	convierte a path	path('((0,0),(1,1),(2,0))'::polygon)
point(circle)	point	convierte a punto (centro)	point('((0,0),2.0)'::circle)

Funciones	Retorna	Descripcion	Ejemplo
point(lseg,lseg)	point	convierte a punto (intersección)	point('((-1,0),(1,0))'::lseg, '((-2,-2),(2,2))'::lseg)
point(polygon)	point	centro de poligono	point('((0,0),(1,1),(2,0))'::polygon)
polygon(box)	polygon	convierte a poligono con 12 puntos	polygon('((0,0),(1,1))'::box)
polygon(circle)	polygon	convierte a poligono con 12 puntos	polygon('((0,0),2.0)'::circle)
polygon(*npts*,circle)	polygon	convierte a poligono *npts*	polygon(12,'((0,0),2.0)'::circle)
polygon(path)	polygon	convierte a polygon	polygon('((0,0),(1,1),(2,0))'::path)

Tabla 5-13. Funciones de Actualización Geométrica

Funciones	Retorna	Descripcion	Ejemplo
isoldpath(path)	path	test path for pre-v6.1 form	isoldpath('(1,3,0,0,1,1,2,0)'::path)
revertpoly(polygon)	polygon	convierte pre-v6.1 polygon	revertpoly('((0,0),(1,1),(2,0))'::polygon)

Funciones	Retorna	Descripcion	Ejemplo
upgradepath(path)	path	convierte pre-v6.1 path	upgrade-path('(1,3,0,0,1,1,2,0)'::path)
upgrade-poly(polygon)	polygon	convierte pre-v6.1 polygon	upgrade-poly('(0,1,2,0,1,0)'::polygon)

5.7. Funciones PostgresIP V4

Tabla 5-14. FuncionesPostgresIP V4

Funciones	Retorna	Descripcion	Ejemplo
broadcast(cidr)	text	contruye la dirección broadcast como texto	broad-cast('192.168.1.5/24')
broadcast(inet)	text	contruye la dirección broadcast como texto	broad-cast('192.168.1.5/24')
host(inet)	text	extrae la dirección host como texto	host('192.168.1.5/24')
masklen(cidr)	int4	calcula la longitud del netmask	mas-klen('192.168.1.5/24')
masklen(inet)	int4	calcula la longitud del netmask	mas-klen('192.168.1.5/24')

Funciones	Retorna	Descripcion	Ejemplo
netmask(inet)	text	contruye el netmask como texto	net-mask(’192.168.1.5/24’)

Capítulo 6. Conversión de tipos

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

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

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

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

6.1. Conceptos generales

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

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

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

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

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

Operadores

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

Llamadas a funciones

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

Objetivos de consultas

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

Consultas UNION

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

Muchas de las reglas de conversión de tipos generales usan convenciones sencillas que están en las tablas del sistema de funciones y operadores de Postgres. Hay algo de

heurística en las reglas de conversión para dar un mejor soporte a las convenciones de los tipos nativos estándar de SQL92 como smallint, integer, y float.

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

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

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

6.1.1. Guidelines

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

- Las conversiones implícitas no deberían tener nunca un resultado sorprendente o impredecible.
- Los tipos definidos por el usuario, de los cuales el analizador no tiene conocimiento a priori, deben de estar situados en un lugar alto dentro de la jerarquía de tipos.

Dentro de expresiones con tipos mezclados, los tipos nativos deberían ser convertidos siempre a tipos definidos por el usuario (por supuesto, solo si la conversión es necesaria).

- Los tipos definidos por el usuario no están relacionados. Por lo general, Postgres no tiene disponible información sobre las relaciones entre tipos aparte de la lógica codificada para los tipos predefinidos y las relaciones implícitas basadas en las funciones disponibles en en el catálogo.
- No debería haber una carga extra del analizador o del ejecutor si una consulta no necesita conversión implícita de tipos. De esta manera, si una consulta esta bien construida y los tipos ya están adaptados, entonces la consulta debería realizarse sin consumir tiempo extra en el analizador y sin realizar funciones de conversión innecesarias dentro de la consulta.

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

6.2. Operadores

6.2.1. Procedimiento de conversión

Operador de evaluación

1. Inspecciona en busca de un ajuste exacto en el catálogo del sistema pg_operator.

    a. Si un argumento de un operador binario es unknown, entonces se asume que es del mismo tipo que el otro argumento.

   b. Invierte los argumentos, y busca un ajuste exacto con un operador el cual apunta a él mismo ya que es conmutativo. Si lo halla, entonces invierte los argumentos en el árbol del analizador y usa este operador.

2. Busca el mejor ajuste.

   a. Hace una lista de todos los operadores con el mismo nombre.

   b. Si solo hay un operador en la lista usa este si el tipo de la entrada puede ser forzado, y genera un error si el tipo no puede ser forzado.

   c. Guarda todos los operadores con los ajustes más explícitos de tipos. Guarda todo si no hay ajustes explícitos y salta al siguiente paso. Si solo queda un candidato, usa este si el tipo puede ser forzado.

   d. Si algún argumento de entrada es "unknown", categoriza los candidatos de entrada como boolean, numeric, string, geometric, o user-defined. Si hay una mezcla de categorías, o más de un tipo definido por el usuario, genera un error porque la elección correcta no puede ser deducida sin más pistas. Si solo está presente una categoría, entonces asigna el tipo preferido a la columna de entrada que previamente era "unknown".

   e. Escoge el candidato con los ajustes de tipos más exactos, y que ajustan el "tipo preferido" para cada categoría de columna del paso previo. Si todavía queda más de un candidato, o si no queda ninguno, entonces se genera un error.

6.2.2. Ejemplos

6.2.2.1. Operador exponente

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

consulta:

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

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

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

or

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

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

6.2.2.2. Concatenación de cadenas

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

Un argumento sin especificar:

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

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

Concatenación con tipos sin especificar:

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

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

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

6.2.2.3. Factorial

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

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

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

6.3. Funciones

Evaluación de función

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

2. Busca la mejor entrada.

    a. Hace una lista de todas las funciones con el mismo nombre y con el mismo número de argumentos.

    b. Si solo hay una función en la lista, usa esta si los tipos de la entrada pueden ser convertidos, y produce un error si los tipos no pueden ser convertidos.

    c. Guarda todas las funciones con los ajustes más explícitos para los tipos. Guarda todas si no hay ajustes explícitos y salta al siguiente paso. Si solo queda un candidato, usa este si el tipo puede ser convertido.

    d. Si cualquiera de los argumentos de entrada son de tipo desconocido, clasifica los argumentos de entrada candidatos en categorías como boolean, numeric, string, geometric o user-defined. Si hay una mezcla de categorías, o más de un tipo definido por el usuario, se produce un error debido a que la elección correcta no puede ser deducida si no se aportan más pistas. Si solo hay una categoría , entonces asigna el "tipo preferido" a la columna de entrada que antes era de tipo desconocido.

    e. Escoge el candidato con el ajuste de tipos más exacto, y el cual ajusta el "tipo preferido" a cada categoría de columna desde el paso anterior. Si hay más de un candidato, o si no hay ninguno, entonces se produce un error.

6.3.1. Ejemplos

6.3.1.1. Función factorial

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

```
tgl=> select int4fac(int2 ’4’);
int4fac
```

```
-----
     24
(1 row)
```

y es de hecho transformado por el analizador a

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

6.3.1.2. Función substring

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

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

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

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

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

```
(1 row)
```

lo que es transformado por el analizador a:

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

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

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

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

realmente se ejecuta como

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

6.4. Resultados de consultas

Evaluación del resultado

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

6.4.1. Ejemplos

6.4.1.1. Almacenamiento de varchar

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

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

6.5. Consultas UNION

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

Evaluación de UNION

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

6.5.1. Ejemplos

6.5.1.1. Tipos sin especificar

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

6.5.1.2. UNION simple

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

6.5.1.3. UNION transpuesto

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

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

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

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

```
(3 rows)
```

Capítulo 7. Índices y claves (keys)

Autor: Escrito por Herouth Maoz (herouth@oumail.openu.ac.il)

Nota del Editor: Este artículo apareció originalmente en la lista de correo, como respuesta a la pregunta: "¿Cual es la diferencia entre las restricciones PRIMARY KEY y UNIQUE?".

```
Asunto: Re: [PREGUNTAS] PRIMARY KEY | UNIQUE

        Cual es la diferencia entre:

              PRIMARY KEY(campos,...) y
              UNIQUE (campos,...)

       - ¿Son sinónimos?
       - Si PRIMARY KEY ya indica una clave (key) única, entonces ¿porqué
 existe otra clase de clave llamada UNIQUE?
```

Una clave primaria es el campo (o los campos) usado para identificar una fila. Por ejemplo, el número de identificación fiscal de una persona.

Una simple combinación única de campos (UNIQUE) no tiene nada que ver con la identificación de la columna. Es simplemente una restricción de integridad. Por ejemplo, yo tengo una colección de enlaces. Cada colección se identifica por medio de un número único, que es la clave primaria. Esta clave se usa en relaciones.

Sin embargo, mi aplicación exige que cada colección tenga también un nombre único. ¿Porqué? Para que un ser humano que quiera modificar una colección también sea capaz de identificarla. Es mucho mas difícil saber, si se tienen dos colecciones llamadas "Ciencia de la Vida", que la que tiene el número 24433 es la que usted necesita y no la que tiene el número 29882.

De esta forma, el usuario selecciona las colecciones por sus nombres. Por lo tanto nos aseguramos que los nombres sean únicos dentro de la base de datos. Sin embargo ninguna otra tabla en la base de datos se refiere a la tabla de colecciones por su nombre. Eso sería bastante ineficiente.

¡Aún mas, a pesar de ser único, el nombre de la colección no define realmente la colección! Por ejemplo, si alguien decidiera cambiar el nombre de la colección de "Ciencia de la Vida" por "Biología", aún seguiría siendo la misma colección, solo que con un nombre diferente. Mientras el nombre sea único no hay problema.

Resumiendo:

- Clave primaria:
  - Usada para identificar la fila y para referirse a ella.
  - Es imposible (o muy difícil) de actualizar.
  - No debe aceptar valores NULL.
- Campos "unique":
  - Se usan como alternativa para acceder una fila.
  - Pueden ser actualizados siempre y cuando mantengan su valor único.
  - Aceptan valores NULL.

En cuanto a la pregunta de ¿por qué no se definen claves no-únicas explícitamente en la sintaxis estándar de SQL? Pues hay que entender que los índices dependen de la implementación específica. SQL no define la implementación, simplemente las relaciones entre los datos y la base de datos. Postgres acepta índices no-únicos, pero los índices usados como claves SQL son siempre únicos.

De esta forma, puede efectuar búsquedas en una tabla por medio de cualquier combinación de columnas, a pesar de que no tenga un índice en esas columnas. Los no índices son sino una ayuda que cada implementación de un RDBMS le ofrece, para permitir que las búsquedas usadas frecuentemente sean hechas de una forma más eficiente. Algunos RDBMS pueden proporcionarle mecanismos adicionales, tales como

el almacenamiento de una clave en la memoria principal. Esos mecanismos tendrán una orden especial, por ejemplo

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

(ésta no es ninguna orden real, sino un ejemplo).

¡De hecho cuando usted crea una clave primaria o una combinación única de campos, la especificación SQL no dice en ninguna parte que sea creado un índice o que la obtención de los datos por medio de la clave sea más eficiente que una búsqueda secuencial!

Así que si usted quiere usar como clave secundaria una combinación de campos que no es única, no tiene que especificar nada - ¡simplemente comience a obtener datos usando esa combinación! Sin embargo, si quiere que la obtención de los datos sea más eficiente, tendrá que optar por los medios que su RDBMS le proporciona - ya sea un índice, la orden MEMSTORE que inventé como ejemplo, o un RDBMS inteligente que cree índices, sin su conocimiento, basándose en el hecho de que usted ha efectuado varias búsquedas con la misma combinación específica de claves... (Aprende con la experiencia).

Capítulo 8. Matrices

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

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

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

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

```
INSERT INTO SAL_EMP
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {}}');

INSERT INTO SAL_EMP
```

```
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"talk", "consult"}, {"meeting"}}');
```

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

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

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

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

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

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

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

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

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

Capítulo 9. Herencia

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

```
CREATE TABLE cities (
    name            text,
    population      float,
    altitude        int     - (in ft)
);

CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

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

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

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

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

+-------+--------+
|name      | altitude |
```

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

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

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

Lo que devuelve lo siguiente:

```
+-------+-------+
|name      | altitude |
+-------+-------+
|Las Vegas | 2174     |
+-------+-------+
|Mariposa  | 1953     |
+-------+-------+
|Madison   | 845      |
+-------+-------+
```

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

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

Multi-Version Concurrency Control (MVCC) es una técnica avanzada para mejorar las prestaciones de una base de datos en un entorno multiusuario. Vadim Mikheev (mailto:vadim@krs.ru) ha proporcionado la implementación para Postgres.

10.1. Introducción

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

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

10.2. Aislamiento transaccional

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

lecturas "sucias"

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

lecturas no repetibles

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

lectura "fantasma"

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

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

Tabla 10-1. Niveles de aislamiento de Postgres

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

Postgres ofrece lectura cursada y niveles de aislamiento serializables.

10.3. Nivel de lectura cursada

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

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

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

10.4. Nivel de aislamiento serializable

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

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

una transacción serializable será parada con el mensaje

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

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

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

10.5. Bloqueos y tablas

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

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

10.5.1. Bloqueos a nivel de tabla

AccessShareLock

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

declaración.

Conflictos con AccessExclusiveLock.

RowShareLock

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

Entra en conflictos con los modos ExclusiveLock y AccessExclusiveLock.

RowExclusiveLock

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

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

ShareLock

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

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

ShareRowExclusiveLock

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

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

ExclusiveLock

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

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

AccessExclusiveLock

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

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

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

10.5.2. Bloqueos a nivel de fila

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

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

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

10.6. Bloqueo e índices

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

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

Indices GiST y R-Tree

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

Indices hash

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

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

Btree

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

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

10.7. Chequeos de consistencia de datos en el nivel de aplicación

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

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

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

Capítulo 11. Configurando su entorno

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

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

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

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

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

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

Si el administrador de su sitio no configuró las cosas como vienen por defecto, quizás tenga que realizar alguna tarea más. Por ejemplo, si el servidor de bases de datos es una máquina remota, necesitará especificar el valor de la variable de entorno PGHOST con el nombre de la máquina que sirve la base de datos. La variable de entorno PGPORT

puede también ser necesaria. La cuestión de fondo es esta; usted intenta arrancar una aplicación y recibe el mensaje de error que dice que no puede conectar con el postmaster. Debería consultar inmediatamente con el administrador de su sitio para asegurarse que su entorno está correctamente configurado.

Capítulo 12. Administración de una Base de Datos

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

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

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

12.1. Creación de Bases de Datos

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

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

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

```
% createdb mibd
```

y para obtener el mismo resultado dentro de psql escriba

```
* CREATE DATABASE mibd;
```

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

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

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

12.2. Ubicaciones Alternativas de las Bases de Datos

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

Las ubicaciones alternativas de bases de datos se crean y son referidas por medio de una variable de estado que da el camino absoluto al lugar donde se almacenará la base de datos. Esta variable de estado debe haber sido definida antes de arrancar el motor y el lugar para donde apunta debe permitir escritura desde la cuenta del administrador postgres. Consulte con el administrador local sobre ubicaciones preconfiguradas para

bases de datos. Se puede usar cualquier nombre de variable válido para indicar locales alternativos, aunque se recomienda usar nombres de variables con el prefijo "PGDATA" para evitar confusiones con otras variables.

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

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

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

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

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

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

y para hacer lo mismo a partir de psql escriba

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

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

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

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

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

12.3. Acceso a una Base de Datos

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

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

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

```
% psql mibd
```

Será saludado con el siguiente mensaje:

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

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

mibd=>
```

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

```
mibd=> \h
```

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

```
mibd=> \g
```

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

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

Para salir de psql y regresar a Unix, escriba

```
mibd=> \q
```

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

12.3.1. Privilegios para Bases de Datos

12.3.2. Privilegios para Tablas

TBD

12.4. Destrucción de una Base de Datos

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

```
% dropdb mibd
```

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

Capítulo 13. Almacenamiento en disco

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

Capítulo 14. Instrucciones SQL

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

ABORT

Nombre

`ABORT` — Aborta la transaccion en curso

Synopsis

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

Inputs

None.

Outputs

`ROLLBACK`

Mensaje devuelto si es completado con exito.

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

Si no hay transacciones en curso actualmente.

Descripcion

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

Notas

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

Utilizacion

Para abortar todos los cambios:

```
ABORT WORK;
```

Compatibilidad

SQL92

Este comando es una extension PostgreSQL presente por razones historicas,

ROLLBACK es el comando SQL92 equivalente.

MODIFICAR GRUPO

Nombre

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

Synopsis

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

Entradas

`nombre`

El nombre del grupo a modificar.

`nombre de usuario`

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

Resultados

`MODIFICAR GRUPO`

Mensaje recibido si la variación fue correcta.

Descripción

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

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

Forma de uso

Añadir usuarios a un grupo:

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

Eliminar un usuario de un grupo

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

Compatibilidad

SQL92

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

MODIFICAR TABLA

Nombre

`MODIFICAR TABLA` — Propiedades de las modificaciones de tablas

Synopsis

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

Entradas

`tabla`

El nombre de un tabla existente para modificarla.

`columna`

Nombre de una columna nueva o ya existente.

`tipo`

Tipo de la nueva columna.

`nueva columna`

Nuevo nombre para una columna ya existente.

`nueva tabla`

Nuevo nombre para la tabla.

Resultados

`MODIFICAR`

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

`ERROR`

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

Descripción

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

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

Notas

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

“*” siguiendo a un nombre de una talba indica que la orden debe ejecutarse sobre esa tabla y todas las tablas que esten bajo ella en la jerarquía subsequente; por defecto, el atributo no será añadido a o renombrado en ninguna de las subclases. Esto siempre se debe hacer cuando se añade o modifica un atributo en una superclase. Si no es así, las preguntas en la jerarquía subsequente como

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

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

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

Usted debe ser el creador de la clase para poder cambiar el esquema. Renombrar cualquier parte de un esquema del catálogo de un sistema no está permitido. La *Guía*

del Usuario de PostgreSQL tiene más información de las herencias subsequentes.

Dirígase a**CREAR TABLA** para una descipción más amplia de los argumentos válidos.

Moda de uso

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

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

Para renombrar una columna existente:

```
MODIFICAR TABLA distribuidores RENOMBRAR COLUMNA direcciones A ciu-
dad;
```

Para renombrar una tabla existente:

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

Compatibilidad

SQL92

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

COLUMNA está en sumisión completa.

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

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

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

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

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

```
MODIFICAR TABLA tabla ELIMINAR [ COLUMNA ] columna { RESTRICT | CAS-
CADE }
```

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

```
CREAR TABLA temp COMO SELECCIONAR did, ciudad DESDE distribui-
dores;
```

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

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

MODIFICAR USUARIO

Nombre

`MODIFICAR USUARIO` — Modificar la información de la cuenta de usuario

Synopsis

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

Entradas

`nombre de usuario`

El nombre del usuario cuyos detalles van a ser modificados.

`palabra clave`

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

CREATEDB
NOCREATEDB

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

CREATEUSER
NOCREATEUSER

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

`abstime`

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

Resultados

`MODIFICAR USUARIO`

Mensaje recibido si la modificación es correcta.

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

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

Descripción

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

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

Modo de uso

Cambiar la palabra clave de un usuario:

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

Cambiar la validez de un usuario hasta la fecha

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

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

más que el UTC

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

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

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

Compatibilidad

SQL92

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

BEGIN

Nombre

`BEGIN` — Comienza una transaccion en modo encadenado

Synopsis

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

Inputs

WORK
TRANSACTION

Palabras clave opcionales. No tienen efecto.

Outputs

`BEGIN`

esto significa que una nueva transaccion ha sido comenzada.

`NOTICE: BEGIN: already a transaction in progress`

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

Descripcion

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

multiples estados dentro de una transaccion tambien es requerida para la consistencia cuando se cambian muchas tablas relacionadas.

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

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

Notas

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

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

Utilizacion

Para comenzar una transaccion de usuario:

```
BEGIN WORK;
```

Compatibilidad

SQL92

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

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

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

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

CLOSE

Nombre

`CLOSE` — Cierra un cursor

Synopsis

```
CLOSE cursor
```

Inputs

`cursor`

El nmbre de un cursor abierto a cerrar.

Outputs

`CLOSE`

Mensaje devuelto si el cursor es cerrado exitosamente.

`NOTICE PerformPortalClose: portal "`*`cursor`*`" not found`

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

Descripcion

CLOSE libera los recuroso asociados con un cursor abierto. Despues de que sea cerrado el cursor, no se permiten operaciones subsiguientes en el. Un cursor deberia ser cerrado cuando no va a ser necesitado.

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

Notas

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

Utilizacion

Cerrar el cursor liahoma:

```
CLOSE liahona;
```

Compatibilidad

SQL92

CLOSE es totalmente compatible con SQL92.

CLUSTER

Nombre

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

Synopsis

```
CLUSTER indexname ON table
```

Entradas

`Nombre del indice`

El nombre de un indice.

`table`

El nombre de una tabla.

Salidas

`CLUSTER`

El agrupamiento se hizo exitosamente.

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

* *Esto no esta documentado en ningun lugar. Parece que no es posible agrupar una tabla que es heredada.*

`ERROR: Relation table does not exist!`

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

Descripccion

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

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

Notas

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

En los casos en que accedes a una lineas solas aleatoreamente dentro de una tlaba, el orden actual de los datos en el global de la tabla no es importante. Sin embargo, si tienes tendencia a acceder a algunos datos mas que a otros, y hay un indice que los agrupa, te beneficiaras del uso de **CLUSTER**.

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

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

Otra manera para agrupar datos es utilizar

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

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

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

Utilizacion

Agrupamiento de la relacion empleados basandose en su atributo salario

```
CLUSTER emp_ind ON emp;
```

Compatibilidad

SQL92

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

COMMIT

Nombre

`COMMIT` — Realiza la transacción actual

Synopsis

```
COMMIT [ WORK | TRANSACTION ]
```

Inputs

WORK
TRANSACTION

Palabra clave opcional. No tiene efecto.

Outputs

`COMMIT`

Mensaje devuelto si la transacción se realiza con exito.

```
NOTICE: COMMIT: no transaction in progress
```

Si no hay transacciones en progreso.

Description

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

Notes

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

Use *ROLLBACK* para abortar una transacción.

Usage

Para hacer todos los cambios permanentes:

```
COMMIT WORK;
```

Compatibilidad

SQL92

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

COPY

Nombre

`COPY` — Copia datos entre ficheros y tablas

Synopsis

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

Inputs

BINARY

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

`table`

El nombre de una tabla existente.

WITH OIDS

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

`filename`

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

`stdin`

Especifica que la entrada viene de un conducto o terminal.

`stdout`

Especifica que la salida va a un conducto o terminal.

`delimiter`

UN caracter que delimita los campos de entrada o salida.

`null print`

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

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

Outputs

`COPY`

La copia se completó satisfactoriamente.

`ERROR:` *`reason`*

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

Descripción

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

Notes

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

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

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

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

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

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

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

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

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

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

máquina del usuario.

File Formats

Text Format

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

El formato para cada instancia es

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

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

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

El caracter barra invertida tiene otros significados especiales. Un caracter barra invertida literal se representa como dos barras consecutivas ("\\". El caracter tabulador se representa con una barra invertida y un tabulador. EL caracter fin de linea se representa como una barra invertida y un fin de linea. Cuando se cargan datos de texto no generados por Postgres necesitará convertir el caracter barra invertida en un par de

barras para asegurar que se carguen adecuadamente. (La secuencia "\N" siempre se interpretará como una barra invertida y un caracter "N", por compatibilidad. La solución más general es "\\N".)

Binary Format

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

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

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

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

Alineación de datos binarios

Sobre equipos Sun-3s, los atributos de 2 bytes se alinean en grupos de cuatro bytes. Los atributos de caracteres se alinean en grupos de un solo byte. En la mayoría de las otras máquinas, todos los atributos mayores de un byte se alinean en grupos de cuatro bytes. Nótese que los atributos de longituv variable vienen precedidos de la longitud del

atributo; las matrices son simplemente cadenas continuas del elemento tipo de la matriz.

Usage

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

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

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

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

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

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

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

```
355  \0  \0  \0 027  \0  \0  \0 001  \0  \0  \0 002  \0  \0  \0
```

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

Compatibility

SQL92

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

CREATE AGGREGATE

Nombre

`CREATE AGGREGATE` — Define una nueva función de agregado

Synopsis

`CREATE AGGREGATE` *`name`* `[ AS ] ( BASETYPE =` *`data_type`*

```
[ , SFUNC1 = sfunc1, STYPE1 = sfunc1_return_type ]
[ , SFUNC2 = sfunc2, STYPE2 = sfunc2_return_type ]
[ , FINALFUNC = ffunc ]
[ , INITCOND1 = initial_condition1 ]
[ , INITCOND2 = initial_condition2 ] )
```

Entradas

name

El nombre de la función de agregado a crear.

data_type

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

sfunc1

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

sfunc1_return_type

El tipo devuelto de la primera función de trancisión.

sfunc2

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

sfunc2_return_type

EL tipo devuelto por la segunda función de trancisión.

ffunc

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

initial_condition1

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

initial_condition2

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

Outputs

CREATE

Mensaje devuelto si el comando se completa satifactoriamente.

Description

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

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

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

y una función final de cálculo, *ffunc*:

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

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

Estas funciones de trancisión han de tener las siguinetes propiedades:

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

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

Notes

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

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

Usage

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

Compatibilidad

SQL92

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

CREATE DATABASE

Nombre

`CREATE DATABASE` — Crea una nueva base de datos

Synopsis

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

Inputs

`name`

Le nombre de la base de datos a crear.

`dbpath`

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

Outputs

`CREATE DATABASE`

Mensaje devuelto si la orden se completa satisfactoriamente.

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

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

`ERROR: createdb: database "name" already exists`

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

`ERROR: Single quotes are not allowed in database names.`
`ERROR: Single quotes are not allowed in database paths.`

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

`ERROR: The path 'xxx' is invalid.`

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

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

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

`ERROR: Unable to create database directory 'xxx'.`
`ERROR: Could not initialize database directory.`

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

Description

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

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

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

Notas

CREATE DATABASE es una extensión del lenguaje de Postgres.

Use drop_database para eliminar la base de datos.

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

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

Uso

Para crear una nueva base de datos:

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

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

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

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

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

Compatibilidad

SQL92

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

CREATE FUNCTION

Nombre

`CREATE FUNCTION` — Defines a new function

Synopsis

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


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

Inputs

`name`

The name of a function to create.

`ftype`

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

rtype

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

attribute

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

definition

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

obj_file , *link_symbol*

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

langname

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

Outputs

`CREATE`

This is returned if the command completes successfully.

Description

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

Notes

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

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

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

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

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

A C function cannot return a set of values.

Usage

To create a simple SQL function:

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

       answer
    ----
    1
```

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

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

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

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

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

The C decalaration of the function is:

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

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

return p;
}
```

Compatibility

SQL92

CREATE FUNCTION is a Postgres language extension.

SQL/PSM

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

SQL/PSM **CREATE FUNCTION** has the following syntax:

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

CREATE GROUP

Nombre

`CREATE GROUP` — Crea un grupo nuevo

Synopsis

```
CREATE GROUP name
    [ WITH
```

```
[ SYSID gid ]
[ USER  username [, ...] ] ]
```

Entradas

name

El nombre del grupo.

gid

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

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

username

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

Salidas

`CREATE GROUP`

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

Descripcion

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

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

Uso

Crear un grupo vacio:

```
CREATE GROUP staff
```

Crear un grupo con miembros:

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

Compatibilidad

SQL92

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

CREATE INDEX

Nombre

`CREATE INDEX` — Construir un índice secundario.

Synopsis

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

Entradas

UNIQUE

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

`nombre_indice`

El nombre del índice que se debe crear.

`tabla`

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

nombre_acceso

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

BTREE

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

RTREE

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

HASH

Una implementación de las dispersiones lineales de Litwin.

columna

El nombre de una columna de la tabla.

nombre_operador

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

nombre_función

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

Salidas

`CREATE`

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

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

Se presenta este error si es imposible crear el índice.

Description

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

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

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

En la segunda sintaxis mostrada antes, se definía un índice como resultado de una función definida por el usuario *`nombre_funcion`* aplicada a uno o más atributos de una única clase. Estos *índices funcionales* pueden utilizarse para conseguir un acceso

rápido a datos basados en operadores que normalmente requerirían algún tipo de transformación para aplicarlos a la base de datos.

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

Notas

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

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

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

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

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

Utilice *DROP INDEX* para eliminar un índice.

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

La siguiente lista select devuelve todos los nombres de operador:

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

Usage

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

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

Compatibilidad.

SQL92

CREATE INDEX es una extensión del lenguaje de Postgres.

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

CREATE LANGUAGE

Nombre

`CREATE LANGUAGE` — Define un nuevo lenguaje para funciones

Synopsis

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

Entradas

TRUSTED

`TRUSTED` especifica que el manipulador para el lenguaje es seguro; es decir, que no ofrece a un usuario no privilegiado nuevas funcionalidades sobrepasando las

restricciones de acceso. Si esta palabra es omitida entonces al registrar el lenguaje, sólo usuarios con privilegio de superusuario Postgres podrán utilizar este lenguaje para crear nuevas funciones (como el lenguaje 'C').

langname

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

HANDLER *call_handler*

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

comment

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

Salidas

`CREATE`

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

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

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

Descripción

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

Escritura de manipuladores PL

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

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

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

Es responsabilidad del manipulador de llamadas obtener la entrada `pg_proc` y analizar el argumento y tipos de retorno del procedimiento llamado. La cláusula AS del **CREATE FUNCTION** del procedimiento estará basada en el atributo `prosrc` de la tabla `pg_proc`. Esto puede ser el texto fuente en el lenguaje procedimental mismo (como en PL/Tcl), una ruta a un fichero o cualquier otra cosa que le indique al handler que hacer en detalle.

Notas

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

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

Remítase a la tabla `pg_language` para más información:

```
      Table    = pg_language
+-----------------+-----------------+-----+
|          Field           |          Type            | Length|
+-----------------+-----------------+-----+
| lanname                  | name                     |    32 |
| lancompiler              | text                     |   var |
+-----------------+-----------------+-----+

lanname |lancompiler
-----+---------
internal|n/a
lisp    |/usr/ucb/liszt
C       |/bin/cc
sql     |postgres
```

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

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

Uso

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

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

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

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

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

          retval = ...
     }
```

```
     *isNull = false;
     return retval;
}
```

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

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

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

Compatibilidad

SQL92

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

CREATE OPERATOR

Nombre

`CREATE OPERATOR` — Define un nuevo operador de usuario

Synopsis

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

Entradas

`name`

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

`func_name`

La función utilizada para implementar este operador.

`type1`

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

`type2`

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

com_op

El commutador para este operador.

neg_op

El negador para este operador.

res_proc

La función estimadora de restricción selectiva para este operador.

join_proc

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

HASHES

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

left_sort_op

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

right_sort_op

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

Salidas

CREATE

Mensaje devuelto si el operador es creado con éxito.

Description

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

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

+ - * / < > = ~ ! @ # % ^ & | ‘ ? $:

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

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

Por lo menos uno de LEFTARG o RIGHTARG deben ser definidos. Para operadores binarios, ambos deberían ser definidos. Para operadores unarios por la derecha, solamente LEFTARG debería ser definido, mientras que en operadores unarios por la derecha solamente RIGHTARG debería ser definido.

También, el procedimiento *func_name* debe haber sido previamente definido utilizando **CREATE FUNCTION** y debe se definido para aceptar el número correcto de argumentos (bien uno o dos).

El operador commutador debería ser identificado si existe uno, para que Postgres pudiese invertir el orden de los operandos si lo desea. Por ejemplo, el operador area-menor-que, «<, debería probablemente tener un operador conmutador area-mayor-que»>. De esta forma, el optimizador de consultas podría convertir libremente:

```
"0,0,1,1"::box  »> MYBOXES.description
```

a

```
MYBOXES.description «< "0,0,1,1"::box
```

Esto permite la ejecución de código para utiliar siempre la última representación y simplifica algo el optimizador.

De forma similar, si existe un operador negador entonces debería ser identificado. Supongamos que un operador, area-igual, ===, existe, y también un operador area-no-igual, !==. El negador permite al optimizador simpificar

```
NOT MYBOXES.description === "0,0,1,1"::box
```

a

```
MYBOXES.description !== "0,0,1,1"::box
```

Si el nombre de un operador commutador es suministrado, Postgres lo busca en el catálogo. Si es encontrado e no tiene aún un commutador él mismo, entonces la entrada del commutador es actualizada para tener el recien creado operador como su commutador. Esto se aplica al negador, también.

Esto es para permitir la definición de dos operadores que son commutadores de los negadores de cada uno de los otros. El primer operador debería der definido sin un commutador o negador (como sea apropiado). Cuando el segundo operador es definido, se debe nombrar el primero como el commutador o negador. El primero será actualizado como un efecto lateral. (En Postgres 6.5, esto también funciona para simplemente que ambos operadores se refieran al otro).

Los siguientes tres especificadores están presentes para auxiliar al optimizador de consultas al realizar uniones ("joins"). Postgres siempre puede evaluar una unión (i.e., procesando una cláusula con dos variables de tuplas separadas por un operador que retorno un booleano) por substitución iterativa [WONG76]. Además,Postgres es capaz de utilizar un algoritmo "hash-join" siguiendo las líneas de [SHAP86]; sin embargo, debe saber si esta estrategia es aplicable. Es algoritmo "hash-join" actual es solamente correcto para operadores que representan tests de igualdad; además la igualdad del tipo de dato debe significar igualdad a nivel de bits de la representación del tipo. (Por ejemplo, un tipo de dato que contiene bits no utilizados que no tienen repercusión para tests de igualdad podría no ser usado en el "hash-join"). El indicador HASHES indica al optimizador de consultas que un hash join poude ser utilizado de forma segura por este operador.

De forma parecida, los dos operadores de orden indican al optimizador de consultas si la estrategia mezclar-ordenar es utilizable y que operadores deberían ser utilizados para ordenar las clases de los dos operadores. Los operadores de orden deberían ser suministrados solamente para un operador de igualdad, y deberían referirse a operadores menor-que para los tipos de la parte izquierda y derecha respectivamente.

Si otras estrategias de unión son consideradas prácticas, Postgres cambiará el optimizador en tiempo de ejecución para utilizarlas y requerirán especificación adicional cuando un operador sea definido. Afortunadamente, la comunidad investigadora inventa nuevas estrategias de unión infrecuentemente, y la generalidad añadida de estrategias definidas por el usuario no merece la complejidad resultante.

Las dos últimas piezas de la especificación están presentes para que el optimizador pueda estimar los tamaños de los resultados. Si una cláusula de la forma:

```
MYBOXES.description «< "0,0,1,1"::box
```

está presente in la cualificación, entoncesPostgres puede tener que estimar la fracción de instancias en MYBOXES que satisfacen la cláusula. La función *`res_proc`* debe ser una función registrada (lo que significa que ya está definida utilizando **CREATE FUNCTION**), acepta argumentos del tipo correcto y devuelve un numero en punto flotante. El optimizador simplemente llama a esta función, pasandole el parámetro

"0,0,1,1" y multiplica el resultado por el tamaño de la relación para obtener el deseado numero de instancias estimado.

Cuando ambos operandos del operador contienen variables de instancia, el optimizador debe estimar el tamaño de la unión resultante. La función join_proc retornara otro numero decimal que será multiplicado por las cardinalidades de las dos clases envueltas en el cómputo del tamaño esperado.

La diferencia entre la función

```
my_procedure_1 (MYBOXES.description, "0,0,1,1"::box)
```

y el operador

```
MYBOXES.description === "0,0,1,1"::box
```

es quePostgres intenta optimizar operadores y puede decidir utilizar un índice para restringir el espacio de búsqueda cuando aparecen operadores. Sin embargo, no se intenta optimizar funciones, y son ejecutadas mediante fuerza bruta. Además, las funciones pueden tener cualquier número de argumentos mientras que los operadores están restringidos a uno o dos.

Notes

Refiérase al capítulo sobre operadores en ls *PostgreSQL User's Guide* para más información. Refiérase a **DROP OPERATOR** para borrar operadores definidos por el usuario de una base de datos.

Utilización Usage

El siguiente comando define un nuevo operador, area-igualdad, para el tipo de dato BOX.

```
CREATE OPERATOR === (
   LEFTARG = box,
   RIGHTARG = box,
   PROCEDURE = area_equal_procedure,
   COMMUTATOR = ===,
   NEGATOR = !==,
   RESTRICT = area_restriction_procedure,
   JOIN = area_join_procedure,
   HASHES,
   SORT1 = «<,
   SORT2 = «<
);
```

Compatibility

SQL92

CREATE OPERATOR is a Postgres extension. There is no **CREATE OPERATOR** statement in SQL92.

CREATE RULE

Nombre

`CREATE RULE` — Define una nueva regla

Synopsis

```
CREATE RULE name AS ON event
    TO object [ WHERE condition ]
    DO [ INSTEAD ] [ action | NOTHING ]
```

Inputs

name

El nombre de la regla a crear.

event

Evente puede ser `select`, `update`, `delete` o `insert`.

object

Object puede ser *`table`* o *`table.column`*.

condition

Cualquiera clausula SQL WHERE. `new` o `current` pueden aparecer en lugar de una variable de instancia*** siempre que una variable de instancia es admisible en SQL.

action

Cualquiera clausula SQL. `new` o `current` pueden aparecer en lugar de una variable de instancia*** siempre que una variable de instancia sea admisible en SQL.

Salidas

`CREATE`

Mensage devulte si la regla es creada con éxito.

Description

El Postgres *rule system*permite que una action alternativa sea realizada en updates, inserts o deletes en tablas o clases. Actualmente se utilizan reglas para implementar vistas de tablas.

El significado de una regla es que cuando una instancia individual es accedida, actualizada, insertada o borrada, existe una instancia actual (para consultas, actualizaciones y borrados) y una nueva instancia (para actualizaciones y añadidos). Si el *`event`* especificado en la clausula ON y la *`condition`* especificada en la clausula WHERE son verdaderas para la instancia actual la parte *`action`* de la regla es ejecutada. Antes, sin embargo, los valores de los campos de la instancia actual y/o la nueva instancia son sustituidos por `current.`*`attribute-name`* y `new.`*`attribute-name`*.

La parte *`action`* de la regla se ejecuta con el mismo identificador de comando y transacción que el comando de usuario que causó la activación.

Notas

Es pertinente la precaución con reglas de SQL. Si el mismo nombre de clase o variable de instancia aparece en el *`event`*, la *`condition`* y la parte *`action`* de la regla, son considerados todos diferentes tuplas. De forma más precisa, `new` y `current` son las únicas tuplas que son compartidas entre cláusulas. Por ejemplo, las siguientes dos

reglas tienen la misma semántica.

```
ON UPDATE TO emp.salary WHERE emp.name = "Joe"
    DO UPDATE emp ( ... ) WHERE ...


ON UPDATE TO emp-1.salary WHERE emp-2.name = "Joe"
    DO UPDATE emp-3 ( ... ) WHERE ...
```

Cada regla puede tener el tag opcional INSTEAD. Sin este tag, la *`action`* sera realizada en adición al comando de usuario cuando el *`event`* en la parte *`condition`* de la regla aparezcan. Alternativamente, la parte *`action`* será realizada en lugar del comando del usuario. En este último caso la instead of the user command. In this later case, the *`action`* puede ser la palabra clave `NOTHING`.

Cuando se elige entre los sistemas de reescritura y reglas de instancia para una aplicación particular de una regla, recuerdese que en el sistema de reescritura, `current` se refiere a la relación y algunos cualificadores mientras que en el sistema de instancias se refiere a una instancia (tupla).

Es muy importante notar que el sistema de reescritura nunca detectará ni procesará reglas circulares. Por ejemplo, aunque cada una de las siguientes dos reglas con aceptadas por Postgres, el comando de recogida causará la caída de Postgres :

Ejemplo 14-1. Ejemplo de combinación circular de regals.

```
CREATE RULE bad_rule_combination_1 AS
    ON SELECT TO emp
    DO INSTEAD SELECT TO toyemp;

CREATE RULE bad_rule_combination_2 AS
    ON SELECT TO toyemp
    DO INSTEAD SELECT TO emp;
```

Este intento de obtención de datos desde EMP provocará la caída de Postgres.

```
SELECT * FROM emp;
```

Es necesario tener permiso de definición de reglas en una clase para poder definir una regla en el. Se debe utilizar el comando **GRANT** y **REVOKE** para modificar estos permisos.

El objeto en una regla SQL no puede ser unna referencia a un array y no puede tener parámetros.

Aparte del campo "oid", los atributos del sistema no pueden ser referenciados en ningún lugar en una regla. Entre otras cosas esto significa que las funciones de instancias (por ejemplo ,`foo(emp)` donde `emp` es una clase) no pueden ser llamadas en ningún lugar dentro de una regla.

El sistema almacena el texto de la regla y los planes de consulta como atributos de texto. Esto implica que la creación de reglas puede fallar si la regla más sus varias internas representaciones exceden algún valor que es del orden de una página.

Uso

Hacer que Sam obtenga el mismo ajuste de salario que Joe:

```
CREATE RULE example_1 AS
    ON UPDATE emp.salary WHERE current.name = "Joe"
    DO UPDATE emp (salary = new.salary)
    WHERE emp.name = "Sam";
```

Al mismo tiempo que Joe recive un ajuste de salario, el evento será verdadero y la instancia actual de Joe y la nueva instancia propuesta están disponibles para las rutinas de ejecución. Por lo tanto, este nuevo salario es sustituido en la parte de acción de la regla que es subsiguientemente ejecutada. Esto propaga el salario de Joe a Sam.

Hacer que Bill obtenga el salario de Joe cuando es accedido:

```
CREATE RULE example_2 AS
    ON SELECT TO EMP.salary
    WHERE current.name = "Bill"
    DO INSTEAD
    SELECT (emp.salary) from emp
        WHERE emp.name = "Joe";
```

Denegar a Joe el acceso al salario de empleados en el departamento de calzado (`current_user` devuelve el nombre del usuario actual):

```
CREATE RULE example_3 AS
    ON SELECT TO emp.salary
    WHERE current.dept = "shoe" AND current_user = "Joe"
    DO INSTEAD NOTHING;
```

Crear una vista de empleados trabajando en el departamente de juguetes.

```
CREATE toyemp(name = char16, salary = int4);

CREATE RULE example_4 AS
    ON SELECT TO toyemp
    DO INSTEAD
    SELECT (emp.name, emp.salary) FROM emp
        WHERE emp.dept = "toy";
```

Todos los nuevos empleados deben hacer 5.000 o menos.

```
CREATE RULE example_5 AS
```

```
ON INERT TO emp WHERE new.salary > 5000
DO UPDATE NEWSET salary = 5000;
```

Compatibility

SQL92

El comando **CREATE RULE** es una extensión de Postgres No existe la sentencia **CREATE RULE** en SQL92.

CREATE SEQUENCE

Nombre

`CREATE SEQUENCE` — Crea una nueva secuencia de generador de numeros

Synopsis

```
CREATE SEQUENCE seqname [ INCREMENT increment ]
    [ MINVALUE minvalue ] [ MAXVALUE maxvalue ]
    [ START start ] [ CACHE cache ] [ CYCLE ]
```

Entradas

seqname

El nombre de una secuencia que sera creada.

increment

La clausula `INCREMENT` *increment* es opcional. Un valor positivo hara una secuencia ascendente, uno negativo hara una secuencia descendente. El valor por omision es uno (1).

minvalue

La clausula opcional `MINVALUE` *minvalue* determina el valor minimo que una secuencia puede generar. El valor por omision es 1 y -2147483647 para secuencias ascendentes y descendentes, respectivamente.

maxvalue

Utilice la clausula opcional `MAXVALUE` *maxvalue* para determinar el valor maximo para una secuencia. Por omision son 2147483647 y -1 para secuencias ascendentes y descendentes, respectivamente.

start

La clausula opcional `START` *start* habilita la secuencia para que comience en cualquier lugar. El valor de inicio por omision es *minvalue* para secuencias ascendentes y *maxvalue* para las descendentes.

cache

La opcion `CACHE` *cache* permite que los numeros de la secuencia sean alojados (preallocated) y almacenados en memoria para un acceso mas rapido. El valor minimo es 1 (solo se puede generar un valor cada vez, i.e. sin cache) y es tambien el valor por omision.

CYCLE

La palabra clave (keyword) CYCLE puede ser utulizada para permitir a la secuencia continuar cuando el valor de *`maxvalue`* o el de *`minvalue`* ha sido alcanzado por una secuencia ascendente o descendente respectivamente. Si el limite es alcanzado, el siguiente numero generado sera cualquiera que para *`minvalue`* o *`maxvalue`* sea tomado como apropiado.

Outputs

`CREATE`

Mensaje devuelto si el comando es ejecutado con exito.

`ERROR: Relation 'seqname' already exists`

Si la secuencia especificada ya existe.

`ERROR: DefineSequence: MINVALUE (start) can't be >= MAXVALUE (max)`

Si el valor de inicio especificado esta fuera de rango.

`ERROR: DefineSequence: START value (start) can't be < MINVALUE (min)`

Si el valor de inicio especificado esta fuera de rango.

`ERROR: DefineSequence: MINVALUE (min) can't be >= MAXVALUE (max)`

Si el valor minimo y maximo son inconsistentes.

Descripcion

CREATE SEQUENCE introducira una nueva secuencia generadora de numeros dentro de la actual base de datos. Esto implica la creacion e inicializacion de una nueva tabla de una linea con el nombre *`seqname`*. La secuencia generadora sera propiedad del usuario que ejecuta el comando.

Depsues de que se crea una secuencia, puede utilizar la funcion `nextval(`*`seqname`*`)` par aobtener una nuevo numero de la secuencia. La funcion `currval(`*`'seqname'`*`)` puede ser utilizada para determinar el numerp devuelto por la ultima llamada a `nextval(`*`seqname`*`)` desde la secuencia especificada en la sesion en curso. La funcion `setval(`*`'seqname', newvalue`*`)` puede ser utilizada para configurar el valor actual de la secuencia especificada. La siguiente llamada a `nextval(`*`seqname`*`)` devolvera el valor dado mas la secuencia de incremento.

Utilice una consulta (query) como

```
SELECT * FROM sequence_name;
```

para obtener los parametros de una secuencia. Aparte de obtener los parametros originales, puede utilizar

```
SELECT last_value FROM sequence_name;
```

para obtener el ultimo valor asignado por cualquier proceso en el servidor (backend). to obtain the last value allocated by any backend. parametros que puedes utilizar

Se utliza bajo nivel de bloque para habilitar multiples llamadas simultaneas a un generador.

Atención

Se pueden obtener resultados inesperados si una configuracion de cache mayor que uno es utilizada por un objeto secuencia que sera usado concurrentemente por multiples procesos en el servidor (backends). Cada proceso en el servidor (backend)asignara valores de secuencia "cache" succesivas durante un acceso al objeto secuencia e incrementara el ultimo valor (last_value) del objeto secuencia en conformidad con esto. De este modo, el siguiente uso de cache-1 de nextval dentro de ese proceso en el servidor (backend) devolvera simplemente los valores asignados sin tocar el objeto compartido. Asi pues, los numeros asignados pero no utilizados en la sesion en curso se perderan. Mas aun, aunque se garantice por multiples procesos en el servidor (backends) la asignacion de distintos valores de secuencia, los valores pueden ser generados fuera de secuencia cuando los procesos en el servidor (backends) son tenidos en cuenta. (Por ejemplo, con una configuracion de cache de 10, el proceso A puede reservar valores 1..10 y devolver un nextval=1, entonces el proceso B puede reservar valores 11..20 y devolver un nextval=11 antes de que el proceso A ha generado un nextval=2.) Asi, con una configuracion de cache de uno es seguro asumir que los valores de nextval seran generados secuencialmente; conuna cache configurada mayor que uno solo podrias asumir que los valores de nextvalue seran todos distintos, no que seran todos generados de un modo puramente secuencial. Tambien, last_value reflejara el ultimo valor reservado por cualquier proceso en el servidor (backend), tanto si ya ha sido devuelto por nextval como si no.

Notas

Remitase a estado **DROP SEQUENCE** para eliminar una secuencia.

Cada proceso en el servidor (backend) utiliza su propia cache para almacenar numeros asignados. Los numeros que estan en la cache pero no son utilizado en la sesion en

curso se perderan, dando como resultado "vacios" en la secuencia.

Uso

Crea una secuencia ascendente llamada `serial`, comenzando en 101:

```
CREATE SEQUENCE serial START 101;
```

Seleccione el siguiente numero de esta secuencia

```
SELECT NEXTVAL ('serial');

nextval
-----
    114
```

Utilice esta secuencia en una INSERT:

```
INSERT INTO distributors VALUES (NEXTVAL('serial'),'nothing');
```

Configurar el valor de la secuencia despues de un COPY FROM:

```
CREATE FUNCTION distributors_id_max() RETURNS INT4
    AS 'SELECT max(id) FROM distributors'
    LANGUAGE 'sql';
BEGIN;
    COPY distributors FROM 'input_file';
    SELECT setval('serial', distributors_id_max());
END;
```

Compatibilidad

SQL92

CREATE SEQUENCE es una extension de lenguaje Postgres. No hay estado **CREATE SEQUENCE** en SQL92.

CREATE TABLE

Nombre

`CREATE TABLE` — Crea una nueva tabla

Synopsis

```
CREATE [ TEMPORARY | TEMP ] TABLE table (
    column type
    [ NULL | NOT NULL ] [ UNIQUE ] [ DEFAULT value ]
    [column_constraint_clause | PRIMARY KEY } [ ... ] ]
    [, ... ]
    [, PRIMARY KEY ( column [, ...] ) ]
    [, CHECK ( condition ) ]
    [, table_constraint_clause ]
    ) [ INHERITS ( inherited_table [, ...] ) ]
```

Entradas

TEMPORARY

Se crea la tabla sólo para esta sesión, y es eliminada automáticamente con el fin de la sesión. Las tablas permanentes existentes con el mismo nombre no son visibles mientras la table temporal existe.

`table`

El nombre de una nueva clase o tabla a crear.

`column`

El nombre de un campo.

`type`

El tipo del campo. Puede incluir especificadores de array. Consulte la *PostgreSQL User's Guide* para más información sobre tipos y arrays.

DEFAULT *`value`*

Un valor por defecto para el campo. Consulte la cláusula DEFAULT para más información.

`column_constraint_clause`

La cláusula opcional de restricciones (constraint) especifica una lista de restricciones de integridad o comprueba que las nuevas inserciones o actualizaciones deben satisfacer para que la inserción o la actualización tenga éxito. Cada restricción debe evaluarse a una expresión booleana. Aunque SQL92 requiere la *`column_constraint_clause`* para referirse a ese campo sólamente, Postgres permite que múltiples campos sean referenciados dentro de un único campo constraint. Consulte la cláusula constraint para más información.

`table_constraint_clause`

La cláusula opcional CONSTRAINT especifica una lista de restricciones de

integridad que las nuevas inserciones o las actualizaciones deberán satisfacer para que una sentencia insert o update tenga éxito. Cada restricción debe ser evaluada a una expresión booleana. Se pueden referenciar múltiples campos con una única restricción. Sólo se puede definir una única cláusula PRIMARY KEY por tabla; PRIMARY KEY *`column`* (una restricción de tabla) and PRIMARY KEY (una restricción de campo) son mutuamente excluyentes. Consulte la cláusula de restricción de tabla para más información.

INHERITS *`inherited_table`*

La cláusula opcional INHERITS especifica una colección de nombres de tabla de las cuales esta tabla hereda todos los campos. Si algún campo heredado aparece más de una vez, Postgres informa de un error. Postgres permite automáticamente a la tabla creada heredar funciones de las tablas superiores a ella en la jerarquía de herencia.

Aside: La herencia de funciones se realiza siguiendo las convenciones del Common Lisp Object System (CLOS).

Salidas

`CREATE`

Mensaje devuelto si la table se ha creado con éxito.

`ERROR`

Mensaje devuelto si la creación de la tabla falla. Este mensaje viene normalmente acompañado por algún texto explicativo, como: `ERROR: Relation 'table' already exists` que ocurre en tiempo de ejecución, si la tabla especificada ya existe en la base de datos.

`ERROR: DEFAULT: type mismatched`

Si el tipo de datos o el valor por defecto no corresponde al tipo de datos de la definición del campo.

Description

CREATE TABLE introducirá una nueva clase o tabla en la base de datos actual. La tabla será poseída por el usuario que introduce la sentencia.

Cada *`type`* puede ser un tipo simple, un tipo complejo (set) o un tipo array. Cada atributo puede ser especificado para ser no nulo, y puede tener un valor por defecto, especificado por la *Cláusula DEFAULT*.

Nota: Al igual que en la versión 6.0 de Postgres, consistant array dimensions within an attribute are not enforced. Esto cambiará probablemente en las versiones futuras.

La cláusula opcional INHERITS especifica una colección de nombres de clases de los cuales esta clase hereda automáticamente todos los campos. Si cualquier nombre de campo heredado aparece más de una vez, Postgres informa de un error. Postgres permite automáticamente a la clase creada heredar funciones de clases superiores en la jerarquía de herencia. La herencia de funciones se hace siguiendo las convenciones del Common Lisp Object System (CLOS).

Cada nueva tabla o clase *`table`* es creada automáticamente como tipo. Por tanto, una o más instancias de la clase son automáticamente un tipo y pueden ser usadas en otras *MODIFICAR TABLA* sentencias **CREATE TABLE**.

The new table is created as a heap with no initial data. Una tabla no puede tener más de 1600 campos (realmente, esto viene limitado por el hecho que el máximo tamaño de una tupla debe ser menor que 8192 bytes), pero este límite puede ser configurado a un tamaño menor en algunos sitios. Una tabla no puede tener el mismo nombre que una tabla de catálogo del sistema.

Cláusula DEFAULT

`DEFAULT` *`value`*

Entradas

`value`

Los posibles valores para la expresión DEFAULT (valor por defecto) son:

- un literal
- una función de usuario
- una función niladic

Salidas

Ninguna.

Descripción

La cláusula DEFAULT asigna una valor por defecto a un campo (a través de una definición de campo en la sentencia CREATE TABLE). El tipo de dato de un valor por defecto debe corresponder al tipo de dato de la definición del campo.

Una operación de inserción (INSERT) que incluya un campo sin un valor especificado por defecto asignará el valor NULL al campo si no se le proporciona un valor explícito. Si el valor por defecto es un *`literal`* significa que que es un valor constante. Si el valor por defecto es una *`niladic-function`* o una *`user-function`* significa que dicho valor es el de la función especificada en el momento de la inserción.

Hay dos tipos de funciones niladic:

niladic USER

CURRENT_USER / USER

Consulte las funciones CURRENT_USER

SESSION_USER

todavía no soportadas

SYSTEM_USER

todavía no soportadas

niladic datetime

CURRENT_DATE

Consulte las funciones CURRENT_DATE

CURRENT_TIME

Consulte las funciones CURRENT_TIME

CURRENT_TIMESTAMP

Consulte la función CURRENT_TIMESTAMP

En la versión actual (v6.5), Postgres evalúa todas las expresiones por defecto en el momento en que la tabla es definida. Por tanto, las funciones que son "non-cacheable" como `CURRENT_TIMESTAMP` pueden no producir el efecto deseado. Para el caso particular de tipos date/time , se puede trabajar sobre este comportamiento usando “DEFAULT TEXT ’now’” en lugar de “DEFAULT ’now’” o “DEFAULT CURRENT_TIMESTAMP”. Esto fuerza Postgres a considerar la constante como un tipo string y así convertirla al valor timestamp en tiempo de ejecución.

Uso

Para asignar un valor constante como valor por defecto para los campos `did` and `number`, y una cadena al campo `did`:

```
CREATE TABLE video_sales (
    did      VARCHAR(40) DEFAULT ’luso films’,
    number   INTEGER DEFAULT 0,
    total    CASH DEFAULT ’$0.0’
);
```

Para asignar una secuencia existente como valor por defecto para el campo `did`, y un literal para el campo `name`:

```
CREATE TABLE distributors (
    did      DECIMAL(3)  DEFAULT NEXTVAL(’serial’),
```

```
    name     VARCHAR(40) DEFAULT 'luso films'
);
```

Column CONSTRAINT Clause

```
[ CONSTRAINT name ] { [
    NULL | NOT NULL ] | UNIQUE | PRIMARY KEY | CHECK constraint } [, ...]
```

Entradas

name

Un nombre arbitrario dado a la restricción de integridad Si no se especifica *name*, se genera de los nombres de l tabla y campos, lo que asegura unicidad para *name*.

NULL

El campo puede contener valores NULL. Ésta es la opción por defecto.

NOT NULL

El campo no puede contener valores NULL. Esto equivale a la restricción de campo CHECK (*column* NOT NULL).

UNIQUE

El campo debe contener un valor único. En Postgres esto es forzado por medio de la creación implícita de un índice único sobre la tabla.

PRIMARY KEY

Este campo es una clave primaria, lo que implica que la unicidad es forzada por el sistema y que otras tablas pueden confiar en este campo como identificador único para los registros. Consulte PRIMARY KEY para más información.

`constraint`

La definición de la restricción.

Descripción

La cláusula opcional de restricción (CONSTRAINT) especifica restricciones o verifica qué deben cumplir las nuevas inserciones o las actualizaciones para que una operación de inserción o de actualización tenga éxito. Cada restricción debe evaluarse a una expresión booleana. Con una única restricción se pueden referenciar múltiples atributos. El uso de PRIMARY KEY como restricción de tabla es mutuamente incompatible con el uso de PRIMARY KEY como restricción de campo.

Una restricción es una regla con nombre: un objeto SQL que ayuda a definir conjuntos de valores válidos poniendo límites a los resultados de las operaciones INSERT, UPDATE or DELETE sobre una tabla.

Existen dos maneras de definir restricciones de integridad: restricciones de tabla, que veremos más adelante, y restricciones de campo, que pasamos a ver.

Una restricción de campo es una restricción de integridad definida como parte de la definición de campo, y lógicamente se convierte en una restricción de tabla nada más ser creada. Las restricciones de campo disponibles son:

PRIMARY KEY
REFERENCES
UNIQUE
CHECK
NOT NULL

Nota: Postgres todavía no soporta (en su versión 6.5) restricciones de integridad especificadas por REFERENCES. Se acepta la sintaxis pero se ignora la cláusula (disponible, en cambio, a partir de la versión 7.0)

Restricción NOT NULL

```
[ CONSTRAINT name ] NOT NULL
```

La restricción NOT NULL especifica una regla que obliga que un campo contenga únicamente valores no nulos. Ésta es únicamente una restricción de campo, y no se permite como restricción de tabla..

Salidas

status

```
ERROR: ExecAppend: Fail to add null value in not null
attribute "column".
```

Este error ocurre en tiempo de ejecución cuando se intenta insertar un valor nulo en un campo que contiene la restricción NOT NULL.

Descripción

Uso

Definir dos restricciones de campo NOT NULL en la tabla `distributors`, una de las cuales se nombra:

```
CREATE TABLE distributors (
    did      DECIMAL(3) CONSTRAINT no_null NOT NULL,
    name     VARCHAR(40) NOT NULL
);
```

Restricción UNIQUE

```
[ CONSTRAINT name ] UNIQUE
```

Entradas

CONSTRAINT *name*

Una etiqueta arbitraria dada a una restricción.

Salidas

status

```
ERROR: Cannot insert a duplicate key into a unique index.
```

Este error ocurre en tiempo de ejecución si se intenta insertar un valor duplicado en un campo.

Descripción

La restricción UNIQUE especifica una regla que obliga a un grupo de uno o más campos de una tabla a contener valores únicos.

Las definiciones de campo de las columnas especificadas no tienen porqué incluir una restricción NOT NULL constraint para ser incluídos en una restricción UNIQUE. Tener más de un valor nulo en un campo sin la restricción NOT NULL, no viola la restricción UNIQUE. (This deviates from the SQL92 definition, but is a more sensible convention. See the section on compatibility for more details.).

Cada restricción de campo UNIQUE debe nombrar un campo que es distitno del conjunto de campos nombrados por cualquier otra restricción UNIQUE o PRIMARY KEY definifas por la tabla.

Nota: Postgres crea automáticamente un índice único por cada restricción UNIQUE, para asegurar la integridad de los datos. Vea CREATE INDEX para más información.

Uso

Define una restricción de campo UNIQUE para la tabla distributors. Las restricciones de campo UNIQUE solo son definidas sobre un campo de la tabla:

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40) UNIQUE
);
```

lo que equivale a la siguiente restricción de tabla:

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40),
    UNIQUE(name)
);
```

La restricción CHECK

```
[ CONSTRAINT name ] CHECK
    ( condition [, ...] )
```

Entradas

name

Un nombre arbitrario dado a la restricción.

condition

Cualquier expresión condicional válida que se evalue a un resultado boolenano.

Outputs

status

```
ERROR: ExecAppend: rejected due to CHECK constraint
"table_column".
```

Este error ocurre en tiempo de ejecución si alguien intenta insertar un valor ilegal en un campo sujeto a una restricción CHECK.

Descripción

La restricción CHECK especifica una restricción sobre los valores permitidos en un campo. La restricción CHECK también se permite como restricción de tabla.

Las resticciones de campo CHECK de SQL92 sólo pueden ser definidas sobre un campo de la tabla, y solamente pueden referirse a un campo. Postgres no tiene esta restricción.

Restricción PRIMARY KEY

```
[ CONSTRAINT name ] PRIMARY KEY
```

Entradas

CONSTRAINT *name*

Un nombre arbitrario para la restricción.

Salidas

```
ERROR: No se puede insertar un valor duplicado en un índice
único.
```

Esto ocurre en tiempo de ejecución si alguien intenta insertar un valor duplicado en una columna sujeta a una restricción PRIMARY KEY.

Descripción

La restricción de campo PRIMARY KEY especifica que un campo de una tabla solamente puede contener valores únicos (no duplicados) y no nulos . La definición de la columna especificada no tiene que incluir una restricción explícita NOT NULL para ser incluída en una restricción PRIMARY KEY.

Sólo se puede especificar una única clave primaria (PRIMARY KEY) por tabla.

Notas

Postgres crea automáticamente un índice único para asegurar la integridad de los datos. (Vea la sentencia CREATE INDEX)

La restricción de clave primaria (PRIMARY KEY) debe nombrar un conjunto de campos que no sean contenidos por ninguna otra restricción UNIQUE definidos por la misma tabla, ya que produciría una duplicación de índices equivalentes y una sobrecarga adicional en tiempo de ejecución. Sin embargo, Postgres no lo deshabilita específicamente.

Cláusula CONSTRAINT para tablas

```
[ CONSTRAINT name ] { PRIMARY KEY |  UNIQUE } ( column [, ...] )
[ CONSTRAINT name ] CHECK ( constraint )
```

Entradas

CONSTRAINT *name*

Un nombre arbitrario dado a una restricción de integridad.

column [, ...]

El nombre de los campos para los que definimos un índice único y, para la PRIMARY KEY, una restricción NOT NULL.

CHECK (*constraint*)

Una expresión booleana a ser evaluado como la restricción.

Salidas

Las posibles salidas para la cláusula de restricción de tablas son las mismas que para las partes correspondientes de la cláusula restricción de campo.

Descripción

Una restricción de tabla es una restricción de integridad definidad sobre uno o más campos de una tabla base. Las cuatro variaciones de restricciones de tabla son:

UNIQUE
CHECK
PRIMARY KEY
FOREIGN KEY

> **Nota:** Postgres todavía no soporta (en su versión 6.5) las restricciones de integridad FOREIGN KEY. El compilador entiende la sintaxis de FOREIGN KEY, pero solo imprime un aviso e ignora la cláusula. Las claves ajenas pueden ser parcialmente emuladas por medio de triggers (Consulte la sentencia CREATE TRIGGER).

Restricción UNIQUE

```
[ CONSTRAINT name ] UNIQUE ( column [, ...] )
```

Entradas

CONSTRAINT *name*

Un nombre arbitrario dado a una restricción.

column

Un nombre de un campo en una tabla.

Salidas

status

ERROR: Cannot insert a duplicate key into a unique index.

Este error ocurre en tiempo de ejecución si se intenta insertar un valor duplicado en un campo.

Descripción

La restricción UNIQUE especifica una regla en la que un grupo de uno The UNIQUE constraint specifies a rule that a group of one or o más campos diversos de una tabla puede contener solo valores únicos. El comportamiento de la restricción de tabla UNIQUE es el mismo que para la restricción de campo, con la posibilidad adicional de aplicarlo a más de un campo.

Consulte la sección sobre la restricción de campo UNIQUE para más detalles.

Uso

Define una restricción de tabla UNIQUE en la tabla distributors:

```
CREATE TABLE distributors (
    did      DECIMAL(03),
    name     VARCHAR(40),
    UNIQUE(name)
);
```

Restricción PRIMARY KEY

```
[ CONSTRAINT name ] PRIMARY KEY ( column [, ...] )
```

Entradas

CONSTRAINT *name*

Un nombre arbitrario para la restricción.

column [, ...]

Los nombres de uno o más campos en la tabla.

Salidas

status

ERROR: Cannot insert a duplicate key into a unique index.

Esto ocurre en tiempo de ejecución si alguien intenta insertar un valor duplicado en un campo sujeto a una restricción de clave primaria (PRIMARY KEY).

Descripción

La restricción PRIMARY KEY especifica una regla en la que un grupo de uno o más campos de una tabla puede contener sólo valores únicos (no duplicados) y no nulos. Las definiciones de campo de los campos especificados no necesita incluir una restricción NOT NULL para ser incluída en una restricción PRIMARY KEY.

La restricción de tabla PRIMARY KEY es similar a la respectiva restricción de campo, con la posibilidad de extenderla with the additional capability of encompassing multiple columns.

Consulte la sección sobre la restricción de campo PRIMARY KEY para más infomación.

Uso

Crea las tablas films y distributors:

```
CREATE TABLE films (
     code      CHARACTER(5) CONSTRAINT firstkey PRIMARY KEY,
     title     CHARACTER VARYING(40) NOT NULL,
     did       DECIMAL(3) NOT NULL,
     date_prod DATE,
     kind      CHAR(10),
     len       INTERVAL HOUR TO MINUTE
);


CREATE TABLE distributors (
     did      DECIMAL(03) PRIMARY KEY DEFAULT NEXTVAL('serial'),
     name     VARCHAR(40) NOT NULL CHECK (name <> '')
);
```

Crea una tabla con un array de 2 dimensiones:

```
   CREATE TABLE array (
          vector INT[][]
          );
```

Define una restricción de tabla UNIQUE para la tabla films. Las restricciones de tabla UNIQUE pueden ser definidas sobre uno o más campos de la tabla:

```
CREATE TABLE films (
    code      CHAR(5),
    title     VARCHAR(40),
    did       DECIMAL(03),
    date_prod DATE,
    kind      CHAR(10),
    len       INTERVAL HOUR TO MINUTE,
    CONSTRAINT production UNIQUE(date_prod)
);
```

Define una restricción de campo CHECK:

```
CREATE TABLE distributors (
    did      DECIMAL(3) CHECK (did > 100),
    name     VARCHAR(40)
);
```

Define una restricción de tabla CHECK:

```
CREATE TABLE distributors (
    did      DECIMAL(3),
```

```
    name     VARCHAR(40)
    CONSTRAINT con1 CHECK (did > 100 AND name > ”)
);
```

Define una restricción de tabla PRIMARY KEY para la tabla films. Las restricciones de tabla PRIMARY KEY pueden ser definidas sobre uno o más campos de la tabla:

```
CREATE TABLE films (
    code      CHAR(05),
    title     VARCHAR(40),
    did       DECIMAL(03),
    date_prod DATE,
    kind      CHAR(10),
    len       INTERVAL HOUR TO MINUTE,
    CONSTRAINT code_title PRIMARY KEY(code,title)
);
```

Define una restricción de campo PRIMARY KEY para la tabla distributors. Las restricciones de campo PRIMARY KEY solamente se pueden definir para un campo de la tabla (los siguientes dos ejemplos serían equivalentes) :

```
CREATE TABLE distributors (
    did      DECIMAL(03),
    name     CHAR VARYING(40),
    PRIMARY KEY(did)
);
```

```
CREATE TABLE distributors (
    did      DECIMAL(03) PRIMARY KEY,
    name     VARCHAR(40)
);
```

Notas

CREATE TABLE/INHERITS es una extensión al lenguaje de Postgres.

Compatibilidad

SQL92

Además de la tabla temporal visible localmente, SQL92 define una sentencia CREATE GLOBAL TEMPORARY TABLE , y opcionalmente una cláusula ON COMMIT:

```
CREATE GLOBAL TEMPORARY TABLE table ( column type [
    DEFAULT value ] [ CONSTRAINT column_constraint ] [, ...] )
    [ CONSTRAINT table_constraint ] [ ON COMMIT { DELETE | PRESER-
VE } ROWS ]
```

Para tablas temporales, la sentencia CREATE GLOBAL TEMPORARY TABLE nombra una nueva tabla visible a otros clientes y define los campos de la tabla y las restricciones.

La cláusula opcional ON COMMIT de CREATE TEMPORARY TABLE especifica si la tabla temporal debe vaciarse de registros cada vez que se ejecuta un COMMIT o no. Si se omite la cláusula ON COMMIT, se asume la opción por defecto, ON COMMIT DELETE ROWS.

Para crear una tabla temporal:

```
CREATE TEMPORARY TABLE actors (
```

```
    id          DECIMAL(03),
    name        VARCHAR(40),
    CONSTRAINT actor_id CHECK (id < 150)
) ON COMMIT DELETE ROWS;
```

Cláusula UNIQUE

SQL92 especifica algunas posibilidades adicionales para UNIQUE:

Definición de restricción de tabla:

```
[ CONSTRAINT name ] UNIQUE ( column [, ...] )
    [ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
    [ [ NOT ] DEFERRABLE ]
```

Definición de restricción de campo:

```
[ CONSTRAINT name ] UNIQUE
      [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
      [ [ NOT ] DEFERRABLE ]
```

cláusula NULL

La restricción NULL (realmente no es una restricción) es una extensión Postgres a SQL92 incluída por simetría con la cláusula NOT NULL. Como es el valor por defecto para cualquier campo, su presencia es redundante.

```
[ CONSTRAINT name ] NULL
```

cláusula NOT NULL

SQL92 especifica alguna posibildad adicional para NOT NULL:

```
[ CONSTRAINT name ] NOT NULL
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

cláusula CONSTRAINT

SQL92 especifica alguna posibilidad adicional para restricciones, y también define restricciones de assertions y de dominio.

> **Nota:** Postgres todavía no soporta ni dominios ni assertions.

Una assertion es un tipo especial de restricción de integridad y comparte el mismo espacio de nombres con otras restricciones. Sin embargo, una assertion no es necesariamente dependiente de una particular tabla base como son las restricciones, así que SQL-92 proporciona la sentencia CREATE ASSERTION como un método alternativo para definir una restricción:

```
CREATE ASSERTION name CHECK ( condition )
```

Las restricciones de dominio se definen con las sentencias CREATE DOMAIN o ALTER DOMAIN:

Restricción de dominio:

```
[ CONSTRAINT name ] CHECK constraint
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

Definición de restricciones de tabla:

```
[ CONSTRAINT name ] { PRIMARY KEY ( column, ... ) | FOREIGN KEY cons-
traint | UNIQUE constraint | CHECK constraint }
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

Definición de restricciones de campo:

```
[ CONSTRAINT name ] { NOT NULL | PRIMARY KEY | FOREIGN KEY constraint | UNI-
QUE | CHECK constraint }
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

Una definición de restricción de integridad (CONSTRAINT) puede contener un atributo o cláusula DEFERRABLE y /o una cláusula del modo inicial de restricción, en cualquier orden.

NOT DEFERRABLE

significa que la restricción debe ser comprobada después de la ejecución de cada sentencia SQL.

DEFERRABLE

significa que la verificación del cumplimiento de la restricción puede ser aplazado hasta más tarde, pero no más tarde que el final de la actual transacción.

El modo de restricción para cada restricción tiene siempre un valor incial por defecto que se establece para la restricción al principio de la transacción.

INITIALLY IMMEDIATE

significa que, desde el principio de la transacción, la restricción debe ser comprobada después de la ejecución de cada sentencia SQL.

INITIALLY DEFERRED

significa que, como se está al principio de la transacción, la comprobación de la restricción puede ser aplazada hasta más tarde, pero no más tarde que en el final de la actual transacción. O sea, que la restricción puede ser incumplida por alguna sentencia SQL en un punto intermedio de la transacción, pero no al final de la misma.

Cláusula CHECK

SQL92 especifica alguna capacidad adicional para CHECK tanto en la restricciones de tabla como en la de campo.

definición de restricción de tabla:

```
[ CONSTRAINT name ] CHECK ( VALUE condition )
      [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
      [ [ NOT ] DEFERRABLE ]
```

definición de restricción de campo:

```
[ CONSTRAINT name ] CHECK ( VALUE condition )
      [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
      [ [ NOT ] DEFERRABLE ]
```

cláusula PRIMARY KEY

SQL92 especifica algunas posibilidades adicionales para la PRIMARY KEY:

Definición de restricciones de tabla:

```
[ CONSTRAINT name ] PRIMARY KEY ( column [, ...] )
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

Definición de restricciones de campo:

```
[ CONSTRAINT name ] PRIMARY KEY
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

CREATE TABLE AS

Nombre

`CREATE TABLE AS` — Crea una nueva tabla

Synopsis

```
CREATE TABLE table [ (column [, ...] ) ]
     AS select_clause
```

Inputs

`table`

El nombre de una nueva tabla a ser creada.

`column`

El nombre de una columna. Se pueden especificar m£ltiples nombres de columna usando una lista de nombres de columna delimitada por comas.

`select_clause`

Una sentencia de consulta valida. Refiérase a SELECT para hallar una descripción de la sintaxis permitida.

Salidas

Refiérase a **CREATE TABLE** y a **SELECT** para hallar un sumario de posibles mensajes de salida.

Descripción

CREATE TABLE AS permite a una tabla ser creada a partir del contenido de una tabla existente. Equivale en funcionamiento a: *SELECT INTO*, pero quiz s con una sintaxis m s directa.

CREATE TRIGGER

Nombre

`CREATE TRIGGER` — Crea un nuevo disparador

Synopsis

```
CREATE TRIGGER name
  { BEFORE | AFTER } { event
  [OR ...] } ON table
  FOR EACH { ROW | STATEMENT } EXECUTE PROCEDURE
  ER">funcBLE>
  ( arguments )
```

Entradas

name

El nombre de un disparador existente.

table

El nombre de una tabla.

event

Uno entre INSERT, DELETE o UPDATE.

funcname

Una función suministrada por el usuario.

Salidas

CREATE

Se devuelve este mensaje si el disparador se ha creado con éxito.

Descripción

CREATE TRIGGER introducir un nuevo disparador en la base de datos actual. El disparador se asociar con la relación *relname* y ejecutar la función especificada *funcname*.

Se puede especificar que el disparador se dispare de cualquiera de estas dos formas: antes (BEFORE) de que la operación sea intentada en un registro (antes de que las restricciones se comprueben y **INSERT**, **UPDATE** o **DELETE** sean intentados) o después (AFTER) de que la operación haya sido intentada (por ejemplo después de que las restricciones sean comprobadas y de que **INSERT**, **UPDATE** o **DELETE** hayan sido completados). Si el disparador se pone en marcha antes del evento, éste puede saltar la operación para el registro actual o cambiar el registro que estaba insert ndose (sólo para las operaciones **INSERT** y **UPDATE**). Si el disparador se dispara después del evento, todos los cambios, incluyendo la última inserción, actualización o borrado, son "visibles" para el disparador.

Refiérase a los cap¡tulos de SPI y Triggers en la gu¡a *PostgreSQL Programmer's Guide* para m s información.

Notas

CREATE TRIGGER es una extensión del lenguaje Postgres.

Sólo el propietario relacionado puede crear un disparador en esta relación.

Hasta la versión actual (v6.4), las sentencias de disparadores no est n implementadas.

Refiérase a **DROP TRIGGER** para obtener información sobre como borrar disparadores.

Uso

Comprueba si el código de distribuidor especificado existe en la tabla de distribuidores antes de añadir o actualizar una fila en los films de la tabla:

```
CREATE TRIGGER if_dist_exists
    BEFORE INSERT OR UPDATE ON films FOR EACH ROW
    EXECUTE PROCEDURE check_primary_key (’did’, ’distributors’, ’did’);
```

Antes de cancelar un distribuidor o de actualizar su código, borra cada referencia en los films de la tabla:

```
CREATE TRIGGER if_film_exists
    BEFORE DELETE OR UPDATE ON distributors FOR EACH ROW
    EXECUTE PROCEDURE check_foreign_key (1, 'CASCADE', 'did', 'films', 'did'
```

Compatibilidad

SQL92

No hay **CREATE TRIGGER** en SQL92.

El segundo ejemplo explicado ateriormente puede implementarse también usando una restricción de FOREIGN KEY (clave for nea) como en:

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40),
    CONSTRAINT if_film_exists
    FOREIGN KEY(did) REFERENCES films
    ON UPDATE CASCADE ON DELETE CASCADE
);
```

En cualquier caso, las claves foráneas todavía no están implementadas (hasta la versión 6.5) en Postgres.

CREATE TYPE

Nombre

`CREATE TYPE` — Define un nuevo tipo de datos base

Synopsis

```
CREATE TYPE typename
( INPUT = input_function,
  OUTPUT = output_function
  , INTERNALLENGTH = { internallength
   | VARIABLE }
  [ , EXTERNALLENGTH = { externallength
    | VARIABLE } ]
  [ , DEFAULT = "default" ]
  [ , ELEMENT = element ]
  [ , DELIMITER = delimiter ]
  [ , SEND = send_function ]
  [ , RECEIVE = receive_function ]
  [ , PASSEDBYVALUE ] )
```

Entradas

`typename`

El nombre del tipo a ser creado.

internallength

Un valor literal, el cual especifica la longitud interna del nuevo tipo.

externallength

Un valor literal, el cual especifica la longitud externa del nuevo tipo.

input_function

El nombre de una función, creada mediante **CREATE FUNCTION**, la cual convierte los datos desde su forma externa a la forma interna de tipo.

output_function

El nombre de una función, creada mediante CREATE FUNCTION, la cual convierte los datos desde su forma interna a una forma conveniente para ser mostrados.

element

El tipo creado es un array; esto especifica el tipo de los elementos del array.

delimiter

El carácter delimitador para el array.

default

El texto por defecto que se mostrar para indicar "datos no presentes"

send_function

El nombre de una función, creada mediante **CREATE FUNCTION**, la cual convierte los datos de este tipo a una forma adecuada para ser transmitidos a otra máquina.

receive_function

El nombre de una función, creada mediante **CREATE FUNCTION**, la cual convierte los datos de este tipo a una forma adecuada para su transmisión desde

otra máquina a la forma interna.

Salidas

`CREATE`

Mensaje que se devuelve si el tipo ha sido creado con éxito.

Descripción

CREATE TYPE permite al usuario registrar un nuevo tipo de datos de usuario con Postgres para ser usado en la base de datos actual. El usuario que define un tipo se convierte en su propietario. *`typename`* es el nombre de del nuevo tipo y debe ser único dentro de los tipos definidos para esta base de datos.

CREATE TYPE necesita el registro de dos funciones (usando create function) antes de definir el tipo. La representación de un nuevo tipo base está determinada por *`input_function`*, la cual convierte la representación externa del tipo a una representación interna, utilizable por los operadores y funciones definidas por el tipo. Naturalmente *`output_function`* ejecuta la transformación inversa. Ambas funciones, la de entrada y la de salida deben ser declaradas para asumir uno o dos argumentos de tipo "`opaque`".

Los nuevos tipos de datos base pueden ser de longitud fija, en cuyo caso *`internallength`* es un entero positivo, o también pueden ser de longitud variable, en cuyo caso, Postgres asume que el nuevo tipo tiene el mismo formato que el tipo de datos suministrado por Postgres, "`text`". Para indicar que un tipo es de longitud variable, se debe especificar *`internallength`* como `VARIABLE`. La representación externa se especifica de forma similar usando la palabra clave *`externallength`*.

Para indicar que un tipo es un array y para indicar que un tipo tiene elementos de array, indique el tipo del elemento del array usando la palabra clave element. Por ejemplo, para definir un array de enteros de cuatro bytes ("int4"), especifique

```
ELEMENT = int4
```

Para indicar el delimitador a ser usado en arrays de este tipo, *`delimiter`* se puede fijar a un carácter espec¡fico. El delimitador por defecto es la coma (",").

Opcionalmente, hay un valor por defecto disponible en caso de que un usuario quiera algún patrón de bit espec¡fico para expresar "datos no presentes." Especifique el valor por defecto con la palabra clave `DEFAULT`.

* *Cómo el usuario especifica este patrón de bit y lo asocia con el hecho de que los datos no estén presentes>*

Los argumentos opcionales *`send_function`* y *`receive_function`* son usados cuando el programa de aplicación que demanda los servicios Postgres reside en una máquina diferente. En este caso, la máquina en la cual se ejecuta Postgres puede usar un formato para el tipo de datos diferente del usado en la máquina remota. En este caso es conveniente convertir los items de datos a una forma estándar cuando se envien desde el servidor al cliente y convertirlos del formato estándar al espec¡fico de la máquina cuando el servidor recibe los datos desde el cliente. Si estas funciones no est n especificadas, se asume que el formato interno del tipo es aceptable en todas las arquitecturas de máquina releveantes. Por ejemplo, los caracteres simples no se tienen que convertir si se pasa desde un Sun-4 a un DECstation, pero muchos otros tipos s¡.

El flag opcional, `PASSEDBYVALUE`, indica que los operadores y funciones que usan este tipo de datos deben pasar los argumentos preferentemente por valor que por referencia. Dese cuenta de que no pasar¡a por valor tipos cuya representación interna es de más de cuatro bytes.

Para nuevos tipos base, un usuario puede definir operadores, funciones y conjuntos usando las facilidades apropiadas descritas en esta sección.

Tipos de Array

Existen dos funciones generalizadas incorporadas, array_in y array_out para la creación rápida de tipos de array de longitud variable. Estas funciones operan en arrays de cualquier tipo Posgres existente.

Tipos de objetos grandes

Un tipo Posgres regular sólo puede ser de longitud 8192 bytes. Si necesita un tipo mayor debe crear un tipo de objeto grande. El interface para estos tipos está ámpliamente explicado en *The PostgreSQL Programmer's Guide*. La longitud de todos los tipos de objeto grande es siempre VARIABLE.

Ejemplos

Este comando crea el tipo de datos box y después usa el tipo en una definición de clase:

```
CREATE TYPE box (INTERNALLENGTH = 8,
    INPUT = my_procedure_1, OUTPUT = my_procedure_2);
CREATE TABLE myboxes (id INT4, description box);
```

Este comando crea un tipo array de longitud variable con elementos enteros:

```
CREATE TYPE int4array (INPUT = array_in, OUTPUT = array_out,
    INTERNALLENGTH = VARIABLE, ELEMENT = int4);
CREATE TABLE myarrays (id int4, numbers int4array);
```

Este comando crea un tipo de objeto grande y lo usa en una definición de clase:

```
CREATE TYPE bigobj (INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE);
CREATE TABLE big_objs (id int4, obj bigobj);
```

Notas

Los nombres de tipos no pueden empezar por el carácter guión bajo ("_") y sólo pueden tener una longitud de 31 carácteres. Esto es debido a que Postgres crea sin informar un tipo array para cada tipo base con un nombre que consiste en el nombre del tipo base precedido de un guión bajo.

Refiérase a **DROP TYPE** para borrar un tipo existente.

Vea también **CREATE FUNCTION**, **CREATE OPERATOR** y el cap¡tulo de Objetos Grandes, ’Large Objects’, en *PostgreSQL Programmer’s Guide*.

Compatibilidad

SQL3

CREATE TYPE es un elemento de SQL3.

CREAR USUARIO

Nombre

`CREAR USUARIO` — Creando un nuevo usuario de base de datos

Synopsis

```
CREAR USUARIO nombre de usuario
    [ CON
     [ SYSID uid ]
     [ PASSWORD 'palabra clave' ] ]
    [ CREARDB   | NOCREARDB ] [ CREARUSUARIO | NOCREARUSUARIO ]
    [ EN EL GRUPO  nombre de grupo [, ...] ]
    [ VALIDO HASTA  'abstime' ]
```

Entradas

`nombre de usuario`

El nombre del usuario.

`uid`

La orden`SYSID` puede ser usada para escoger el identificador de usuario PostgreSQL del usuario que se esta creando. No es nada necesario que corresponda a los identificadores de usuarios de UNIX , pero algunas personas eligen mantener los números iguales.

Si no se especifica, se usará por defecto el número más alto asignado más uno.

palabra clave

Pide la palabra clave del usuario. Si no va a usar autentificación por palabra clave puede omitir esta opción, de otra manera el usuario no será capza de conectar con el servidor de autentifación de palabras clave. Mire en pg_hba.conf(5) o la Guia del administrador para más detalles de como usar mecanismos de autentificación.

CREATEDB
NOCREATEDB

Estas órdenes definen la capacidad de un usuario para crear bases de datos. Si se especifica CREATEDB, el usuario definido tendrá permiso para crear sus propias bases de datos. Usando NOCREATEDB se denegará a un usuario la capacidad de crear bases de datos. Si se omite esta orden, NOCREATEDB se usa por defecto.

CREATEUSER
NOCREATEUSER

Estas ordenes determinan si a un usuario se le permitirá crear nuevos usuarios. Esta opción harán del usuario un superusuario que podrá pasar por encima de todas las restriciones de acceso. Si se omite esta orden se cogerá la orden de NOCREATEUSER como valor por defecto del usuario.

nombre de grupo

El nombre de un grupo dentro del cual se coloca al usuario como un nuevo miembro.

abstime

La orden VALIDO HASTA pone un valor absoluto a la fecha en la que la palabra clave del usuario pierde su validez. Si se omite esta orden el login valdrá para siempre.

Resultados

`CREAR USUARIO`

Mensaje devuelto si el comando se completa satisfactoriamente.

Descripción

CREAR USUARIO añadirá un nuevo usuario a un ejemplo de PostgreSQL. Vease la Guia del Administrador para más información sobre el manejo de usuarios y la autentificación. Debe ser un superusuario de bases de datos para usar este comando.

Use *MODIFICAR USUARIO* para cambiar la palabra clave y los privilegios de un usuario, y *DROP USER* para eliminar a un usuario. Use **MODIFICAR GRUPO** para añadir o eliminar a un usuario de otros grupos. PostgreSQL viene con un script *createuser* que tiene la misma funcionalidad que este comando (de hecho, llama a este comando) pero puede ser ejecutado desde la linea de comandos.

Modo de uso

Crear un usuario sin palabra clave:

```
CREAR USUARIO jonathan
```

Crear un usuario con palabra clave:

```
CREAR USUARIO david CON PALABRA CLAVE ’jw8s0F4’
```

Crear un usuario con una palabra clave, cuya cuenta es válida hasta el final del 2001. Notese que un segundo dentro del año 2002 la cuenta no es valida:

```
CREAR USUARIO miriam CON PALABRA CLAVE 'jw8s0F4' VALIDA HASTA '1 En 2002'
```

crear una cuenta con la que el usuario pueda crear bases de datos:

```
CREAR USUARIO manuel CON PALABRA CLAVE 'jw8s0F4' CREARDB
```

Compatibilidad

SQL92

No existe la orden **CREATE USER** en SQL92.

CREAR VISTA

Nombre

`CREAR VISTA` — Construir una tabla virtual

Synopsis

`CREAR VISTA` *`vista`* `COMO SELECCIONADO` *`query`*

Entradas

`vista`

El nombre de la vista que se va a crear.

`consulta`

Una consulta en SQL indica las columnas y filas de la vista.

Dirijase a la orden SELECCIONAR para más información sobre los argumentos válidos.

Resultados

`CREADA`

El mensaje recibido si la vista se crea satisfactoriamente.

`ERROR: Relación ’`*`view`*`’ ya existe`

Este error ocurre si la vista especificada ya existe en la base de datos.

`AVISO creado: el nombre atribuido "`*`column`*`" tiene un caracter desconocido`

La vista será creada teniendo una columna con un carácter desconocido si usted no lo especifica. Por ejemplo, el siguiente comando da un error:

```
CREAR VISTA vista COMO SELECCIONADO ’Hola Mundo’
```

mientras que este comando no lo hace:

```
CREAR VISTA vista COMO SELECCIONADO 'Hola Mundo'::texto
```

Descripción

CREAR VISTA definirá una vista de una tabla o class. Esta vista no se materializa físicamente. Especificamente, una consulta reescrita genera automaticamente una regla para mantener las operaciones ejecutadas en la vista.

Notas

Normalmente, las vista son de sólo lectura.

Use la orden **TIRAR VISTA** para deshacerse de la vista.

Modo de uso

Crear una vista conteniendo todas las películas de Comedia:

```
CREAR VISTA clases COMO
    SELECCIONAR *
    DESDE películas
    DONDE clase = 'Comedia';

SELECCIONAR * DESDE clases;

codigo |título                   |did| date_prod|Clase      |Dur
-----+-----------------+--+-------+-------+----
UA502  |Bananas                  |105|13-07-1971|Comedia   | 01:22
C_701  |There's a Girl in my Soup|107|11-06-1970 Comedia   | 01:36
```

Compatibilidad

SQL92

SQL92 especifica algunas capacidades específicas para la orden**CREAR VISTA** :

```
CREAR VISTA view [ columna [, ...] ]
    COMO SELECCIONADO expresión [ COMO nombre de columna ] [, ...]
    DESDE tabla [ DONDE condición ]
    [ CON [ CASCADA | LOCAL ] COMPROBAR OPCION ]
```

Las clausulas opcionales para todos los comandos SQL92 son:

COMPROBAR OPCION

Esta opción es para hacer vistas renovables. Todos los INSERTAR Y RENOVAR en la vista seran ccomprobados para asegurar que los datos satisfacen las condiciones definidas en la tabla. Si no lo cumplen, la renovación no será ejecutada.

LOCAL

Comprobar la integridad de esta vista.

CASCADA

Comprobar la integridad de esta vista y cualquier vista dependiente. CASCADA se asume si ni CASCADA ni LOCAL son especificadas.

DECLARE

Nombre

`DECLARE` — Define un cursor para acceso a una tabla

Synopsis

```
 DECLARE cursorname [ BINARY ] [ INSENSITIVE ] [ SCROLL
] CURSOR FOR query [ FOR { READ
ONLY | UPDATE [ OF column [, ...]
] ]
```

Inputs

`cursorname`

El nombre del cursor a ser usado en subsecuentes operaciones FETCH..

BINARY

Provoca que el cursor traiga datos en formato binario en vez de formato texto.

INSENSITIVE

SQL92 palabra clave indicando que los datos recuperados del cursor no deben ser afectados por actualizaciones desde otros procesos o cursores. Ya que la operación de los cursores ocurre dentro de las transacciones, en Postgres este siempre es el caso. Esta palabra clave no tiene efecto.

SCROLL

SQL92 palabra clave indicando que los datos deben ser recuperados en múltiples filas por cada operación FETCH. Ya que esto es siempre permitido por Postgres esta palabra clave no tiene efecto.

query

Una consulta SQL la cual proveera las filas a ser governadas por el cursor. Referirse al comando SELECT para mayor información acerca de los argumentos válidos.

READ ONLY

SQL92 palabra clave indicando que el cursor sera usado en modo solo lectura. Ya que este es el único modo de acceso de cursor disponible en Postgres esta palabra clave no tiene efecto.

UPDATE

SQL92 palabra clave indicando que el cursor sera usado para actualizar tablas. Ya que la actualización de cursores no esta actualmente soportada en Postgres esta palabra clave provoca un mensaje de error informativo.

column

Columna(s) a ser actualizadas. Ya que la actualización de cursores no esta actualmente soportada en Postgres la clausula UPDATE provoca un mensaje de error informativo.

Outputs

SELECT

El mensaje devuelto si el SELECT es ejecutado exitosamente.

`NOTICE BlankPortalAssignName: portal"`*`cursorname`*`" already exists`

Este error ocurre si *`cursorname`* ya esta declarado.

`ERROR: Named portals may only be used in begin/endtransaction blocks`

Este error ocurre si el cursor no esta declarado dentro de un transaction block.

Description

DECLARE permite a un usuario crear cursores, los cuales pueden ser usados para recuperar un pequeno número de filas a la vez provenientes de una consulta mas extensa. Los cursores pueden devolver datos ya sea en formato de texto o en foemato binario *FETCH*.

Los cursores comunes retornan datos en formato texto, ya sea ASCII u otro esquema de codificacion dependiendo en como el Postgres backend fue creado. Ya que los datos estan guardados nativamente en formato binario, el sistema debe hacer una conversión para producir formato texto. Ademas, los formatos de texto son a menudo mayores en tamano que sus correspondientes en formato binario. Una vez que la información viene en formato texto, la aplicación cliente podria necesitar convertirlos a un formato binario para manipularlos. los cursores BINARY devuelven los datos en una representación binaria nativa.

Como ejemplo, si una consulta devuelve un valor de uno desde una columna integer, usted obtendria un string de ’1’ con un cursor default mientras que con un cursor binario usted obtendria un valor 4-byte igual a un control-A (’^A’).

Los cursores BINARY deben ser usados cuidadosamente. Aplicaciones de usuario tales como psql no son conscientes de los cursores binarios y esperan que los datos vengan en formato texto.

La representación de los string es neutral respecto a la arquitectura, mientra que la representación binaria puede diferir entre diferentes arquitecturas de máquinas y

Postgres no resuelve el ordenamiento de bytes o las cuestines de representacion para los cursores binarios. Por consiguiente, si su máquina cliente y su máquina servidor usa diferentes representaciones (e.g. "big-endian" contra "little-endian"), probablemente usted no deseara sus datos devueltos en formato binario. Sin embargo, los cursores binarios pueden ser un poco más eficientes ya que hay menos overhead debido a la conversión en la transferencias de datos del servidor al cliente.

> **Sugerencia:** Si usted pretende mostrar los datos en ASCII, recuperarlos en ASCII le ahorraran un poco de esfuerzo del lado cliente.

Notes

Los cursores solo estan disponibles en las transacciones. Usar para *BEGIN*, *COMMIT* y *ROLLBACK* para definir un transaction block.

En SQL92 los cursores estan disponibles solo en aplicaciones SQL (ESQL) embebidas. EL Postgres backend no implementa un comando explicito **OPEN cursor** ; un cursor se considera abierto cuando este es declarado. Sin embargo, ecpg, el preprosesador embebido de SQL para Postgres, soporta la convención de cursores SQL92 , incluyendo aquellos que involucran los comandos DECLARE y OPEN.

Uso

Para declarar un cursor:

```
DECLARE liahona CURSOR FOR SELECT * FROMfilms;
```

Compatibilidad

SQL92

SQL92 permite cursores solo en SQL embebido y en modulos. Postgres permite cursores para ser usados en forma interactiva. SQL92 permite cursores embebidos o modulares para actualizar información de la base de datos. Todos los cursores Postgres son de solo lectura. La palabra clave BINARY es una extensión de Postgres.

DELETE

Nombre

`DELETE` — Borra filas de una tabla

Synopsis

```
DELETE FROM table [ WHERE condition ]
```

Inputs

`table`

El nombre de una tabla existente.

condition

Esta es una consulta SQL de selección la cual devuelve las filas a ser borradas.

Referirse al comando SELECT para una mayor descripción de la clausula WHERE.

Outputs

DELETE *count*

Mensaje devuelto si los items son borrados exitosamente. El valor *count* es la cantidad de filas borradas.

Si *count* es 0, ninguna fila fue borrada.

Description

DELETE borra las filas que satisfacen la clausula WHERE de la tabla especificada.

Si la *condicion* (clusula WHERE) esta ausente, el efecto es borrar todas las filas de la tabla. El resultado es una tabla valida, pero vacia.

> **Sugerencia:** *TRUNCATE* es una extensión de Postgres el cual provee un mecanismo más rápido para borrar todas las filas de una tabla.

Para modificar la tabla usted debe poseer acceso de escritura a la misma, asi como acceso de lectura a cualquier tabla cuyos valores son leidos en la *`condicion`*.

Uso

Borra todos los films excepto los musicales:

```
DELETE FROM films WHERE kind <> 'Musical';
SELECT * FROM films;

    code |title                    |did| date_prod|kind      |len
---+-----------------+--+-------+-------+----
UA501|West Side Story          |105|1961-01-03|Musical   | 02:32
TC901|The King and I           |109|1956-08-11|Musical   | 02:13
WD101|Bed Knobs and Broomsticks|111|          |Musical   | 01:57
(3 rows)
```

Borra completamente la tabla `films`:

```
DELETE FROM films;
SELECT * FROM films;

    code|title|did|date_prod|kind|len
---+---+--+------+---+--
(0 rows)
```

Compatibility

SQL92

SQL92 permite un comando DELETE posicionado:

```
DELETE FROM table WHERE CURRENT OF cursor
```

donde *cursor* corresponde a un cursor abierto. En Postgres los cursores interactivos son de solo-lectura.

DROP AGGREGATE

Nombre

DROP AGGREGATE — Elimina la definición de una función agregada

Synopsis

```
DROP AGGREGATE name type
```

Entradas

name

El nombre de una función de agregado existente.

type

El tipo de una función de agregado existente. (Véase la *PostgreSQL User's Guide* para más información sobre los topos de datos).

* *Esto debería ser una referencia cruzada más que un punto de un capítulo*

Salidas

DROP

Mensaje devuelto si el comando se ejecuta satisfactoriamente.

WARN RemoveAggregate: aggregate *'agg'* for *'type'* does not exist

Este mensaje aparece si la función agragada especificada no existe en la base de datos.

Descripción

DROP AGGREGATE eliminará todas las referencias a la definición de una función de agregado existente. Para ejecutar esta orden el usuario actual debe ser el propietario del agregado.

Notas

Use *CREATE AGGREGATE* para crear funciones de agregado.

Uso

Para eliminar el agregado `myavg` de tipo `int4`:

```
DROP AGGREGATE myavg int4;
```

Compatibilidad

SQL92

No existe la sentencia **DROP AGGREGATE** en SQL92; la sentencia es una extensión de lenguaje de Postgres.

DROP DATABASE

Nombre

`DROP DATABASE` — Elimina una base de datos existente

Synopsis

```
DROP DATABASE name
```

Entradas

name

El nombre de una base de datos existente que se desaea eliminar.

Salidas

`DROP DATABASE`

Este mensaje se devuelve si la orden se ejecuta satisfactoriamente.

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

Debe tener el privilegio especial CREATEDB para eliminar bases de datos. Ver *CREAR USUARIO*.

`ERROR: dropdb: cannot be executed on the template database`

La base de datos `template1` no puede ser eliminada. No es conveniente hacerlo.

`ERROR: dropdb: cannot be executed on an open database`

NO puede conectarse a la base de datos que quiere eliminar. En su lugar, ha de conectar a `template1` o cualquier otra base de datos, y ejecutar el comando de nuevo.

`ERROR: dropdb: database 'name' does not exist`

Este mensaje ocurre si la base de datos especificada no existe.

`ERROR: dropdb: database 'name' is not owned by you`

Debe ser el propietario de la base de datos. Ser el propietario normalmente significa que también la ha creado.

`ERROR: dropdb: May not be called in a transaction block.`

Ha de completar primero la transacción en progreso antes de poder ejecutar este comando.

`NOTICE: The database directory 'xxx' could not be removed.`

la base de datos fué eliminada (a menos que haya aparecido otro mensaje de error), pero el directorio donde se almacenaben los datos no pudo ser eliminado. Debe borrarlo manualmente.

Descripción

DROP DATABASE elimina las entradas de catálogo de una base de datos existente y borra el directorio que contiene los datos. Solamente puede ser ejecutado por el propietario de la base de datos (normalmente quien la creó).

Notas

Esta orden no puede ser ejecutada mientras se está conectado a la base de datos objetivo. Por lo tanto, puede ser más conveniente usar el shell script *dropdb*, que emplea este comando.

Véase Refer to *CREATE DATABASE* para más información sobre como crear una base de datos.

Compatibilidad

SQL92

La sentencia **DROP DATABASE** es una extensión de lenguaje de Postgres; no existe ese comando en SQL92.

DROP FUNCTION

Nombre

`DROP FUNCTION` — Elimina una función de usuario escrita en C

Synopsis

```
DROP FUNCTION name ( [ type [, ...] ] )
```

Entradas

`name`

El nombre de una función existente.

`type`

El tipo de los parámetros de la función.

Salidas

`DROP`

Mensaje devuelto si la orden se completa satisfactoriamente.

`WARN RemoveFunction: Function "`*`name`*`" ("`*`types`*`") does not exist`

Este mensaje se obtiene sila función espacificada no existe en la base de datos actual.

Descripción

DROP FUNCTION eliminará las referencias a una función C existente. Para ejecutar esta orden el usuario debe ser el propietario de la función. Los tipos de argumentos de entrada de la función han de especificarse, dado que solo la función con el nombre dado, y los tipos de argumentos daos se eliminará.

Notas

Véase *CREATE FUNCTION* para más información sobre la creación de funciones de agregado.

No se hacen comprobaciones para verificar los tipos de datos, operadores o método de acceso relacionados con la función que ha de eliminarse.

Uso

Esta orden elimina la función raiz cuadrada:

```
DROP FUNCTION sqrt(int4);
```

Compatibilidad

SQL92

DROP FUNCTION es una extensión de lenguaje de Postgres.

SQL/PSM

SQL/PSM es un estandar propuesto para habilitar la extensionalidad de la funciones. La sentencia DROP FUNCTION de SQL/PSM tienen la siguiente sintaxis:

```
DROP [ SPECIFIC ] FUNCTION name { RESTRICT | CASCADE }
```

DROP GROUP

Nombre

`DROP GROUP` — ELimina un grupo

Synopsis

`DROP GROUP` *`name`*

Entradas

`name`

El nombre de un grupo existente.

Salidas

`DROP GROUP`

El mensaje devuelto si es grupo es eliminado satisfactoriamente.

Descripción

DROP GROUP elimina el grupo especificado de la base de datos. Los usuarios del grupo no se eliminan.

Use *CREATE GROUP* para añadir nuevos grupos, y *MODIFICAR GRUPO* para cambiar la pertenencia a un grupo.

Uso

Para eliminar un grupo:

```
DROP GROUP staff;
```

Compatibilidad

SQL92

No existe el comando **DROP GROUP** en SQL92.

DROP INDEX

Nombre

`DROP INDEX` — Elimina un indica de la base de datos

Synopsis

```
DROP INDEX index_name
```

Entradas

`index_name`

El nombre del indice a eliminar.

Salidas

`DROP`

El mensaje devuelto si el indice es eliminado satisfactoriamente.

`ERROR: index "index_name" nonexistent`

Este mensaje tiene lugar si *`index_name`* no es un indice de la base de datos.

Descripción

DROP INDEX elimina un indice existente del sistema de base de datos. Quien ejecute este comando, ha de ser el propietario del índice.

Notas

DROP INDEX es una extensión del lenguaje de Postgres.

Véase Refer to *CREATE INDEX* para más información sobre como crear índices.

Uso

Este comando eliminará el índice `title_idx`:

```
DROP INDEX title_idx;
```

Compatibilidad

SQL92

SQL92 define comandos con los que acceder a una base de datos relacional genérica. Los índices son una característica dependiente de la implementación, por lo que no existe comandos o de finiciones específicos para los índices en el lenguaje SQL92.

DROP LANGUAGE

Nombre

`DROP LANGUAGE` — Elimina un lengueja procedural definido por el usuario

Synopsis

```
DROP PROCEDURAL LANGUAGE 'name'
```

Entradas

`name`

El nombre de un lenguaje procedural existente.

Salidas

`DROP`

Este mensaje es devuelto si el lenguaje es eliminado satisfactoriamente.

`ERROR: Language "name" doesn't exist`

Este mensaje tiene lugar si el lenguaje llamado *`name`* no se encuentra en la base de datos.

Descripción

DROP PROCEDURAL LANGUAGE eliminará la definición del lenuaje procedural llamdo *name*, previmente resgistrado.

Notas

La sentencia **DROP PROCEDURAL LANGUAGE** es una extensión de lenguaje de Postgres.

Véase Refer to*CREATE LANGUAGE* para más información sobre como crear lenguajes procedurales.

No se realiza ninguna comprobación acerca de si existen funciones o procedimientos desencadenados por eventos escritos en este lenguaje. Para re-habilitarlos sin tener que eliminar y recrear todas las funciones, el tributo pg_proc's prolang de las funciones ha de ser ajustado para el nuevo identificador de objeto de la entrada pg_language del lenguaje procedural nuevamente creado.

Uso

Este comando elimina el lenguaje PL/Sample:

```
DROP PROCEDURAL LANGUAGE 'plsample';
```

Compatibilidad

SQL92

No existe el comando **DROP PROCEDURAL LANGUAGE** en SQL92.

DROP OPERATOR

Nombre

`DROP OPERATOR` — Quita un operador de la base de datos

Synopsis

```
DROP OPERATOR id ( type | NONE [,...] )
```

Entradas

`id`

El identificador de un operador existente.

`type`

El tipo de los parámetros de la función.

Salidas

`DROP`

Mensaje devuelto si la operación es exitosa.

`ERROR: RemoveOperator: binary operator 'oper' taking 'type' and 'type2' does not exist`

Este mensaje se muestra si el operador binario especificado no existe.

`ERROR: RemoveOperator: left unary operator 'oper' taking 'type' does not exist`

Este mensaje se muestra si el operador unario izquierdo especificado no existe.

`ERROR: RemoveOperator: right unary operator 'oper' taking 'type' does not exist`

Este mensaje se muestra si el operador unario derecho especificado no existe.

Description

DROP OPERATOR quita un operador de la base de datos. Para ejecutar este comando usted debe ser el propietario del operador.

La calidad de derecho o izquierdo de un operador unario izquierdo o derecho, respectivamente, puede ser especificada como `NONE`.

Notas

La declaración **DROP OPERATOR** es una extensión de lenguaje de Postgres.

Consulte *CREATE OPERATOR* por información sobre cómo crear operadores.

Es responsabilidad del usuario remover cualquier método de acceso y clases de operador que dependan del operador que se quitó.

Utilización

Quita el operador de potencia `a^n` para `int4`:

```
DROP OPERATOR ^ (int4, int4);
```

Quita el operador unario izquierdo de negación (`b !`) para expresiones booleanas:

```
DROP OPERATOR ! (none, bool);
```

Quita el operador unario derecho de factorial (`! i`) para `int4`:

```
DROP OPERATOR ! (int4, none);
```

Compatibilidad

SQL92

No existe un comando **DROP OPERATOR** en SQL92.

DROP RULE

Nombre

`DROP RULE` — Quita una regla existente de la base de datos

Synopsis

`DROP RULE` *`name`*

Entradas

`name`

El nombre de una regla existente para quitar.

Salidas

`DROP`

Mensaje devuelto en caso de que la operación sea exitosa.

`ERROR: RewriteGetRuleEventRel: rule "`*`name`*`" not found`

Este mensaje se muestra si la regla especificada no existe.

Descripción

DROP RULE quita una regla del sistema de reglas de Postgres especificado. Postgres dejaraá de aplicarla inmediatamente y quitarára su definición de los catálogos del sistema.

Notas

La declaración **DROP RULE** es una extensión de lenguaje de Postgres.

Consulte **CREATE RULE** para información sobre cómo crear reglas.

Una vez que se quita una regla, el acceso a la información histórica que la regla haya escrito puede desaparecer.

Utilización

Para quitar la regla de reescritura `newrule`:

```
DROP RULE newrule;
```

Compatibilidad

SQL92

No existe **DROP RULE** en SQL92.

DROP SEQUENCE

Nombre

`DROP SEQUENCE` — Quita una secuencia existente

Synopsis

`DROP SEQUENCE` *`name`* `[, ...]`

Entradas

`name`

El nombre de una secuencia.

Salidas

`DROP`

Mensaje devuelto si la secuencia se elimina exitosamente.

`WARN: Relation "`*`name`*`" does not exist.`

Este mensaje se muestra si la secuencia especificada no existe.

Descripción

DROP SEQUENCE quita una secuencia generadora de números de la base de datos. Con la actual implementación de las secuencias como tablas especiales, trabaja igual que la declaración **DROP TABLE**.

Notas

La declaración **DROP SEQUENCE** es una extensión de lenguaje de Postgres.

Consulte la declaración **CREATE SEQUENCE** para obtener información sobre cómo crear una secuencia.

Utilización

Para quitar la secuencia `serial` de la base de datos:

```
DROP SEQUENCE serial;
```

Compatibilidad

SQL92

No existe **DROP SEQUENCE** en SQL92.

DROP TABLE

Nombre

`DROP TABLE [Eliminar Tabla]` — Elimina tablas de una base de datos

Synopsis

`DROP TABLE` *`nombre`* `[, ...]`

Entradas

`nombre`

El nombre de una tabla vista existente para eliminarla.

Salidas

`DROP`

El mensaje devuelto si el comando concluyo exitosamente.

`ERROR Relation "`*`nombre`*`" Does Not Exist!`

Si la tabla o vista especificada no existe en la base de datos.

Descripción

DROP TABLE elimina tablas y vistas de una base de datos. Solo su propietario (owner) puede destruir una tabla o vista. Una tabla puede ser vaciada de sus filas, pero no destruida, usando **DELETE**.

Si una tabla a ser destruida tiene un índice secundario, este debe ser removido primero. La remoción de solo un índice secundario no afecta el contenido de la tabla subyacente.

Notas

Consultar en **CREATE TABLE** y **ALTER TABLE** para información sobre como crear o modificar tablas.

Uso

Para destruir dos tablas, `cintas` y **distribuidores**:

```
DROP TABLE cintas, distribuidores;
```

Compatibilidad

SQL92

SQL92 especifica algunas capacidades adicionales a DROP TABLE:

```
DROP TABLE table { RESTRICT | CASCADE }
```

RESTRICT

Asegura que solo una tabla sin vistas dependientes o restricciones de integridad pueda ser destruida.

CASCADE

Cualquier vista o restricción de integridad sería tambien eliminada.

Sugerencia: Por el momento, para eliminar una vista dependiente se debe eliminar esta explicitamente.

DROP TRIGGER

Nombre

`DROP TRIGGER` — Borra la definición de un disparador

Synopsis

`DROP TRIGGER` *`nombre`* `ON` *`tabla`*

Entradas

`nombre`

El nombre de un disparador existente.

`tabla`

El nombre de una tabla.

Salidas

`DROP`

Mensaje devuelto si el disparador se borró correctamente.

`ERROR: DropTrigger: there is no trigger` *`name`* `on relation "`*`table`*`"`

Este mensaje se da cuando el disparador especificado no existe.

Descripción

DROP TRIGGER borrará todas las referencias existentes a la definición de un disparador. Para poder ejecutar este comando el usuario actual debe ser el propietario del disparador.

Notas

DROP TRIGGER es una extensión del lenguaje de Postgres.

Consulte **CREATE TRIGGER** para obtener información a cerca de cómo crear disparadores (triggers).

Utilización

Destruye el disparador `if_dist_exists` en la tabla `films`:

```
DROP TRIGGER if_dist_exists ON films;
```

Compatibilidad

SQL92

No existe ninguna declaración **DROP TRIGGER** en SQL92.

DROP TYPE

Nombre

`DROP TYPE` — Retira un tipo, definido por el usuario, de los catálogos del sistema

Synopsis

```
DROP TYPE tipo
```

Entradas

tipo

El nombre del tipo catalogado.

Salidas

`DROP`

El mensaje que se obtiene si el comando ha sido ejecutado con éxito.

`ERROR: RemoveType: type 'tipo' does not exist`

Este mensaje ocurre si el tipo dado no ha sido encontrado.

Descripción

DROP TYPE retira un tipo, definido por el usuario, de los catálogos del sistema.

Un tipo puede ser retirado únicamente por su dueño.

Notas

La cláusula DROP TYPE es una extensión del lenguaje Postgres.

Consulte el comando **CREATE TYPE** para obtener información sobre como crear tipos.

Es responsabilidad del autor retirar cualquier operador, función, agregado, método de acceso, subtipo y clase que usen el tipo que ha sido borrado.

Si se retira un tipo predefinido, el comportamiento del servidor será impredecible.

Uso

Para retirar el tipo `caja`:

```
DROP TYPE caja;
```

Compatibilidad

SQL3

DROP TYPE es una cláusula de SQL3.

DROP USER

Nombre

`DROP USER` — Retira un usuario

Synopsis

`DROP USER` *`nombre`*

Entradas

`nombre`

El nombre de un usuario existente.

Salidas

`DROP USER`

El mensaje que se obtiene si el usuario ha sido retirado con éxito.

`ERROR: DROP USER: user "`*`nombre`*`" does not exist`

Este mensaje ocurre si no ha sido encontrado el usuario dado.

`DROP USER: user "`*`nombre`*`" owns database "`*`base_datos`*`", cannot be removed`

Deberá eliminar primero la base de datos perteneciente al usuario, o modificar su propietario, antes de poder retirar al usuario.

Descripción

DROP USER retira de la base de datos el usuario dado. No retira tablas, vistas u otros objetos que pertenezcan al usuario. Si el usuario es dueño de una base de datos, se producirá un error.

Use *CREAR USUARIO* para adicionar nuevos usuarios, y *MODIFICAR USUARIO* para modificar las propiedades de un usuario. PostgreSQL viene con un guión *dropuser* que tiene la misma función de este comando (de hecho, invoca este comando) pero que puede ser ejecutado desde la shell.

Uso

Para eliminar la cuenta de un usuario:

```
DROP USER juan;
```

Compatibilidad

SQL92

No existe comando **DROP USER** en SQL92.

DROP VIEW

Nombre

`DROP VIEW` — Retira una vista definida en una base de datos

Synopsis

`DROP VIEW` *`nombre`*

Entradas

`nombre`

El nombre de la vista definida.

Salidas

`DROP`

El mensaje que se obtiene si el comando ha sido ejecutado con éxito.

`ERROR: RewriteGetRuleEventRel: rule "_RET`*`nombre`*`" not found`

Este mensaje ocurre si la vista dada no existe en la base de datos.

Descripción

DROP VIEW retira una vista definida en una base de datos. Para poder ejecutar este comando, deberá ser el dueño de la vista.

Notas

La cláusula **DROP TABLE** de Postgres también elimina vistas.

Consulte **CREATE VIEW** para una explicación de como se crean vistas.

Uso

Este comando retirará la vista llamada `variedades`:

```
DROP VIEW variedades;
```

Compatibilidad

SQL92

SQL92 especifica algunas funcionalidades adicionales para **DROP VIEW**:

```
DROP VIEW vista { RESTRICT | CASCADE }
```

Entradas

RESTRICT

Asegura que sean destruidas únicamente vistas sin otras listas dependientes y sin restricciones de integridad.

CASCADE

Cualquier vista que se refiera a esta será también eliminada, al igual que cualquier restricción de integridad.

Notas

Actualmente, para retirar una vista referida en una base de datos Postgres esta debe ser eliminada explícitamente.

END

Nombre

`END` — Lleva a cabo la transacción actual

Synopsis

```
END [ WORK | TRANSACTION ]
```

Entradas

WORK
TRANSACTION

Palabras clave opcionales. No tienen ningún efecto.

Salidas

`COMMIT`

Es el mensaje que se devuelve si la transacción se ha llevado a cabo correctamente.

`NOTICE: COMMIT: no transaction in progress`

Se da cuando no hay ninguna transacción en curso.

Descripción

END es un sinónimo de PostgreSQL para *COMMIT*.

Notas

Las palabras clave WORK y TRANSACTION son "ruidosas" y pueden ser omitidas.

Use *ROLLBACK* para abortar una transacción.

Utilización

Para hacer que todos los cambios sean permanentes:

```
END WORK;
```

Compatibilidad

SQL92

END es una extensión de PostgreSQL que proporciona una funcionalidad equivalente a *COMMIT*.

EXPLAIN

Nombre

`EXPLAIN` — Muestra el plan de ejecución de la sentencia

Synopsis

```
EXPLAIN [ VERBOSE ] consulta
```

Entradas

VERBOSE

Bandera para mostrar el plan detallado de la consulta.

`consulta`

Cualquier *`consulta`*.

Salidas

`NOTICE: QUERY PLAN: plan`

Plan de consulta explícito del backend Postgres.

`EXPLAIN`

Bandera enviada luego de mostrarse el plan.

Descripción

Este comando muestra el plan de ejecución que el planificador Postgres genera para la consulta dada. El plan de ejecución muestra la manera en que serán escaneadas las tablas referenciadas — ya sea escaneo secuencial plano, escaneo por índice, etc. — y si se referencian varias tablas, los algoritmos de unión que serán utilizados para agrupar las tuplas requeridas para cada tabla de entrada.

La parte más critica de la presentación es el costo estimado de ejecución de la consulta, que es la suposición del planificador sobre el tiempo que tomará correr la consulta

(medido en unidades de captura de páginas de disco). En realidad se muestran dos números: el tiempo inicial que toma devolverse la primer tupla, y el tiempo total para devolver todas las tuplas. Para la mayoría de las consultas lo que importa es el tiempo total, pero en algunos casos como una sub-consulta EXISTS el planificador escogerá el menor tiempo inicial en vez del menor tiempo total (ya que en todo caso el ejecutor se detendrá después de obtener la primer tupla). También, si Ud. limita el número de tuplas a devolver con una cláusula LIMIT, el planificador realiza una interpolación apropiada entre los dos costos finales para estimar cuál de los planes es realmente el menos costoso.

La opción VERBOSE emite la representación interna completa del árbol del plan, en vez de un resumen (y lo envía al archivo log del postmaster también). Usualmente esta opción es únicamente útil para la corrección de errores (debug) de Postgres.

Notas

Existe escasa documentación en Postgres con respecto a la utilización por parte del optimizador de la información de costos. Información general sobre la estimación de costos para la optimización de las consultas puede encontrarse en libros de textos de bases de datos. Refiérase a los capítulos sobre índices y el optimizador genético de consultas de la *Guía del Programador* para mayor información.

Uso

Para mostrar un plan de consulta para una consulta simple sobre una tabla con una única columna de tipo int4 y 128 filas:

```
EXPLAIN SELECT * FROM foo;
    NOTICE:  QUERY PLAN:

Seq Scan on foo  (cost=0.00..2.28 rows=128 width=4)

EXPLAIN
```

Para la misma tabla con un índice para lograr una condición *equijoin* en la consulta, **EXPLAIN** mostrará un plan distinto:

```
EXPLAIN SELECT * FROM foo WHERE i = 4;
    NOTICE:  QUERY PLAN:

Index Scan using fi on foo  (cost=0.00..0.42 rows=1 width=4)

EXPLAIN
```

Y para terminar, para la misma tabla con un índice para lograr una condición *equijoin* en la consulta, **EXPLAIN** mostrará lo siguiente para una consulta que utilice una función de agregación:

```
EXPLAIN SELECT sum(i) FROM foo WHERE i = 4;
    NOTICE:  QUERY PLAN:

Aggregate  (cost=0.42..0.42 rows=1 width=4)
  ->  Index Scan using fi on foo  (cost=0.00..0.42 rows=1 width=4)
```

Nótese que los números específicos mostrados, y aún la estrategia de consulta seleccionada, pueden variar entre dos versiones de Postgres debido al mejoramiento del planificador.

Compatibilidad

SQL92

No existe una sentencia **EXPLAIN** definida en SQL92.

FETCH

Nombre

`FETCH` — Selecciona filas usando un cursor

Synopsis

```
FETCH [ selector ] [ count ] { IN | FROM } cursor
FETCH [ RELATIVE ] [ { [ # | ALL | NEXT | PRIOR ] } ] FROM ] cur-
sor
```

Entradas

selector

selector define la dirección de FETCH. Puede ser una de las siguientes:

FORWARD

selecciona la(s) siguiente(s) filas. Es el valor por defecto si se omite *selector*.

BACKWARD

selecciona la(s) fila(s) anterior(es).

RELATIVE

Palabra sin significado (Noise word), para compatibilidad con SQL92.

count

count determina cuántas filas hay que seleccionar. Puede ser uno de los siguientes:

#

Un entero con signo que especifica cuántas filas hay que seleccionar. Dese cuenta de que un entero negativo es equivalente a cambiar el sentido de FORWARD y BACKWARD.

ALL

Devuelve todas las filas restantes.

NEXT

Equivalente a especificar un "count" de **1**.

PRIOR

Equivalente a especificar un "count" de **-1**.

cursor

El nombre de un cursor abierto.

Salidas

FETCH retorna el resultado de la consulta definida por el cursor especificado. Si la consulta falla serán mostrados los siguientes mensajes:

`NOTICE: PerformPortalFetch: portal "`*cursor*`" not found`

Si el *cursor* no está previamente declarado. El cursor debe ser declarado dentro de un bloque de operación (transaction block).

`NOTICE: FETCH/ABSOLUTE not supported, using RELATIVE`

Postgres no soporta el posicionamiento absoluto de los cursores.

`ERROR: FETCH/RELATIVE at current position is not supported`

SQL92 permite devolver de forma repetida el cursor en su "posición actual" usando la sintaxis

`FETCH RELATIVE 0 FROM `*cursor*

Postgres actualmente no soporta este concepto, de hecho, el valor cero está reservado para indicar que todas las filas deben ser devueltas y es equivalente a especificar la palabra clave ALL. Si se ha usado la palabra clave RELATIVE, Postgres asume que el usuario desea un comportamiento como en SQL92 y devuelve este mensaje de error.

Description

FETCH permite a un usuario devolver filas usando un cursor. El número de filas devueltas está especificado mediante #. Si el número de filas restantes en el cursor es menor a than #, sólo serán seleccionadas las disponibles. Sustituyendo la palabra clave ALL en lugar de un número provocará que sean devueltas todas las filas restantes en el cursor. Las instancias pueden ser seleccionadas en ambas direcciones hacia adelante y hacia atrás (FORWARD y BACKWARD). La dirección por defecto es FORWARD.

> **Sugerencia:** Se permite especificar números negativos en el contador. Un número negativo es equivalente a modificar el sentido de las palabras clave FORWARD y BACKWARD. Por ejemplo, **FORWARD -1** es igual a**BACKWARD 1**.

Notas

Dese cuenta de que las palabras clave FORWARD y BACKWARD son extensiones Postgres. La sintaxis SQL92 también es soportada, especificada en la segunda forma del comando. Véanse más abajo detalles y temas de compatibilidad.

Una vez todas las filas se han seleccionado, todos los demás accesos de fetch no devuelven filas.

Postgres no soporta la característica de actualizar los datos en un cursor, ya que volver a mapear las actualizaciones del cursor en las tablas base no es posible por regla general, como sucede también en las actualizaciones de las vistas (VIEW). Por consiguiente, los usuarios deben explicitar comandos UPDATE para sustituir los datos.

Los cursores sólo sólo se deberían usar dentro de transacciones, ya que los datos que almacenan abarcan múltiples consultas de usuario.

Usar *MOVE* para modificar la posición del cursor. *DECLARE* definirá un cursor. Refiérase a *BEGIN*, *COMMIT*, y a *ROLLBACK* para mayor información acerca de las transacciones.

Uso

Los siguientes ejemplos recorren una tabla unsando un cursor. The following examples traverses a table using a cursor.

```
-montar y usar un cursor:
-
BEGIN WORK;
  DECLARE liahona CURSOR
     FOR SELECT * FROM films;

-seleccionar las primeras cinco filas en el cursor liahona:
-
  FETCH FORWARD 5 IN liahona;

      code |title                 |did| date_prod|kind      |len
  ---+--------------+--+-------+-------+----
  BL101|The Third Man          |101|1949-12-23|Drama     | 01:44
  BL102|The African Queen      |101|1951-08-11|Romantic  | 01:43
  JL201|Une Femme est une Femme|102|1961-03-12|Romantic  | 01:25
  P_301|Vertigo                |103|1958-11-14|Action    | 02:08
  P_302|Becket                 |103|1964-02-03|Drama     | 02:28


-Seleccionar la fila anterior:
-
  FETCH BACKWARD 1 IN liahona;

      code |title                 |did| date_prod|kind      |len
  ---+--------------+--+-------+-------+----
  P_301|Vertigo                |103|1958-11-14|Action    | 02:08
```

```
- cerrar el cursor y commit work:
-
  CLOSE liahona;
COMMIT WORK;
```

Compatibilidad

SQL92

> **Nota:** El uso no embebido de los cursores es una extensión Postgres. La sintaxis y el uso de los cursores está siendo comparada en contraposición a la forma embebida de los cursores definida en SQL92.

SQL92 permite el posicionamiento absoluto del cursor para FETCH y también la localización de los resultados en variables explícitas.

```
FETCH ABSOLUTE #
    FROM cursor
    INTO :variable [, ...]
```

ABSOLUTE

El cursor debe ser posicionado al número de fila absoluto especificado. Todos los números de filas en Postgres son números relativos, por lo tanto no se soporta esta

característica.

:*variable*

Variable(s) objetivo del host.

GRANT

Nombre

GRANT — otorga privilegios de acceso a un usuario, un grupo o a todos los usuarios

Synopsis

```
GRANT privilege [, ...] ON object [, ...]
    TO { PUBLIC | GROUP group | username }
```

Entradas

privilege

Los posibles privilegios son:

SELECT

Acceso a todas las columnas de una tabla/vista específica.

INSERT

Inserta datos en todas las columnas de una tabla específica.

UPDATE

Actualiza todas las columnas de una tabal específica.

DELETE

Elimina filas de una tabla específica.

RULE

Define las reglas de la tabla(vista (cer sentencia CREATE RULE).

ALL

Otorga todos los privilegios-

object

El nombre de un objeto al que se quiere conceder el acceso. Los posibles objetos son:

- tabla
- vista
- secuencia
- indice

PUBLIC

Una abreviación para representar a todos los usuarios.

GROUP *group*

Un *grupo* al que se otorgan privilegios. En la actual versión, el grupo debe haber sido creado explícitamente como se describe más adelante.

username

El nombre de un usuario al que se quiere conceder privilegios. PUBLIC es una abreviatura para representar a todos los usuarios.

Salidas

CHANGE

Mensaje devuelto se la acción se ha realizado satisfactoriamente.

ERROR: ChangeAcl: class "*object*" not found

Mensaje devuelto si el objeto especificado no está disponible o si es imposible dar los provilegios a grupo o usuarios especificado.

Descripción

GRANT permite al creador de un objeto el dar permisos específicos a todos los usuarios (PUBLIC) o a un cierto usuario o grupo. Usuarios distintos al creador pueden no tener permisos de acceso a menos que el creador se los conceda, una vez que el objeto ha sido creado.

Una vez que un usuario tiene privilegios sobre un objeto, tiene posibilidad de ejecutar ese privilegio. No hay necesidad de conceder privilegios al creador de un objeto; el creador obtiene automáticamente TODOS los privilegios, y puede también eliminar el objeto.

Notas

Actualmente, para conceder prvilegios en Postgres a solo algunas columnas, he de crear una vista que contenga las columnas deseadas, y conceder provilegios sobre esa vista.

Use **psql \z** para obtener más información sobre los permisos de los objetos existentes:

```
Database    = lusitania
 +------------+-----------------------------+
 |  Relacion        |        Conceder/Eliminar Permisos           |
 +------------+-----------------------------+
 | mytable          | {"=rw","miriam=arwR","group todos=rw"}      |
 +------------+-----------------------------+
 Leyenda:
       uname=arwR - se conceden privilegios a un usuario
 group gname=arwR - se conceen privilegios al un GRUPO
            =arwR - se conceden privilegios a PUBLIC

                r - SELECT
                w - UPDATE/DELETE
                a - INSERT
                R - RULE
             arwR - ALL
```

> **Sugerencia:** Actualmente, para crear un GRUPO ha de insertar los datos manualmente en la tabla pg_group como sigue:
>
> ```
> INSERT INTO pg_group VALUES (’todos’);
> CREATE USER miriam IN GROUP todos;
> ```

Véase la sentencia REVOKE para ver como eliminar los privilegios de acceso.

Uso

Concede privilegios de inserción a todos los usuarios de la tabla ’films’:

```
GRANT INSERT ON films TO PUBLIC;
```

Concede todos los privilegiso al usuario ’manuel’ sobre la vista ’kinds’:

```
GRANT ALL ON kinds TO manuel;
```

Compatibilidad

SQL92

La sintaxis de SQL92 para GRANT permite establecer derechos sobre columnas individuales, y permite establecer el privilegio de conceder el mismo privilegio a otros:

```
GRANT privilege [, ...]
    ON object [ ( column [, ...] ) ] [, ...]
    TO { PUBLIC | username [, ...] } [ WITH GRANT OPTION ]
```

Los campos son compatibles con los de la implementación de Postgres, con las siguientes incorporaciones:

privilege

SQL92 permite privilegios adicionales a los mencionados:

SELECT

REFERENCES

Permitido para hacer referencia a alguna o todas las columnas de una tabla/vista específica en limitaciones de integridad.

USAGE

Permitido para usar un dominio, un conjunto de caracteres, cotejo o traducción. Si un objeto especifica algo que no sea una tabla/vista, *privilegio* ha de especificar solo USAGE.

object

[TABLE] *table*

SQL92 permite adicionalmente la palabra clave no funcional TABLE.

CHARACTER SET

Se permite usar el juego de caracteres especificado.

COLLATION

Se permite usar la secuencia de cotejo especificada.

TRANSLATION

Se permite usar la conversión de juego de caracters especificada.

DOMAIN

Se permite usar el dominio especificado.

WITH GRANT OPTION

Se permite conceder el mismo privilegio a otros.

INSERT

Nombre

`INSERT` — Inserta filas nuevas en una tabla

Synopsis

```
INSERT INTO table [ ( column [, ...] ) ]
    { VALUES ( expression [, ...] ) | SELECT query }
```

Entradas

`table`

El nombre de una tabla existente.

column

El nombre de una columna en *table*.

expression

Una expresión o un valor válidos a asignar en *column*.

query

Una consulta válida. Vea la instrución SELECT para una mejor descripción de argumentos válidos.

Salidas

INSERT *oid* 1

Mensaje devuelto si solo se ha insertado una fila. *oid* es el número OID de la fila insertada.

INSERT 0 #

Mensaje devuelto si se ha insertado más de una fila. # es el número de filas insertadas.

Descripción

INSERT permite la inserción de nuevas filas en una clase o una tabla. Se puede insertar una fila a la vez o varias como el resultado de una consulta. Las columnas en el resultado pueden ser listadas en cualquier orden.

Cada columna que no esté presente en la lista de origen será insertada usando el valor por defecto, que puede ser tanto un valor por defecto declarado DEFAULT o bien NULL. Postgres rechazará la nueva columna si se inserta un NULL en una columna declarada como NOT NULL.

Si la expresión para cada columna no es del tipo de datos correcto, se intentará una coerción de tipos automaticamente.

Debe tener privilegios de inserción en la tabla para añadir en ella, asi como privilegios de selección en cualquier tabla especificada en una clausula WHERE.

Uso

Inserta una fila en la tabla `films`:

```
INSERT INTO films VALUES
    (’UA502’,’Bananas’,105,’1971-07-13’,’Comedy’,INTERVAL ’82 minute’);
```

En este segundo ejemplo la columna `date_prod` se omite y entonces tendrá el valor por defecto de NULL:

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES (’T_601’, ’Yojimbo’, 106, DATE ’1961-06-16’, ’Drama’);
```

Inserta una fila simple en la tabla distributors; note que solo se especifica la columna `name` , de forma que la columna omitida `did` sera asignada con su valor por defecto.

```
INSERT INTO distributors (name) VALUES (’British Lion’);
```

Inserta varias filas en la tabla films desde la tabla `tmp`:

```
INSERT INTO films SELECT * FROM tmp;
```

Inserción en arrays (vea *The PostgreSQL User's Guide* para mayor información sobre los arrays):

```
- Crea un tablero de juego vacio de 3x3 para cruz y raya
- (todos estos querys generan el mismo efecto)
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1,'{{"","",""},{},{"",""}}');
INSERT INTO tictactoe (game, board[3][3])
    VALUES (2,'{}');
INSERT INTO tictactoe (game, board)
    VALUES (3,'{{„},{„},{„}}');
```

Compatibilidad

SQL92

INSERT es totalmente compatible con SQL92. Las posibles limitaciones en las caracteristicas de la clausula *`query`* estan documentadas en *SELECT*.

LISTEN

Nombre

`LISTEN` — Recibir aviso de la notificación de una condición

Synopsis

```
LISTEN nombre
```

Entradas

`nombre`

Nombre de la condición de notificación.

Salidas

`LISTEN`

Mensaje devuelto cuando se completa exitosamente el registro.

`NOTICE Async_Listen: We are already listening on nombre`

Si este backend ya fue registrado para ser avisado cuando se notifica esa condición.

Descripción

LISTEN registra al backend Postgres para recibir aviso de la notificación de una condición *nombre*.

Cada vez que el comando **NOTIFY *nombre*** es invocado, ya sea por este backend u otro conectado a la misma base de datos, todos los backends que están registrados para ser avisados de la notificación de esa condición, reciben el aviso, y en su momento cada uno de ellos notificará a su aplicación frontend. Véase el tratamiento de **NOTIFY** para mayor información.

Un backend puede anular su registro de recepción de aviso de una condición de notificación dada a través del comando **UNLISTEN**. Asimismo, todos los registros de recepción de avisos se anulan automáticamente cuando finaliza el proceso backend.

El método mediante el cual la aplicación frontend detecta los eventos de notificación depende de la interfaz de programación de aplicaciones Postgres utilizada. Con la librería básica libpq, la aplicación envía **LISTEN** como un comando SQL ordinario, y entonces llama periódicamente a la rutina `PQnotifies` para averiguar si se ha recibido algún evento de notificación. Otras interfaces como libpgtcl proporcionan métodos de alto nivel para el manejo de eventos de notificación; de hecho, con libpgtcl el programador de aplicaciones no debe enviar **LISTEN** o **UNLISTEN** directamente. Véase la documentación de la librería utilizada para mayores detalles.

NOTIFY contiene un tratamiento más extenso de la utilización de **LISTEN** y **NOTIFY**.

Notas

nombre puede ser cualquier cadena válida como nombre; no es necesario que sea igual al nombre de una tabla existente. Si *nombre* se encierra entre comillas, ni siquiera es necesario que sea un nombre válido, sino cualquier cadena de hasta 31 caracteres de largo.

En algunas versiones previas de Postgres, *nombre* debía ser encerrado entre comillas cuando no se correspondía con el nombre de una tabla existente, aunque fuera sintácticamente correcto como nombre. Actualmente no es requerido.

Uso

Configura y ejecuta una secuencia recepción de aviso/notificación desde psql:

```
LISTEN virtual;
NOTIFY virtual;

ASYNC NOTIFY of 'virtual' from backend pid '11239' received
```

Compatibilidad

SQL92

El comando **LISTEN** no existe en SQL92.

LOAD

Nombre

`LOAD` — Carga dinamicamente un fichero objeto

Synopsis

`LOAD '`*`nombrefichero`*`'`

Parametros de Entrada

nombrefichero

Nombre del fichero para cargar dinamicamente.

Outputs

LOAD

Mensaje devuelto en caso de suceso en la operacion.

ERROR: LOAD: could not open file *'nombrefichero'*

Mensaje devuelto si el fichero especificado no es encontrado.El fichero debe ser visible *alPostgres backend*, y debe ser enviado con su apropiado camino completo (path), para no obtener este tipo de error.

Descripcion

Carga un fichero objeto (o ".o") en el espacio de direccionamiento Postgres . Una vez que el fichero es cargado en memoria, todas las funciones de ese fichero pueden ser llamadas. Esta funcion es usada para soporte de tipos y funciones definidas por el usuario.

Si un fichero no es cargado usando **LOAD**, el fichero sera cargado automaticamente la primera vez que una funcion sea llamada por el Postgres. **LOAD** Puede ser usado para

recargar un fichero objeto si este ha sido editado y recompilado. Por el momento, unicamente son soportados ficheros objeto que son creados con el lenguaje C.

Notas

Funciones que se encuentran en ficheros objeto no deberian llamar a otras funciones en otros ficheros objeto que fueron cargados por medio del comando **LOAD**. Por ejemplo, todas la funciones en el fichero `A` pueden llamar a otras funciones que se encuentran en las librerias standard o math, o en las del propio Postrgres. Estas no deverian llamar funciones definidas en otro fichero cargado `B`. Esto es asi porque si `B` es recargado, el cargador del Postgres no esta preparado para realocar las llamadas desde las funciones en `A` en el nuevo espacio de direccionamiento de `B`. Si `B` no es recargado, entonces no habra problemas.

Ficheros objeto deben ser compilados para contener codigo sin dependencia de posicion. Por ejemplo, en estaciones DEC, debe usar /bin/cc con la opcion `-G 0` cuando compila ficheros objeto para ser cargados.

Si esta pensando en portar Postgres a una nueva plataforma, **LOAD** debe trabajar de forma tal que soporte ADTs.

Uso

Carga el fichero `/usr/postgres/demo/circle.o`:

```
LOAD '/usr/postgres/demo/circle.o'
```

Compatibilidad

SQL92

No existe el comando **LOAD** en SQL92.

LOCK

Nombre

`LOCK` — Explícitamente bloquea una tabla dentro de una transacción

Synopsis

```
LOCK [ TABLE ] name
LOCK [ TABLE ] name IN [ ROW | ACCESS ] { SHARE | EXCLUSIVE } MODE
LOCK [ TABLE ] name IN SHARE ROW EXCLUSIVE MODE
```

Entradas

name

El nombre de una tabla existente para bloquear.

ACCESS SHARE MODE

Nota: A este modo de bloqueo se accede automáticamente sobre tablas que estan siendo consultadas. Postgres libera automáticamente los bloqueos accedidos ACCESS SHARE despues de que se haya hecho la sentencia.

Este es el modo de bloqueo menos restrictivo el cual entra en conflicto sólo con el modo ACCESS EXCLUSIVE . Se pretende proteger una tabla que está siendo consultada de sentencias concurrentes **ALTER TABLE**, **DROP TABLE** y **VACUUM** sobre la misma tabla.

ROW SHARE MODE

Nota: Se accede automáticamente por cualquier declaración **SELECT FOR UPDATE.**

Conflictos con los modos de bloqueo EXCLUSIVE y ACCESS EXCLUSIVE.

ROW EXCLUSIVE MODE

Nota: Se accede automáticamente por cualquier sentencia **UPDATE**, **DELETE**, **INSERT.**

Conflictos con los modos SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE ACCESS EXCLUSIVE. Generalmente significa que una transacción actualiza o inserta algunas tuplas en una tabla.

SHARE MODE

Nota: Se accede automáticamente por cualquier sentencia **CREATE INDEX**

Conflictos con los modos ROW EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE y ACCESS EXCLUSIVE . Este modo protege una tabla contra actualizaciones concurrentes.

SHARE ROW EXCLUSIVE MODE

Conflictos con los modos ROW EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE y ACCESS EXCLUSIVE. Este modo es más restrictivo que el modo SHARE debido a que sólo puede soportar este bloqueo una transacción por vez .

EXCLUSIVE MODE

Conflictos con los modos ROW SHARE, ROW EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE y ACCESS EXCLUSIVE modes. Este modo es aún más restrictivo que ése de SHARE ROW EXCLUSIVE; bloquea todas las consultas concurrentes SELECT FOR UPDATE .

ACCESS EXCLUSIVE MODE

> **Nota:** Se accede automáticamente por las sentencias **ALTER TABLE**, **DROP TABLE**, **VACUUM** .

Este es el modo de bloqueo más restrictivo y es incompatible con todos los demás modos de bloqueo y protege una tabla bloqueada de cualquier otra operación concurrente.

> **Nota:** Este modo de bloqueo se accede también por un **LOCK TABLE** sin cualificar. (i.e. el comando sin una opción de bloqueo explícita).

Salidas

`LOCK TABLE`

El bloqueo se activó con éxito.

`ERROR` *`name`*`: La tabla no existe.`

Mensaje devuelto si el *`nombre`* no existe.

Description

Postgres siempre usa el modo de bloqueo menos restrictivo cuando le es posible. **LOCK TABLE** toma medidas para cuando se pueda necesitar un modo de bloqueo mas restrictivo.

Por ejemplo, una aplicación ejecuta una transacción en el nivel de aislamiento READ COMMITTED y necesita asegurar la existencia de datos en una tabla para la duracion de la transacción. Para ello tú podrías usar el modo de bloqueo SHARE sobre la tabla antes de la consulta. Esto protegerá los datos de cambios concurrentes y proporcionará cualquier otra operación de escritura sobre la tabla con datos en su verdadero estado actual, porque el modo de bloqueo SHARE es incompatible con cualquier ROW EXCLUSIVE accedido por los que esriben, y **LOCK TABLE "tabla" *en sentencia* IN SHARE MODE** esperará hasta que se produzca o se "baje" cualquier operación de escritura concurrente.

> **Nota:** Para leer datos en su verdadero estado actual cuando ejecutas una transacción en el nivel de aislamiento SERIALIZABLE tienes que ejecutar una declaración LOCK TABLE antes de la ejecución de cualquier sentencia DML, cuando la transacción define qué cambios concurrentes serán visibles por ellos mismos.

Además de los requerimientos precedentes, si una transacción va a cambiar datos en una tabla entonces se debería acceder al modo SHARE ROW EXCLUSIVE para evitar condiciones de punto muerto cuando dos transacciones coincidentes intentan bloquear la tabla en modo SHARE y entonces intentan cambiar datos en esta tabla, ambas (implicitamente) accediendo al modo de bloqueo ROW EXCLUSIVE que es incompatible con el bloqueo SHARE .

Para continuar con los puntos muertos (cuando dos transacciones se esperan la una a la otra) tema tratado arriba, deberías seguir dos reglas generales para evitar condiciones de punto muerto :

- Las transacciones tienen que acceder a bloqueos de los mismos objetos en el mismo orden.

  Por ejemplo, si una aplicación actualiza la fila R1 y después actualiza la fila R2 (en la misma transacción) entonces la segunda aplicación no debería actualizar la fila R2 si ello va a actualizar la fila R1 más tarde (en una transacción simple). En cambio, debería actualizar la fila R1 y R2 en el mismo orden como en la primera aplicación.

- Las transacciones deberían procurarse dos modos de bloqueo conflictivos sólo si uno de ellos es auto-conflictivo (i.e. podría ser soportado por sólo una transacción cada vez). Si estan involucrados modos de bloqueo múltiples, entonces las transacciones deberían siempre acceder primero al modo más restrictivo.

  Un ejemplo para esta regla se dió antes cuando se discutió el uso del modo SHARE ROW EXCLUSIVE mejor que el modo SHARE.

> **Nota:** Postgres no detecta puntos muertos "bajará" una transacción a la espera para resolver el punto muerto.

Notas

LOCK es una extension del lenguaje Postgres.

Excepto para los modos de bloqueo ACCESS SHARE/EXCLUSIVE , todos los demás modos de bloqueo de Postgres y las sentencias **LOCK TABLE** son compatibles con aquellos presentes en Oracle.

LOCK funciona sólo dentro de transacciones.

Uso

Illustrate a SHARE lock on a primary key table when going to perform inserts into a foreign key table:

```
BEGIN WORK;
LOCK TABLE películas IN SHARE MODE;
SELECT id FROM películas
    WHERE name = 'Star Wars: Episodio I - La amenaza fantasma';
- Haz ROLLBACK si el registro no fue devuelto
INSERT INTO comentarios_usuario_películas VALUES
    (_id_, 'GUAY! Llevaba tanto tiempo esperándola!');
COMMIT WORK;
```

Toma un bloqueo SHARE ROW EXCLUSIVE clave de tabla primaria cuando vayas a hacer una operación de borrado:

```
BEGIN WORK;
LOCK TABLE películas IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM comentarios_usuario_películas WHERE id IN
    (SELECT id FROM películas WHERE clasificación < 5);
DELETE FROM películas WHERE clasificación < 5;
COMMIT WORK;
```

Compatibilidad

SQL92

No hay **LOCK TABLE** en SQL92, que usa en cambio **SET TRANSACTION** para especificar niveles de concurrencia en transacciones. Nosotros también la tenemos; ver

SET para más detalles.

MOVE

Nombre

`MOVE` — Mueve la posición del cursor

Synopsis

```
MOVE [ selector ] [ count ]
    { IN | FROM } cursor
    FETCH [ RELATIVE ] [ { [ # | ALL | NEXT | PRIOR ] } ] FROM ] cur-
sor
```

Descripción

MOVE permite al usuario mover la posición del cursor un número específico de filas. **MOVE** funciona como el comando **FETCH**, pero sólo posiciona el cursor y no devuelve filas.

Ir a *FETCH* para detalles de sintaxis y uso.

Notes

MOVE es una extensión del lenguage Postgres.

Ir a *FETCH* para una descripción de los argumentos válidos. Ir a *DECLARE* par definir un cursor. Ir a *BEGIN*, *COMMIT*, y *ROLLBACK* para más información acerca de transacciones.

Usage

Configurar y usar un cursor:

```
BEGIN WORK;
DECLARE liahona CURSOR  FOR SELECT * FROM films;
-Saltarse las 5 primeras filas:
MOVE FORWARD 5 IN liahona;
MOVE
-Fetch la 6ª fila en el cursor liahona:
FETCH 1 IN liahona;
FETCH

  code |title |did| date_prod|kind      |len
  ---+----+--+-------+-------+----
  P_303|48 Hrs|103|1982-10-22|Action    | 01:37
  (1 row)
- cierra el cursor liahona and commit work:
CLOSE liahona;
COMMIT WORK;
```

Compatibility

SQL92

No hay sentencia SQL92 **MOVE** . En cambio, SQL92 permite one to **FETCH** filas de

una posición absoluta del cursor, moviendo implícitamente el cursor a una posición correcta.

NOTIFY

Nombre

`NOTIFY` — Señala todos los "fronends" y "backends" a la escucha de una condición notify.

Synopsis

`NOTIFY` *`name`*

Entradas

`notifyname`

Notifica la condición a ser señalada.

Salidas

`NOTIFY`

Acuse de recibo de que el comando notify ha sido ejecutado.

`Eventos Notify`

Los eventos son repartidos a los "frontends" que están a la escucha; el cómo y si cada aplicacion "frontend" reacciona depende de su programación.

Descripción

El comando **NOTIFY** envía un evento notify a cada aplicación frontend que previamente ha ejecutado **LISTEN *`notifyname`*** para la condición notify específica en la base de datos en curso.

La información pasada al "frontend" para un evento notify incluye el nombre de la condición notify y el PID de la notificación del proceso "backend". Es asunto del diseñador de la base de datos el definir los nombres de las condiciones que serán usadas en una base de datos dada y que significa cada una.

Comunmente, el nombre de una condición notify es el mismo que el de alguna tabla en la base de datos, y el evento notify esencialmente significa "He cambiado ésta tabla, echale un vistazo para ver los cambios". Pero dicha asociación no es obligada por lo comandos **NOTIFY** y **LISTEN**. Por ejemplo, un diseñador de bases de datos podría usar varios nombres de condición diferentes para señalar diferentes tipos de cambios en una misma tabla.

NOTIFY provee un modo simple de señalar o un mecanismo de comunicación entre procesos (IPC interprocess communication) para el conjunto de procesos que acceden a la misma base de datos Postgres. Se pueden construir mecanismos de más alto nivel

usando tablas en la base de datos para pasar datos adicionales (más allá de un mero nombre de condición) desde el notificador al o a los que estén a la escucha.

Cuando se usa **NOTIFY** para señalar la ocurrencia de cambios en una tabla en particular, una técnica útil de programación es poner **NOTIFY** en una norma que es disparada por actualizaciones de la tabla. De esta manera, la notificación es automática cuando la tabla cambia, y el programador de la aplicación no puede olvidarse de ello de forma accidental.

NOTIFY interactúa con transacciones SQL de una manera importante. Primero, si se ejecuta un **NOTIFY** dentro de una transacción, lo eventos notify no son repartidos hasta y a menos que la transacción se haya hecho. Esto es adecuado, dado que si una transacción se aborta nos gustaría que todos los comandos dentro de ella no hubieran tenido efecto, incluyendo **NOTIFY**. Pero puede ser desconcertante si uno está esperando que los eventos notify se repartan inmediatamente. Segundo, si un "backend" a la escucha recibe una señal notify mientras está en una transacción, el evento notify no se repartirá al "frontend" conectado hasta justo después de que la transacción se haya completado (tanto si se ejecuta como si se aborta). De nuevo, la razón es que si un notify fuera repartido dentro de una transacción que después fue abortado, sería deseable que la notificación se deshiciera de alguna manera — pero el "backend" no puede echar marcha atrás un notify una vez que ha sido enviado al "frontend". Por tanto los eventos notify son sólo repartidos entre transacciones. El resultado de ésto es que las aplicaciones que usan **NOTIFY** para señalar en tiempo real deberían tratar de mantener cortas sus transacciones.

NOTIFY se comporta como las señales Unix en un aspecto importante:si una misma condición es señalada varias veces en una sucesión rápida, los receptores puede que sólo recibieran un evento notify para varias ejecucuiones de **NOTIFY**. Por ello es mala idea depender del número de notificaciones recibidas. En cambio, usaremos **NOTIFY** para "despertar" a las aplicaciones que necesitan prestar atención a algo, y usaremos un objeto de base de datos (tal como una secuencia) para mantener un registro de lo que ha ocurrido o cuantas veces ha ocurrido.

Es usual para un "frontend" que envía **NOTIFY** estar él mismo a la escucha del mismo nombre notify. En ese caso recibirá un evento notify , justo igual que los otros "frontends" a la escucha. Dependiendo de la lógica de la aplicación, esto podría

acarrear un trabajo inútil — por ejemplo, releyendo una tabla de una base de datos para encontrar la misma actualización que ése mismo frontend acababa de escribir. En Postgres 6.4 y posteriores , es posible evitar dicho trabajo extra notificando si el PID del proceso de notificación del "backend" (suministrado en el mensaje del evento notify) es el mismo que el PID del backend de uno mismo (valga la redundancia) (disponible en libpq). Cuando son el mismo, el evento notificación es la recuperación del propio trabajo de uno mismo, y puede ser ignorado. (A pesar de lo que se dijo en el párrafo precedente, esto es una técnica segura. Postgres mantiene las auto-notificaciones separadas de las notificaciones que llegan de otros "backends", de manera que no puedes perder una notificación de fuera por ignorar tus propias notificaciones.(Si alguien entiende ésto que me lo explique))

Notas

name puede ser una cadena válica com un nombre; no es necesaría una relación con el nombre de la tabla en sí. Si *name* se encierra entre dobles comillas, ni siquiera necesita un nombre sintácticamente válido, sino que pude ser cualquier cadena de hasta 31 caracteres de longitud.

En algunas versiones previas de Postgres, *name* tenía que encerrarse entre comillas dobles cuando no había relación con ningún nombre de tabla existente, incluso si sintácticamente era válido como nombre. Esto ya no es necesario.

En versiones Postgres anteriores a la 6.4, el PID de backend repartido en un mensaje notify era siempre el PID del backend del frontend de uno mismo. Por eso no se podía distinguir las notificaciones deuno mismo de las notificaciones de otros clientes en aquellas versiones.

Uso

Configura y ejecuta una secuencia listen(escucha)/notify(notificación) desde psql:

```
LISTEN virtual;
```

```
NOTIFY virtual;
ASYNC NOTIFY de ’virtual’ desde el pide de backend  ’11239’ recivido
```

Compatibilidad

SQL92

No hay sentencia**NOTIFY** en SQL92.

RESET

Nombre

`RESET` — Restaura los parámetros en tiempo de ejecución a sus valores por defecto para la sesión actual.

Synopsis

`RESET` *`variable`*

Entradas

`variable`

Refiérase a *SET* para mayor información sobre variables disponibles.

Salidas

`RESET VARIABLE`

Mensaje devuelto si la *`variable`* pudo ser restaurada exitosamente a su valor por defecto.

Descripción

RESET restaura variables a sus valores por defecto. Refiérase a *SET* para mayores detalles sobre valores permitidos y por defecto. **RESET** es una forma alternativa para

`SET` *`variable`* `= DEFAULT`

Notas

RESET es una extensión del lenguaje de Postgres

Utilice *SET* and *SHOW* para manipular el valor de las variables.

Uso

Establecer DateStyle (estilo de fecha) a su valor por defecto:

```
RESET DateStyle;
```

Establecer Geqo a su valor por defecto:

```
RESET GEQO;
```

Compatibilidad

SQL92

No existe **RESET** en SQL92.

REVOKE

Nombre

`REVOKE` — Revoca el privilegio de acceso a un usuario, a un grupo o a todos los usuarios.

Synopsis

```
REVOKE privilegio [, ...]
    ON objeto [, ...]
    FROM { PUBLIC | GROUP ER">gBLE> | nombre_usuario }
```

Entradas

privilegio

Los posibles privilegios son:

SELECT

Privilegio para acceder a todas las columnas de una tabla o vista específica.

INSERT

Privilegio de insertar datos en todas las columnas de una tabla específica.

UPDATE

Privilegio para actualizar todas las columnas de tabla.

DELETE

Privilegio para borrar filas de una tabla específica.

RULE

Privilegio para definir reglas en una tabla o vista. (Veáse *CREATE RULE*).

ALL

Rescinde todos los privilegios.

objeto

El nombre de un objeto sobre el que revocar el acceso. Los posibles objetos son:

- tablea
- vista
- secuencia
- índice

grupo

El nombre de un grupo al cual se revocan privilegios.

nombre_usuario

El nombre de un usuario al cual se revocan privilegios. Utilice la palabra clave PUBLIC para especificar todos los usuarios.

PUBLIC

Rescinde el/los privilegio(s) especificado(s) a todos los usuarios.

Salidas

CHANGE

Mensaje devuelto si ha tenido éxito.

ERROR

Mensaje que se devuelve si el objeto no está disponible o si es imposible revocar privilegios al grupo o a los usuarios.

Descripción

REVOKE permite al creador de una objeto revocar permisos asignados anteriormente a todos los usuarios (mediante PUBLIC) o a un usuario o a un grupo.

Notas

Consulte el comando psql \z para obtener más información sobre permisos en objetos existentes:

```
Database    = lusitania
+------------+-----------------------------+
|  Relation        |        Grant/Revoke Permissions             |
+------------+-----------------------------+
| mytable          | {"=rw","miriam=arwR","group todos=rw"}      |
+------------+-----------------------------+
Legend:
     uname=arwR - privileges granted to a user
     group gname=arwR - privileges granted to a GROUP
     =arwR - privileges granted to PUBLIC

     r - SELECT
     w - UPDATE/DELETE
     a - INSERT
     R - RULE
     arwR - ALL
```

Sugerencia: Actualmente, para crear un grupo debe insertar los datos manualmente en la tabla table pg_group de este modo:

```
INSERT INTO pg_group VALUES (’todos’);
CREATE USER miriam IN GROUP todos;
```

Utilización

Revoca el privilegio de inserción a todos los usuarios en la tabla `films`:

```
REVOKE INSERT ON films FROM PUBLIC;
```

Revoca todos los privilegios al usuario `manuel` en la vista `kinds`:

```
REVOKE ALL ON kinds FROM manuel;
```

Compatibilidad

SQL92

La sintaxis de SQL92 para el comando **REVOKE** tiene capacidades adicionales para rescindir privilegios, incluso aquellos en columnas individuales en tablas:

```
REVOKE { SELECT | DELETE | USAGE | ALL PRIVILEGES } [, ...]
    ON objeto
    FROM { PUBLIC | nombre_usuario [, ...] } { RESTRICT | CASCADE }
REVOKE { INSERT | UPDATE | REFERENCES } [, ...] [ ( columna [, ...] ) ]
    ON objeto
```

```
    FROM { PUBLIC | nombre_usuario [, ...] } { RESTRICT | CASCADE }
```

Vea *GRANT* para más detalles en campos individuales.

```
REVOKE GRANT OPTION FOR privilegio [, ...]
    ON objeto
    FROM { PUBLIC | nombre_usuario [, ...] } { RESTRICT | CASCADE }
```

Rescinde a un usuario la autoridad para garantizar el privilegio especificado a otros usuarios. Veáse *GRANT* para los detalles en campos individuales.

Los objetos posibles son:

[TABLE] tabla/vista
CHARACTER SET conjunto_caracteres
COLLATION colección
TRANSLATION traslación
DOMAIN dominio

Si user1 da un privilegio con la opción GRANT a user2 y user2 se lo da a user3, entonces user1 puede revocar este privilegio en cascada usando la palabra clave CASCADE.

Si user1 da un privilegio con GRANT a user2 y user2 se lo da a user3, entonces si user1 intenta revocar este privilegio, fallará si ha especificado la palabra clave RESTRICT.

ROLLBACK

Nombre

`ROLLBACK` — Interrumpte la transacción en curso

Synopsis

```
ROLLBACK [ WORK | TRANSACTION ]
```

Entrada.

Ninguna.

Salida.

`ABORT`

Mensaje devuelto si la operación es exitosa.

`NOTICE: ROLLBACK: no transaction in progress`

Si no hay transacciones en progreso actualmente.

Descripción

ROLLBACK deshace la transacción actual y provoca que todas las modificaciones originadas por la misma sean descartadas.

Notas

Utilice *COMMIT* para terminar una transacción de forma exitosa. *ABORT* es un sinónimo de **ROLLBACK**.

Usage

Para cancelar todos los cambios:

```
ROLLBACK WORK;
```

Compatibilidad

SQL92

SQL92 sólo especifica las dos formas siguientes: `ROLLBACK` y `ROLLBACK WORK`. De cualquier otra forma, la compatibilidad es completa.

SELECT

Nombre

`SELECT` — Recupera registros desde una tabla o vista.

Synopsis

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    expression [ AS name ] [, ...]
    [ INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table ]
    [ FROM table [ alias ] [, ...] ]
    [ WHERE condition ]
    [ GROUP BY column [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION [ ALL ] | INTERSECT | EXCEPT } select ]
    [ ORDER BY column [ ASC | DESC | USING operator ] [, ...] ]
    [ FOR UPDATE [ OF class_name [, ...] ] ]
    LIMIT { count | ALL } [ { OFFSET | , } start ]
```

Inputs

`expression`

El nombre de una columna de la tabla o una expresión.

`name`

Especifica otro nombre para una columna o una expresión que utilice la cláusula AS. Este nombre se utiliza principalmente como etiqueta para la columna de salid. El nombre no puede ser utilizado en las cláusulas WHERE, GROUP BY o HAVING. Sin embargo, puede ser referenciado en cláusulas ORDER BY.

TEMPORARY
TEMP

La tabla se crea solamente para esta sesión, y es automáticamente descartada al finalizar la misma.

new_table

Si se utiliza la cláusula INTO TABLE, el resultado de la consulta se almacenará en otra tabla con el nombre indicado. La tabla objetivo (*new_table*) será creada automáticamente y no deberá existir previamente a la utilización de este comando. Consulte el comando **SELECT INTO** para más información.

Nota: La declaración **CREATE TABLE AS** también creará una nueva tabla a partir de la consulta.

table

El nombre de una tabla existente a la que se refiere la cláusula FROM.

alias

Un nombre alternativo para la tabla precedente *table*. Se utiliza para abreviar o eliminar ambigüedades en uniones dentro de una misma tabla.

condition

Una expresión booleana que da como resultado verdadero o falso (true or false). Consulte la cláusula WHERE.

column

El nombre de una columna de la tabla.

select

Una declaración de selección (select) exceptuando la cláusula ORDER BY.

Outputs

Registros

El conjunto completo de registros (filas) que resultan de la especificación de la consulta.

`count`

La cantidad de registros (filas) devueltos por la consulta.

Descripción

SELECT devuelve registros de una o más tablas. Los candidatos a ser seleccionados son aquellos registros que cumplen la condición especificada con WHERE; si se omite WHERE, se retornan todos los registros. (Consulte *Cláusula WHERE*.)

DISTINCT elimina registros duplicados del resultado. **ALL** (predeterminado) devolverá todos los registros, que cumplan con la consulta, incluyendo los duplicados.

DISTINCT ON elimina los registros que cumplen con todas las expresiones especificadas, manteniendo solamente el primer registro de cada conjunto de duplicados. Note que no se puede predecir cuál será "el primer registro" a menos que se utilice **ORDER BY** para asegurar que el registro eseado es el que efectivamente aparece primero. Por ejemplo:

```
SELECT DISTINCT ON (location) location, time, report
FROM weatherReports
ORDER BY location, time DESC;
```

recuperea el reporte de tiempo (weather report) más reciente para cada locación (location). Pero si no se hubiera utilizado ORDER BY para forzar el orden descendente de los valores de fecha para cada locación, se hubiesen recuperado reportes de una fecha impredecible para cada locación.

La cláusula GROUP BY permite al usuario dividir una tabla conceptualmente en grupos. (Consulte *Cláusula GROUP BY*.)

La cláusula HAVING especifica una tabla con grupos derivada de la eliminación de grupos del resultado de la cláusula previamente especificada. (Consulte *Cláusula HAVING*.)

La cláusula ORDER BY permite al usuario especificar si quiere los registros ordenados de manera ascendente o descendente utilizando los operadores de modo ASC y DESC. (Consulte *Cláusula ORDER BY*.)

El operador UNION permite permite que el resultado sea una colección de registros devueltos por las consultas involucradas. (Consulte *Cláusula UNION*.)

El operador INTERSECT le da los registros comunes a ambas consultas. (Consulte *Cláusula INTERSECT*.)

El operador EXCEPT le da los registros devueltos por la primera consulta que no se encuentran en la segunda consulta. (Consulte *Cláusula EXCEPT*.)

La cláusula FOR UPDATE permite a SELECT realizar un bloqueo exclusivo de los registros seleccionados.

La cláusula LIMIT permite devolver al usuario un subconjunto de los registros producidos por la consulta. (Consulte *Cláusula LIMIT*.)

Usted debe tener permiso de realizar SELECT sobre una tabla para poder leer sus valores. (Consulte las declaraciones **GRANT**/**REVOKE**).

Cláusula WHERE

La condición opcional WHERE tiene la forma general:

```
WHERE boolean_expr
```

boolean_expr puede consistir de cualquier expresión cuyo resultado sea un valor booleano. En muchos casos, esta expresión será:

```
expr cond_op expr
```

o

```
log_op expr
```

donde *cond_op* puede ser uno de: =, <, <=, >, >= or <>, un operador condicional como ALL, ANY, IN, LIKE o operador definido localmente, y *log_op* puede ser uno de: AND, OR, NOT. La comparación devuelve TRUE (verdadero) o FALSE (falso) y todas las instancias serán descartadas si la expresión resulta falsa.

Cláusula GROUP BY

GROUP BY especifica una tabla con grupos derivada de la aplicación de esta cláusula:

```
GROUP BY column [, ...]
```

GROUP BY condensará en una sola fila todos aquellos registros que compartan los mismos valores para las columnas agrupadas. Las funciones de agregación, si las hubiera, son computadas a través de todas las filas que conforman cada grupo, produciendo un valor separado por cada uno de los grupos (mientras que sin GROUP BY, una función de agregación produce un solo valor computado a través de todas las filas seleccionadas). Cuando GROUP BY está presente, no es válido hacer referencia a columnas no agrupadas excepto dentro de funciones de agregación, ya que habría más de un posible valor de retorno para una columna no agrupada.

Cláusula HAVING

La condición opcional HAVING tiene la forma general:

```
HAVING cond_expr
```

donde *cond_expr* cumple las mismas condiciones que las especificadas para WHERE.

HAVING especifica una tabla con grupos derivada de la eliminación de grupos, del resultado de la cláusula previamente especificada, que no cumplen con *cond_expr*.

Cada columna referenciada en *cond_expr* debe referirse precisamente (sin ambigüedades) a una columna de grupo, a menos que la referencia aparezca dentro de una función de agregación.

Cláusula ORDER BY

```
ORDER BY column [ ASC | DESC ] [, ...]
```

column puede ser tanto el nombre de una columna como un número ordinal.

Los números ordinales hacen referencia a la posición (de izquierda a derecha) de la columna. Esta característica hace posible definir un orden basado en una columna que no tiene un nombre adecuado. Esto nunca es absolutamente necesario ya que siempre es posible asignar un nombre a una columna calculada utilizando la cláusula AS, por ej.:

```
SELECT title, date_prod + 1 AS newlen FROM films ORDER BY newlen;
```

A partir de la versión 6.4 de PostgreSQL, es también posible ordenar, con ORDER BY, según expresiones arbitrarias, incluyendo campos que no aparecen en el resultado de SELECT. Por lo tanto, la siguiente declaración es legal:

```
SELECT name FROM distributors ORDER BY code;
```

Opcionalmente una puede agregar la palabra clave DESC (descendente) o ASC (ascendente) luego del nombre de cada columna en la cláusula ORDER BY. Si no se especifica, se asume ASC de forma predeterminada. Alternativamente, puede indicarse un nombre de operador de orden específico. ASC es equivalente a USING '<' y DESC es equivalente a USING '>'.

Cláusula UNION

```
table_query UNION [ ALL ] table_query
    [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

donde *table_query* especifica cualquier declaración SELECT sin la cláusula ORDER BY.

El operador UNION permite que el resultado sea una colección de registros devueltos por las consultas involucradas. Los dos SELECTs que representan los dos operandos directos de la UNION deben producir el mismo número de columnas, y las columnas correspondientes deben ser de tipos de datos compatibles.

De forma predeterminada, el resultado de UNION no contiene registros duplicados a menos que se especifique la cláusula ALL.

Si se utilizan varios operadores UNION en la misma declaración SELECT se evalúan de izquierda a derecha. Note que la palabra clave ALL no es global, siendo aplicada solamente al par de tablas de resultado actual.

Cláusula INTERSECT

```
table_query INTERSECT table_query
    [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

donde *table_query* especifica cualquier expresión SELECT sin la cláusula ORDER BY.

El operador INTERSECT le da los registros comunes a ambas consultas. Los dos SELECTs que representan los operandos directos de la intersección deben producir el mismo número de columnas, y las columnas correspondientes deben ser de tipos de datos compatibles.

Si se utilizan varios operadores INTERSECT en la misma declaración SELECT se evalúan de izquierda a derecha, a menos que se utilicen paréntesis para modificar esto.

Cláusula EXCEPT

```
table_query EXCEPT table_query
     [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

donde *table_query* especifica cualquier expresión SELECT sin la cláusula ORDER BY.

El operador EXCEPT le da los registros devueltos por la primera consulta pero no por la segunda. Los dos SELECTs que representan los operandos directos de la intersección deben producir el mismo número de columnas, y las columnas correspondientes deben ser de tipos de datos compatibles.

Si se utilizan varios operadores INTERSECT en la misma declaración SELECT se evalúan de izquierda a derecha, a menos que se utilicen paréntesis para modificar esto.

Cláusula LIMIT

```
LIMIT { count | ALL } [ { OFFSET | , } start ]
OFFSET start
```

donde *`count`* especifica el máximo número de registros a devolver y *`start`* especifica el número de registros a saltear antes de empezar a devolver registros.

LIMIT le permite recuperar sólo una porción de los registros que se generan por el resto de la consulta. Si se especifica un número límite, no se devolverán más registros que esa cantidad. Si se da un valor de desplazamiento, esa cantidad de registros será salteada antes de comenzar a devolver registros.

Cuando se utiliza LIMIT es una buena idea utilizar la cláusula ORDER BY para colocar los registros del resultado en un orden único. De otra forma obtendrá un subconjunto impredecible de los registros de la consulta — tal vez esté buscando los registros del décimo al vigésimo, ¿pero del décimo al vigésimo en qué orden? Usted no conoce el orden a menos que utilice ORDER BY.

Ya en Postgres 7.0, el optimizador de consultas toma en cuenta a LIMIT cuando genera un plan de consulta, así que es muy factible que usted obtenga diferentes planes (abarcando diferentes criterios de ordenamiento de registros) dependiendo de los valores dados a LIMIT y OFFSET. Por lo tanto, utilizar diferentes valores para LIMIT/OFFSET para seleccionar diferentes subconjuntos del resultado de una consulta, *provocará resultados inconsistentes* a menos que usted se asegura un resultado predecible ordenando con ORDER BY. Esto no es un bug; es una consecuencia inherente al hecho de que SQL no establece ningún compromiso de entregar los resultados de una consulta en un orden en particular a menos que se utilice ORDER BY para especificar un criterio de orden explícitamente.

Uso

Para unir la tabla `films` con la tabla `distributors`:

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
    FROM distributors d, films f
    WHERE f.did = d.did
```

```
title                    |did|name            | date_prod|kind
-------------------+--+-----------+-------+-------
The Third Man            |101|British Lion    |1949-12-23|Drama
The African Queen        |101|British Lion    |1951-08-11|Romantic
Une Femme est une Femme  |102|Jean Luc Godard |1961-03-12|Romantic
Vertigo                  |103|Paramount       |1958-11-14|Action
Becket                   |103|Paramount       |1964-02-03|Drama
48 Hrs                   |103|Paramount       |1982-10-22|Action
War and Peace            |104|Mosfilm         |1967-02-12|Drama
West Side Story          |105|United Artists  |1961-01-03|Musical
Bananas                  |105|United Artists  |1971-07-13|Comedy
Yojimbo                  |106|Toho            |1961-06-16|Drama
There's a Girl in my Soup|107|Columbia        |1970-06-11|Comedy
Taxi Driver              |107|Columbia        |1975-05-15|Action
Absence of Malice        |107|Columbia        |1981-11-15|Action
Storia di una donna      |108|Westward        |1970-08-15|Romantic
The King and I           |109|20th Century Fox|1956-08-11|Musical
Das Boot                 |110|Bavaria Atelier |1981-11-11|Drama
Bed Knobs and Broomsticks|111|Walt Disney     |          |Musical
```

Para sumar la columna `len` (duración) de todos los filmes y agrupar los resultados según la columna `kind` (tipo):

```
SELECT kind, SUM(len) AS total FROM films GROUP BY kind;
```

```
    kind      |total
    -------+----
```

```
    Action    | 07:34
    Comedy    | 02:58
    Drama     | 14:28
    Musical   | 06:42
    Romantic  | 04:38
```

Para sumar la columna `len` de todos los filmes, agrupar los resultados según la columna `kind` y mostrar los totales de esos grupos que sean menores a 5 horas:

```
SELECT kind, SUM(len) AS total
    FROM films
    GROUP BY kind
    HAVING SUM(len) < INTERVAL ’5 hour’;

    kind      |total
    -------+----
    Comedy    | 02:58
    Romantic  | 04:38
```

Los siguientes dos ejemplos muestran maneras idénticas de ordenar los resultados individuales de acuerdo con los contenidos de la segunda columna (`name`):

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;

    did|name
    --+-----------
    109|20th Century Fox
    110|Bavaria Atelier
    101|British Lion
    107|Columbia
    102|Jean Luc Godard
```

```
    113|Luso films
    104|Mosfilm
    103|Paramount
    106|Toho
    105|United Artists
    111|Walt Disney
    112|Warner Bros.
    108|Westward
```

Este ejemplo muestra cómo obtener la union de las tablas `distributors` y `actors`, restringiendo los resultados a aquellos que comienzan con la letra W en cada tabla. No se quieren duplicados, así que la palabra clave ALL se omite.

```
    -        distributors:               actors:
    -        did|name                    id|name
    -        --+--------             -+---------
    -        108|Westward                 1|Woody Allen
    -        111|Walt Disney              2|Warren Beatty
    -        112|Warner Bros.             3|Walter Matthau
    -        ...                         ...
```

```
SELECT distributors.name
    FROM   distributors
    WHERE  distributors.name LIKE 'W%'
UNION
SELECT actors.name
    FROM   actors
    WHERE  actors.name LIKE 'W%'

name
---------
Walt Disney
Walter Matthau
Warner Bros.
Warren Beatty
```

```
Westward
Woody Allen
```

Compatibilidad

Extensiones

Postgres permite omitir la cláusula **FROM** de una consulta. Esta característica fue conservada del lenguaje original de consulta PostQuel:

```
SELECT distributors.* WHERE name = ’Westwood’;

    did|name
    --+----------
    108|Westward
```

SQL92

Cláusula SELECT

En el estándar SQL92, la palabra clave opcional "AS" es totalmente prescindible y puede ser omitida sin afectar el significado. El analizador sintáctico de Postgres requiere la presencia de esta palabra cuando se renombran columnas debido a las características de extensibilidad de tipos que pueden llevar a interpretaciones ambiguas en este contexto.

DISTINCT ON no es parte de SQL92. Tampoco los son LIMIT y OFFSET.

Cláusula UNION

La sintaxis de SQL92 para UNION admite una cláusula adicional CORRESPONDING BY:

```
table_query UNION [ALL]
    [CORRESPONDING [BY (column [,...])]]
    table_query
```

La cláusula CORRESPONDING BY no es soportada por Postgres.

SELECT INTO

Nombre

SELECT INTO — Crear una nueva tabla a partir de una tabla o vista ya existente.

Synopsis

```
SELECT [ ALL | DISTINCT [ ON ( expresión [, ...] ) ] ]
    expresión [ AS nombre ] [, ...]
    [ INTO [ TEMPORARY | TEMP ] [ TABLE ] nueva_tabla ]
    [ FROM tabla [ alias ] [, ...] ]
    [ WHERE condición ]
    [ GROUP BY columna [, ...] ]
    [ HAVING condiciónn [, ...] ]
```

```
[ { UNION [ ALL ] | INTERSECT | EXCEPT } select ]
[ ORDER BY columna [ ASC | DESC | USING operador ] [, ...] ]
[ FOR UPDATE [ OF Nombre_de_clase [, ...] ] ]
LIMIT { contador | ALL } [ { OFFSET | , } incio ]
```

Inputs

Todos los campos de entrada se describen en detalle en *SELECT*.

Outputs

Todos los campos de salida se describen en detalle en *SELECT*.

Descripción

SELECT INTO Crea una nueva tabla a partir del resultado de una query. Típicamente, esta query recupera los datos de una tabla existente, pero se permite cualquier query de SQL.

> **Nota:** *CREATE TABLE AS* es funcionalmente equivalente al comando **SELECT INTO**.

SET

Nombre

`SET` — Fija parámetros de tiempo de ejecución para la sesión.

Synopsis

```
SET variable { TO | = } { 'value' | DEFAULT }
SET TIME ZONE { 'timezone' | LOCAL | DEFAULT }
SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }
```

Inputs (Valores de entrada)

`variable`

Parámetro glogal que se quiere fijar.

`value`

Nuevo valor del parámetro. Se puede utilizar el valor `DEFAULT` para especificar que se devuelve el parámetro a su valor de defecto.

Las variables posibles y los valores permitidos son:

CLIENT_ENCODING | NAMES

Fija la codificación para clientes mult-byte. Los parámetros son:

value

Fija la codificación de cliente multi-byte a: *value*. La codificación especificada debe estar soportada por el servidor.

Esta opción solo es utilizable si el soporte MULTIBYE se autorizó durante el paso de configuración en la construcción de Postgres.

DateStyle

Fija el estilo de representación de fecha/hora. Afecta al formato de salida, y en algunos casos puede afectar a la interpretación de la entrada.

ISO

utiliza fechas y horas de estilo ISO 8601.

SQL

utiliza fechas y horas de estilo Oracle/Ingres.

Postgres

utiliza el formato tradicional de Postgres.

European

utiliza dd/mm/yyyy para la representación numérica de las fechas.

NonEuropean

utiliza mm/dd/yyyy para la representación numérica de las fechas.

German

utiliza dd.mm.yyyy para la representación numérica de las fechas.

US

igual que ’NonEuropean’

DEFAULT

recupera los valores de defecto (’US,Postgres’)

La inicialización del formato de la fecha se puede hacer:
Fijando la variable de entorno PGDATESTYLE. Si PGDATESTYLE se fija en el ambiente de un
Ejecutando postmastes utilizando la opción `-o -e` se fijan las fechas a la convención `Europea`.
Cambiando las variables en `src/backend/utils/init/globals.c`.

Las variables de `globals.c` que se pueden cambiar son:

bool EuroDates = false | true
int DateStyle = USE_ISO_DATES | USE_POSTGRES_DATES | USE_SQL_DATES | USE_GE

SERVER_ENCODING

Fija la codificación multi-byte para el servidor.

value

Fija la codificación multi-bye para el servidor.

Esta opción sólo está disponible si se habilitó el soporte MULTIBYTE durante el paso de configuración de la construcción de Postgres.

TIMEZONE

Los valores posibles para timezone dependen de su sistema operativo. Por ejemplo, en Linux /usr/lib/zoneinfo contiene la base de datos de zonas horarias.

Aquí tiene algunos valores válidos para zonas horarias:

'PST8PDT'

situa la zona horaria de California.

'Portugal'

sitúa la zona horaria de Portugal.

'Europe/Rome'

sitúa la zona horaria de Italia.

DEFAULT

fija la zona horaria a su valor local. (el valor de la variable de entorno TZ).

Si se especifica una zona horaria invalida, será fijada a GMT (en la mayoría de sistemas en cualquier caso).

La segunda sistaxis mostrada más arriba, permite fijar la zona horaria con una sintaxis similar a **SET TIME ZONE** de SQL92. La palabra clave LOCAL es sólo un formato alternativo a DEFAULT para mantener la compatibilidad con SQL92.

Si la variable de entorno PGTZ se fija en el ambiente de la aplicación de un cliente basado en libpq (en el ambiente del frontend), libpq fijará automáticamente TIMEZONE al valor de PGTZ durante el arranque de la conexión.

TRANSACTION ISOLATION LEVEL

Fija el nivel de aislamiento para la transacción actual.

READ COMMITTED

Las consultas de la transacción actual leen sólo filas aseguradas (committed) antes de empezar una consulta. READ COMMITTED es el valor de defecto.

Nota: El estandar SQL92 requiere que se fije el valor de aislamiento de defecto a SERIALIZABLE.

SERIALIZABLE

Las consultas de la transacción llen sólo fila aseguradas antes de la primera instrucción DML (**SELECT/INSERT/DELETE/UPDATE/FETCH/COPY_TO**) que se ejecute en esta transacción.

Hay también varios parámetros internos o de optimización que se pueden especificar con el comando **SET**:

RANDOM_PAGE_COST

Fija la estimación del optimizador del coste de una página de disco leída no secuencialmente. Eso se mide como un múltiplo del coste de una lectura de página secuencial.

float8

Fija el coste de un acceso aleatorio a un página al valor punto flotante especificado.

CPU_TUPLE_COST

Fija la estimación que hará el optimizador del coste de procesar cada tupla durante una consulta. Esto se mide como una fracción del coste de una lectura

secuencial de una página.

float8

Fija el coste de proceso de CPU por tupla al valor de de punto flotante especificado.

CPU_INDEX_TUPLE_COST

Fija la estimación que hará el optimizador sobre el coste de procesar cada tupla del índice durante el procesado de un barrido del índice (index scan). Se mide como una fracción del coste de una lectura secuencial de página.

float8

Fija el coste de CPU de procesado por tupla de índice al valor de punto flotante especificado.

CPU_OPERATOR_COST

Fija la estimación que hará el optimizador del coste de procesar cada operador en una cláusula WHERE. Esto se mide como una fracción del coste de un acceso secuencial a una página.

float8

Fija le coste de CPU para procesar cada operador al valor de punto flotante especificado.

EFFECTIVE_CACHE_SIZE

Fija la estimación que hará el optimizador sobre el tamaño efectivo de la caché en disco (es decir, la porción de la caché en disco del kernel que será utilizada por los ficheros de datos de Postgres). Esto se mide en páginas de disco, normalmente en

piezas de 8 Kb.

float8

Fija el tamaño estimado de la caché en el valor de punto flotante especificado.

ENABLE_SEQSCAN

Habilita o inhabilita el uso por el planificador de tipos de planes de barrido secuencial. (No es posible suprimir completamente los barridos secuenciales, pero desactivando esta variable se disuade al planificador de utilizar uno de ellos si dispone de otro método utilizable).

ON

Habilita el uso de barridos secuenciales (valor de defecto).

OFF

Inhabilita el uso de barridos secuenciales.

ENABLE_INDEXSCAN

Habilita o inhabilita el uso por el planificador de tipos de planes de barrido de índices.

ON

Habilita el uso de barridos de índices (valor de defecto).

OFF

Inhabilita el uso de barridos de índices.

ENABLE_TIDSCAN

Habilita o inhabilita el uso por el planificador de tipos tipos de planes por barrido TID.

ON

Habilita el uso de barridos TID (valor de defecto).

OFF

Inhabilita el uso de barridos TID.

ENABLE_SORT

Habilita o inhabilita el uso por el planificador pasos de ordenación explícita. (No es posible suprimir por completo las ordenaciones explititas, pero fijando en OFF esta variable disuade al planificador de usar uno cuando tiene otro método utilizable.)

ON

Habilita el uso de ordenaciones (valor de defecto).

OFF

Inhabilita el uso de ordenaciones.

ENABLE_NESTLOOP

Habilita o inhabilita el uso por el planificador de planes de join de blucle anidado. (No es posible suprimer por completo las joins de bucle anidado, pero fijar en OFF esta variable disuade al planificador de utilizar uno de ellos si dispone de otro método).

ON

Habilita el uso de joins de bucle anidado (valor de defecto).

OFF

Inshabilita el uso de joins de bucle anidado.

ENABLE_MERGEJOIN

Habilita o inhabilita el uso por el planificador de planes de tipo "enlace intercalado" (mergejoin).

ON

Habilita el uso de enlaces intercalados (valor de defecto).

OFF

Inhabilita el uso de enlaces intercalados.

ENABLE_HASHJOIN

Habilita o inhabilita el uso por el planificador de planes de tipo enlace hash (hashjoin).

ON

Habilita el uso de enlaces hash (valor de defecto).

OFF

Inhabilita el uso de enlaces hash.

GEQO

Fija el porcentaje de uso del algoritmo genérico del optimizador.

ON

Habilita el algoritmo genérico del optimizador para instrucciones con 11 tablas o más. (Este es también el valor de defecto DEFAULT).

ON=#

Toma un argumento entero para habilitar el algoritmo genérico para instrucciones con # o más tablas en la consulta.

OFF

Inhabilita el algoritmo genérico del optimizador.

Vea el capítulo sobre GEQO de la Guía del Programador para obtener más información sobre la optimización de la consulta.

Si la variable de entorno PGGEQO se fija en el ambiente de usuario de un cliente basado en libpq, libpq automáticamente fijará GEQO al valor de PGGEQO durante el arranque de la conexión.

KSQO

Key Set Query Optimizer (Optimizador de la Consulta Fijado por Clave) lleva al planificador de la consulta a convertir aquellas consultas cuyas cláusulas WHERE incluyan muchas cláusulas OR y AND (tales como "WHERE (a=1 AND b=2) OR (a=2 AND b=3) ...") en una consulta UNION. Este metodo puede ser más rápido que la implementación de defecto, pero no necesariamente produce exactamente el mismo resultado, puesto que UNION implítitamente añade una cláusula SELECT DISTINCT para eliminar las filas resultantes que sean identicas. KSQO se utiliza habitualmente cuando se trabaja con productos como MicroSoft Access, que tienden a generar las consultas de esta forma.

ON

Habilita esta optimización.

OFF

Inhabilita esta optimización (valor de defecto).

DEFAULT

Equivalente a especificar **SET KSQO='OFF'**.

El algoritmo KSQO se utilizaba por ser absolutamente esencial para consultas con muchas cláusulas OR y AND, pero en Postgres 7.0 y posteriores, el planificador estandar manipula estas consultas correctamente.

Outputs

`SET VARIABLE`

Mensaje devuelto si se fija el valor con éxito.

`WARN: Bad value for` *`variable`* `(`*`value`*`)`

Si el comando falla al fijar el valor especificado.

Descripción

SET modificará los parámetros de configuración para la variable durante una sesión.

Los valores en vigor se pueden obtener utilizando el **SHOW**, y los valores pueden devolverse a su situación de defecto utilizando **RESET**. Valores y parámetros son sensibles a mayúsculas y minúsculas. Nótese que el campo "valor" siempre se especifica como una cadena de caracteres, de modo que se encierra entre comillas simples.

SET TIME ZONE cambia la asignación de zona horaria de defecto de la sesión. Una sesión SQL siempre empieza con un valor inicial de asignación de zona horaria. La

instrucción **SET TIME ZONE** se utiliza para cambiar la asignación de zona horaria para la sesión SQL actual.

Notas

La instrucción **SET *variable*** es una extensión del lenguaje de Postgres.

Refierase a **SHOW** y **RESET** para mostrar o inicializar los valores actuales.

Uso

Fijar el estilo de la fecha a ISO:

```
SET DATESTYLE TO 'ISO';
```

Habilitar GEQO para consultas con 4 o más tablas:

```
SET GEQO ON=4;
```

Fijar GEQO a su valor de defecto:

```
SET GEQO = DEFAULT;
```

Fijar la zona horaria a Berkeley, California:

```
SET TIME ZONE 'PST8PDT';
SELECT CURRENT_TIMESTAMP AS ahora;

   ahora
   ---------------
   1998-03-31 07:41:21-08
```

Fijar la zona horaria para Italia:

```
SET TIME ZONE 'Europe/Rome';
SELECT CURRENT_TIMESTAMP AS ahora;

   ahora
   ---------------
   1998-03-31 17:41:31+02
```

Compatibilidad

SQL92

No hay **SET *`variable`*** general en SQL92 (con la excepción de **SET TRANSACTION ISOLATION LEVEL**). La sintaxis de SQL92 para **SET TIME ZONE** es ligeramente diferente, que permite sólo un único valor entero para la especificación de la zona horaria:

```
SET TIME ZONE { expresión_de_valor_del_intervalo | LOCAL }
```

SHOW

Nombre

`SHOW` — Muestra los parámetros en tiempo de ejecución de la sesión

Synopsis

`SHOW` *`palabra_clave`*

Entradas

`palabra_clave`

Veáse el comando **SET** para obtener más información de los argumentos disponibles.

Outputs

`NOTICE:` *`variable`* `is` *`value`* `SHOW VARIABLE`

Mensaje que se devuelve si todo ha ido bien.

`NOTICE: Unrecognized variable` *`value`*

Mensaje que se devuelve si value no existe.

```
NOTICE: Time zone is unknown SHOW VARIABLE
```

Si las variables de entorno TZ o PGTZ no están definidas.

Descripción

SHOW mostrará la configuración actual de un parámetro en tiempo de ejecución durante una sesión.

A estas variables se les puede asignar un valor usando la declaración **SET** y se puede restaurar su valor por defecto con la declaración **RESET**. Los parámetros y los valores son sensibles a mayúsculas y minúsculas.

Notas

SHOW es una extensión del lenguaje de Postgres.

Veáse **SET**/**RESET** para fijar los valores de una variable.

Utilización

Muestra el estilo de fecha (`DateStyle`):

```
SHOW DateStyle;
NOTICE:DateStyle is Postgres with US (NonEuropean) conventions
```

Muestra la configuración del optimizador genético (`geqo`):

```
SHOW GEQO;
```

```
NOTICE:GEQO is ON
```

Compatibilidad

SQL92

No hay ningún **SHOW** definido en SQL92.

TRUNCATE

Nombre

`TRUNCATE` — Vacia una tabla

Synopsis

`TRUNCATE [ TABLA ] *NOMBRE*`

Entradas

`nombre`

El nombre de la tabla a truncar.

Salidas

`TRUNCATE`

Mensaje retornado si la tabla ha sido vaciada (truncada) exitosamente.

Description

TRUNCATE remueve rapidamente todas las filas de una tabla. Tiene el mismo efecto que el **DELETE** pero al no recorrer la tabla resulta mas rapido. Es mas efectivo en tablas grandes.

Usage

Truncar la tabla `tablagrande`:

```
TRUNCATE TABLE tablagrande;
```

Compatibilidad

SQL92

El **TRUNCATE** no existe en SQL92.

UNLISTEN

Nombre

`UNLISTEN` — Deja de prestar atención a las notificaciones

Synopsis

```
UNLISTEN { nombre_notif | * }
```

Entradas

`nombre_notif`

Nombre de la notificación previamente registrada.

`*`

Se limpiarán todos los registros en escucha para este backend.

Salidas

`UNLISTEN`

Acuse de recibo de que la declaración se ha ejecutado.

Descripción

UNLISTEN se usa para borrar un registro **NOTIFY** existente. UNLISTEN cancela cualquier registro existente de la sesión actual de Postgres en la condición de notificación *`nombre_notif`*. La condición asterisco "*" cancela todos los registros "listener" de la sesión actual.

NOTIFY contiene una discusión más extensa del uso de **LISTEN** y **NOTIFY**.

Notas

`nombre_clase` no necesariamente ha de ser un nombre de clase válido, pero puede ser cualquier cadena (string) válida de hasta 32 caracteres de largo.

El backend no muestra errores si usted hace un UNLISTEN sobre algo al que no estuviera atendiendo (escuchando). Cada backend ejecutará automáticamente **UNLISTEN *** cuando termine.

Una restricción que se daba en versiones anteriores de Postgres, que hacía que un *`nombre_clase`* que no se correspondiera con la tabla en curso debía ser entrecomillada, ya no se da actualmente.

Usage

Para suscribirse a un registro existente:

```
postgres=> LISTEN virtual;
LISTEN
postgres=> NOTIFY virtual;
NOTIFY
ASYNC NOTIFY of ’virtual’ from backend pid ’12317’ received
```

Una vez que UNLISTEN se ha ejecutado, posteriores comandos NOTIFY serán ignorados:

```
postgres=> UNLISTEN virtual;
UNLISTEN
postgres=> NOTIFY virtual;
NOTIFY
- notice no NOTIFY event is received
```

Compatibilidad

SQL92

No existe **UNLISTEN** en SQL92.

UPDATE

Nombre

`UPDATE` — Substituye valores de columnas en una tabla

Synopsis

```
UPDATE tabla SET columna = expresión [, ...]
    [ FROM lista ]
    [ WHERE condición ]
```

Entradas

`table`

El nombre de una tabla existente.

`columna`

El nombre de la columna en *`tabla`*.

`expresión`

Una expresión válida o valor a ser asignado a la columna.

`lista`

Es una extensión no estándar de Postgres que permite la aparición de columnas de otras tablas en la condición WHERE.

condición

Consulte la cláusula SELECT para un descripción mas extensa de la cláusula WHERE.

Salidas

`UPDATE #`

Mensaje obtenido si ha habido éxito. El símbolo # representa el número de filas que han sido actualizadas. Si # es igual a 0, ninguna fila fue actualizada.

Descripción

UPDATE cambia el valor de la columnas especificadas por todas las filas que satisfacen la condición dada. Solamente necesita indicar las columnas que serán modificadas.

Para referencias a listas se usa la misma sintaxis de *SELECT*. O sea, puede substituir un único elemento de una lista, un rango de elementos o una lista completa con una única petición.

Debe tener permiso de escribir en la tabla para poder modificarla, así como permiso de lectura de cualquier tabla cuyos valores sean mencionados en la condición WHERE.

Uso

Para cambiar la palabra "Drama" por "Dramática" en la columna categoría:

```
UPDATE películas
```

```
    SET categoría = 'Dramática'
    WHERE categoría = 'Drama';
SELECT * FROM películas WHERE categoría = 'Dramático' OR categoría = 'Drama'

   code |título            |did|fecha_prod|categoría |durac
   ---+------------+--+-------+-------+----
   BL101|El tercer hombre  |101|1949-12-23|Dramática | 01:44
   P_302|Becket            |103|1964-02-03|Dramática | 02:28
   M_401|La paz y la guerra|104|1967-02-12|Dramática | 05:57
   T_601|Yojimbo           |106|1961-06-16|Dramática | 01:50
   DA101|Das Boot          |110|1981-11-11|Dramática | 02:29
```

Compatibilidad

SQL92

SQL92 define una sintaxis diferente para la cláusula UPDATE:

```
UPDATE tabla SET columna = expresión [, ...]
    WHERE CURRENT OF cursor
```

donde *`cursor`* identifica un cursor abierto.

VACUUM

Nombre

`VACUUM` — Limpia y analiza una base de datos Postgres

Synopsis

```
VACUUM [ VERBOSE ] [ ANALYZE ] [ tabla ]
VACUUM [ VERBOSE ] ANALYZE [ tabla [ (columna [, ...] ) ] ]
```

Entrada

VERBOSE

Imprime un reporte detallado de la actividad de vacuum para cada tabla.

ANALYZE

Actualiza las estadísticas de columnas usadas por el optimizador para determinar la manera más eficiente de ejecutar una consulta. Las estadísticas representan la dispersión de los datos en cada columna. Esta información es valiosa cuando hay la posibilidad de ejecución desde varios puntos.

`tabla`

El nombre de una tabla específica a la que se va a realizar el vacuum. El estándar es hacerlo a todas las tablas.

columna

El nombre de una columna específica a analizar. El estándar es hacerlo para todas las columnas.

Salida

VACUUM

El comando ha sido aceptado y la base de datos está siendo limpiada.

NOTICE: -Relation *tabla*-

El encabezado de reporte para *tabla*.

```
NOTICE: Pages 98: Changed 25, Reapped 74, Empty 0, New 0; Tup
1000: Vac 3000, Crash 0, UnUsed 0, MinLen 188, MaxLen 188;
Re-using: Free/Avail. Space 586952/586952; EndEmpty/Avail. Pages
0/74. Elapsed 0/0 sec.
```

El análisis para la *tabla* misma.

```
NOTICE: Index indice: Página 28; Tuples 1000: Deleted 3000.
Elapsed 0/0 sec.
```

El análisis para un índice en la tabla destino.

Descripción

VACUUM sirve para dos propósitos en Postgres como medio para reclamar almacenamiento, y también para recolectar información para el optimizador.

VACUUM abre cada clase en la base de datos, limpia los registros de transacciones ya pasadas y actualiza las estadísticas en los catálogos del sistema. Las estadísticas mantenidas incluyen el número de tuples y el número de páginas almacenadas en todas las clases.

La ejecución de **VACUUM** periódicamente aumentará la velocidad de la base de datos al procesar las consultas del usuario.

Notas

La base de datos abierta es el objetivo del comando **VACUUM**.

Recomendamos que la base de datos principal activa sea limpiada cada noche para mantener las estadísticas relativamente actualizadas. Sin embargo, la consulta **VACUUM** puede ser ejecutada en cualquier momento. Particularmente, despúes de copiar una clase grande en Postgres o despúes de borrar un gran número de registros, puede ser una buena idea emitir una consulta **VACUUM**. Esto actualizará los catálogos del sistema con todos los cambios recientes, y permitirá al organizador de consultas de Postgres tomar las mejores decisiones al planear las consultas de los usuarios.

Uso

El siguiente es un ejemplo de la ejecución del comando **VACUUM** en una tabla en la base de datos de regresión:

```
regresión=> vacuum verbose analyze onek;
NOTICE:  -Relation onek-
NOTICE:  Pages 98: Changed 25, Reapped 74, Empty 0, New 0;
         Tup 1000: Vac 3000, Crash 0, UnUsed 0, MinLen 188, MaxLen 188;
         Re-using: Free/Avail. Space 586952/586952; EndEmpty/Avail. Pa-
ges 0/74.
         Elapsed 0/0 sec.
NOTICE:  Index onek_stringu1: Pages 28; Tuples 1000: Deleted 3000. Elap-
sed 0/0 sec.
```

```
NOTICE:  Index onek_hundred: Pages 12; Tuples 1000: Deleted 3000. Elap-
sed 0/0 sec.
NOTICE:  Index onek_unique2: Pages 19; Tuples 1000: Deleted 3000. Elap-
sed 0/0 sec.
NOTICE:  Index onek_unique1: Pages 17; Tuples 1000: Deleted 3000. Elap-
sed 0/0 sec.
NOTICE:  Rel onek: Pages: 98 -> 25; Tuple(s) moved: 1000. Elapsed 0/1 sec.
NOTICE:  Index onek_stringu1: Pages 28; Tuples 1000: Deleted 1000. Elap-
sed 0/0 sec.
NOTICE:  Index onek_hundred: Pages 12; Tuples 1000: Deleted 1000. Elap-
sed 0/0 sec.
NOTICE:  Index onek_unique2: Pages 19; Tuples 1000: Deleted 1000. Elap-
sed 0/0 sec.
NOTICE:  Index onek_unique1: Pages 17; Tuples 1000: Deleted 1000. Elap-
sed 0/0 sec.
VACUUM
```

Compatibilidad

SQL92

No existe el comando **VACUUM** en SQL92.

Capítulo 15. Aplicaciones

Esta es la información de referencia para las aplicaciones y utilidades de soporte de Postgres.

createdb

Nombre

`createdb` — Crea una nueva base de datos PostgreSQL

Synopsis

```
createdb [ options ] dbname [ descripcion ]
```

Inputs

-h, –host *`host`*

Especifica el nombre de host (hostname) de la maquina sobre la que esta ejecutandose la postmaster.

-p, –port *`port`*

Especifica el puerto TCP/IP Internet o la extension del fichero de socket del dominio local Unix en el cual la postmaster esta escuchando para recibir

conexiones.

-U, –username *username*

Usuario como el que se conecta.

-W, –password

Fuerza a que se teclee password.

-e, –echo

Muestra la consulta que createdb genera y envia al motor de la base de datos (backend)

-q, –quiet

No muestra ninguna respuesta.

-D, –location *datadir*

especifica localizacion alternativa de la base de datos para esta instalacion de la base de datos. Esta es la localizacion de las tablas del sistema, no la localizacion de esta base de datos especifica, que puede ser diferente.

-E, –encoding *encoding*

Especifica el esquema de codificacion de caracteres que se usara con esta base de datos.

dbname

Especifica el nombre de la base de datos que sera creada. El nombre debe ser unico entre todas las bases de datos PostgreSQL en esta instalacion. El valor por omision es crear una base de datos con el mismo nombre que el usuario en curso del sistema.

description

Opcionalmente esto especifica un comentario que sera asociado con la base de

datos nuevamente creada.

Las opciones `-h`, `-p`, `-U`, `-W`, y `-e` son pasadas literalmente a *psql*.

Outputs

`CREATE DATABASE`

La base de datos fue creada exitosamente.

`createdb: Creacion de la base de datos fallida.`

(Lo dice todo.)

`createdb: Comentario a la creacion fallida. (La base de datos fue creada.)`

El comentario/descripcion para la base de datos que no ha podido ser creada. La base de datos misma podria haber sido creada ya. Puedes utilizar el comando SQL **COMMENT ON DATABASE** para crearle el comentario despues.

Si hay un error en la condicion, el error del motor de base de datos (backend) sera mostrado. Vease *CREATE DATABASE* y *psql* para mas posibilidades.

Descripcion

createdb crea una nueva base de datos PostgreSQL. El usuario que ejecuta este comando se convierte en el propietario de la base de datos.

createdb es una script shell que envuelve un comando SQL *CREATE DATABASE* a traves del terminal interactivo de PostgreSQL *psql*. Asi pues, no hay nada especial sobre la creacion de bases de datos por este u otros metodos. Esto significa que el psql debe ser encontrado por el script y que un servidor de base de datos esta ejecutandose

en el hosts destino. Tambien, cualquier cualquier configuracion por defecto y variable de entorno disponible para psql y la libreria front-end libpq se aplicaran.

Uso

Para crear la base de datos `demo` utilizando el servidor por defecto de base de datos:

```
$ createdb demo
CREATE DATABASE
```

La respuesta es la misma que hubieses tenido de ejecutar el comando de SQL **CREATE DATABASE**.

Para crear una base de datos `demo` utilizando la postmaster en la maquina (host) eden, puerto 5000, utilizando el esquema de codificacion `LATIN1` con una mirada en la consulta subrayada:

```
$ createdb -p 5000 -h eden -E LATIN1 -e demo
CREATE DATABASE "demo" WITH ENCODING = 'LATIN1'
CREATE DATABASE
```

createlang

Nombre

`createlang` — Añade una nuevo lenguaje de programación a una base de datos

PostgreSQL

Synopsis

```
createlang [ opciones_conexion ] [ nom_leng [ nombre_bd ] ]
createlang [ opciones_conexion ] -lista|-l
```

Inputs

createlang acepta los siguientes argumentos:

langname

Especifica el nombre del lenguaje de programación del backend que va a ser definido. createlang preguntará por *langname* si no está definido en la línea de comandos.

[-d, –dbname] *nombre_bd*

Específica a qué base de datos se va a añadir el lenguaje.

-l, –list

Muestra una lista de los lenguajes ya instalados en la base de datos destino (que debe ser especificada).

createlang también acepta los siguientes argumentos en la línea de comandos como parámetros de conexión:

-h, –host *host*

Especifica el nombre de host de la máquina sobre la que postmaster está corriendo.

-p, –puerto *`port`*

Especifica el puerto TCP/IP o el socket del dominio Unix en el que el postmaster está atendiendo a las conexiones.

-U, –nombre usuario *`username`*

Usuario con el que se va a conectar.

-W, –password

Fuerza a que se pregunte el password.

Outputs

La mayoría de los mensaje de error son lo suficientemente explicativos. Si no es así, ejecute createlang cona la opción `-echo` y vea el comando SQL correspondiente para más detalles. Pruebe también bajo *psql* para ver más posibilidades.

Descripción

createlang es una utilidad para añadir un nuevo lenguaje de programación a una base de datos PostgreSQL . Actualmente createlang currently acepta dos lenguajes: `plsql` y `pltcl`.

Aunque los lenguajes de programación del backend pueden ser añadidos directamente usando varios comandos SQL , se recomienda usar createlang porque hace una serie de chequeos y es más fácil de usar. Vea *CREATE LANGUAGE* para más información.

Notas

Utilice *droplang* para borrar un lenguaje.

Uso

Para instalar `pltcl`:

```
$ createlang pltcl
```

createuser

Nombre

`createuser` — Crea un nuevo usuario PostgreSQL

Synopsis

`createuser [ *opciones* ] [ *nombre_usuario* ]`

Inputs

-h, –host *`host`*

Especifica el nombre del host de la máquina sobre la que el postmaster corre.

-p, –puerto *`puerto`*

Especifica el pueto TCP/IP o el socket local Unix sobre el que el postmaster atiende a las conexiones.

-e, –echo

Muestra las consultas que createdb genera y envía al backend.

-q, –quiet

No muestra respuesta alguna.

-d, –createdb

Permite al nuevo usuario crear bases de datos.

-D, –no-createdb

Impide al nuevo usuario crear bases de datos.

-a, –adduser

Permite al nuevo usuario crear otros usuarios.

-A, –no-adduser

Impide al nuevo usuario crear otros usuarios.

-P, –pwprompt

Si se especifica este parámetro, createuser motrará un mensaje preguntando por el password del nuevo usuario. Esto no es necesario si no planea usar autentificación por password.

-i, –sysid *`id_usuario`*

Le permite elegir otro id de usuario que no sea el que se da por defecto. Esto no es necesario, pero a algunos les gusta.

`nombre_usuario`

Especifica el nombre del usuario PostgreSQL que se va a crear. Este nombre debe ser único dentro de todos los existentes en PostgreSQL .

Se le preguntará por un nombre y cualquier otra información que no se haya especificado en la línea de comandos.

Las opciones `-h`, `-p`, y `-e`, son pasadas literalmente a *psql*. Las opciones psql `-U` y `-W` también se pueden usar, pero su uso puede ser confuso en este contexto.

Outputs

`CREATE USER`

Todo ha ido bien.

`createuser: creation of user "`*`username`*`" failed`

Algo no salió bien. El usuario no fue creado.

Si se da un error, el mensaje de error del backend se mostrará. Vea *CREAR USUARIO* y *psql* para más posibilidades.

Descripción

createuser crea un nuevo usuario PostgreSQL . Solamente los usuarios con `usesuper` activado en en la clase `pg_shadow` pueden crear nuevos usuarios Postgres .

createuser es un envoltorio del shell script entorno al comando SQL *CREAR USUARIO* a través del terminal interactivo *psql* de PostgreSQL . Así, no hay nada especial en el momento de crear usuarios por medio de estos otros métodos. Esto significa que psql debe ser encontrado por el script y que que un servidor de bases de datos está corriendo en la máquina al que se accede. Asimismo, cualquier valor por defecto y cualquier variable de entorno disponible para psql y libpq se aplican.

Uso

Para crear un usuario `joe` en la base de datos por defecto:

```
$ createuser joe
Is the new user allowed to create databases? (y/n) n
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER
```

Para crear al mismo usuario `joe` usando el postmaster en la máquina eden, puerto 5000, evitando las preguntas en el prompt y teniendo en cuenta la consulta en curso:

```
$ createuser -p 5000 -h eden -D -A -e joe
CREATE USER "joe" NOCREATEDB NOCREATEUSER
CREATE USER
```

dropdb

Nombre

`dropdb` — Borra una base de datos PostgreSQL existente

Synopsis

```
dropdb [ opciones ] nombre_bd
```

Inputs

-h, –host *host*

Especifica el nombre de host de la máquina sobre la que el postmaster esté corriendo.

-p, –port *puerto*

Especifica el puerto TCP/IP o el socket Unix local en el que postmaster atiende conexiones.

-U, –username *nombre_usuario*

Nombre de usuario con el que se va a conectar.

-W, –password

Fuerza la introducción de un password.

-e, –echo

Muestra en pantalla las consultas que dropdb genera y envía al backend.

-q, –quiet

No muestra respuesta alguna.

-i, –interactive

Antes de hacer algo destructivo, pide confirmación a través del prompt.

nombre_bd

Especifica el nombre de la bases de datos que va a ser borrada. Debe ser una de las existentes en esta instalación de PostgreSQL .

Las opciones `-h`, `-p`, `-U`, `-W`, y `-e` se pasan literalmente a *psql*.

Outputs

`DROP DATABASE`

La base de datos ha sido borrada con éxito.

`dropdb: Database removal failed.`

Algo no ha ido bien.

Si se produce un error, se mostrará el mensaje de error del backend. Vea drop_database y *psql* para más información.

Descripción

dropdb destruye una base de datos PostgreSQL existente. El usuario que ejecute este comando debe ser un superusuario de la base de datos o su propietario.

dropdb es un envoltorio del shell script alrededor del comando SQL drop_database por medio del terminal interactivo *psql* de PostgreSQL. De este modo, no hay nada especial en en borrar bases de datos por medio de este u otros métodos. Esto significa que psql debe ser encontrado por el script y que un servidor de bases de datos está en marcha en el host de destino. También cualquier valor por defecto o cualquier variable de entorno disponible para psql y libpq se aplican.

Uso

Para destruir la base de datos `demo` en el servidor de bases de datos por defecto:

```
$ dropdb demo
```

```
DROP DATABASE
```

Para destruir la base de datos `demo` usando el postmaster del host eden, puerto 5000, con verificación y echando un vistazo a la consulta en marcha:

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE "demo"
DROP DATABASE
```

droplang

Nombre

`droplang` — Borra un lenguaje de programación de una base de datos PostgreSQL

Synopsis

```
droplang [ opciones de conexión ] [ nombre_lenguaje [ nombre_bd ] ]
droplang [ opciones de conexión ] -list|-l
```

Inputs

droplang acepta los siguientes argumento en la línea de comandos:

nombre_lenguaje

Especifica el nombre del lenguaje de programación del backend que se va a borrar. droplang preguntará por *nombre_lenguaje* si no se ha especificado en la línea de comandos.

[-d, –dbname] *nombre_bd*

Especifica desde qué base de datos se debe borrar el lenguaje.

-l, –list

Muestra la lista de los lenguajes ya instalados en la base de datos destino (que debe ser especificada).

droplang también acepta los siguientes argumentos en la línea de comandos como parámetros de conexión:

-h, –host *host*

Especifica el nombre de host de la máquina sobre la que corre postmaster.

-p, –port *puerto*

Especifica el puerto TCP/IP o el socket local sobre el que postmaster atiende a las conexiones.

-U, –username *nombre_usuario*

Nombre de usuario con el que se vaya a conectar.

-W, –password

Fuerza la utilización de un password.

Outputs

La mayoría de los mensajes de error se explican por sí solos. Si no es asi, ejecútelo droplang con la opción `-echo` y mire bajo el comando SQL correspondiente para más detalles. Mire también bajo *psql*.

Descripción

droplang es una utilidad para borrar un lenguaje de programación existente en la base de datos PostgreSQL. droplang actualmente acepta dos lenguajes: `plsql` y `pltcl`.

Aunque los lenguajes de programación del backend pueden ser borrados directamente utilizando varios comandos SQL, es recomendable usar droplang porque realiza comprobaciones y es más fácil de usar. Vea *DROP LANGUAGE* para más detalles.

Notas

Utilice *createlang* para agregar un lenguaje.

Uso

Para borrar `pltcl`:

```
$ droplang pltcl
```

dropuser

Nombre

`dropuser` — Borra un usuario Postgres

Synopsis

`dropuser [ *opciones* ] [ *nombre_usuario* ]`

Inputs

-h, –host *`host`*

Especifica el nombre de host de la máquina en la que el postmaster se está ejecutanto.

-p, –port *`puerto`*

Especifica el puerto TCP/IP el socket local sobre el que postmaster escucha conexiones.

-e, –echo

Muestra en pantalla las consultas que createdb genera y envía al backend.

-q, –quiet

No muestra respuesta alguna.

-i, –interactive

Antes de borrar al usuario, pregunta.

nombre_usuario

Especifica el nombre de usuario PostgreSQL que va a ser borrado. Este nombre debe existir en la instalación Postgres. Se le preguntará un nombre si no se ha especificado ninguno en la línea de comandos.

Las opciones `-h`, `-p`, y `-e`, son pasadas literalmente a *psql*. Las opciones psql `-U` y `-W` también están disponibles, pero pueden ser confusas en este contexto.

Outputs

`DROP USER`

Todo ha ido bien.

`dropuser: deletion of user "`*username*`" failed`

Algo salió mal. No se ha borrado al usuario.

Cuando se da un error, el mensaje de error del backend será mostrado. Vea *DROP USER* y *psql* para más posibilidades.

Descripción

dropuser borrar un usuario PostgreSQL existente *y* las bases de datos que ese usuario posee. Solamente los usuarios con `usesuper` activado en la clase `pg_shadow` pueden destruir usuarios de PostgreSQL.

dropuser es un envoltorio del shell script alrededor del comando SQL *DROP USER* por medio del terminal interactivo *psql* de PostgreSQL. De este modo, no hay nada especial en en borrar bases de datos por medio de este u otros métodos. Esto significa que psql debe ser encontrado por el script y que un servidor de bases de datos está en marcha en

el host de destino. También cualquier valor por defecto o cualquier variable de entorno disponible para psql y libpq se aplican.

Uso

Para borrar al usuario `joe` del servidor de bases de datos por defecto:

```
$ dropuser joe
DROP USER
```

Para borrar al usuario `joe` usando el postmaster en el host eden, puerto 5000, con verificación y echando un vistazo a la consulta en curso:

```
$ dropuser -p 5000 -h eden -i -e joe
User "joe" and any owned databases will be permanently deleted.
Are you sure? (y/n) y
DROP USER "joe"
DROP USER
```

ecpg

Nombre

`ecpg` — Embedded SQL C preprocessor (preprocesador C incorporado en SQL)

Synopsis

```
ecpg [ -v ] [ -t ] [ -I include-path ] [ -o outfile ]  file1 [ file2 ] [ ...
```

Inputs

ecpg acepta los siguiente argumentos en línea de comandos:

`file`

Outputs

ecpg creará un fichero o escribirá en `stdout` (salida estándar).

Descripción

pgaccess

Nombre

`pgaccess` — Cliente gráfico interactivo dePostgres

Synopsis

```
pgaccess [ dbname ]
```

Entradas

dbname

El nombre de una base de datos existente.

Salidas

Descripción

pgaccess proporciona una interfaz gráfica para Postgres donde se pueden gestionar las tablas, editarlas, definir consultas, secuencias y funciones.

Otra forma de acceder a Postgres a través de tcl es con el uso de *pgtclsh* o *pgtksh*.

pgaccess permite:

- Abrir cualquier bases de datos en un determinado host, especificando dicho host, el puerto,n el puerto especificado, el nombre de usuario y password.
- Ejecutar *VACUUM*.
- Guardar preferencias en el archivo `~/.pgaccessrc`.

Con tablas, pgaccess permite:

- Abrir múltiples tablas para visualización, con un máximo de n registros (configurable).
- Cambiar el tamaño de las columnas desplazando las líneas verticales que la forman.
- Introducir texto en celdas.
- Ajustar dinámicamente la altura de la celda durante la edición.
- Guardar un formato de tabla para cada tabla.
- Importar / exportar a/de archivos externos (SDF,CSV).
- Usar filtros; introducir filtros como precio>3.14.
- Especificar el orden; introducir manualmente los campos por los que realizar la ordenación.
- Edición; doble click sobre el texto que queremos cambiar.
- Borrar registros; situándose en el registro, se pulsa la tecla Del.
- Añadir nuevos registros; guardar nuevo registro con el botón derecho del ratón.
- Crear tablas con un asistente.
- Renombrar y borrar (drop) tablas.
- Recuperar información sobre las tablas, incluyendo propietario, información de los campos, índices.

Con consultas, pgaccess permite:

- Definir, editar y almacenar *user defined queries*.
- Guardar formatos de vistas.
- Almacenar consultas como vistas.
- Ejecutar con parámetros opcionales de entrada introducidos por el usuario; p.ej.

```
select * from invoices where year=[parameter "Year of selection"]
```

- Visualizar cualquier resultado de una consulta de selección (select).
- Ejecutar consultas de acción (insert, update, delete).
- Definir consultas usando un constructor visual de consultas con soporte drag & frop, y aliasing de tablas.

Con secuencias, pgaccess permite:

- Definir nuevas instancias.
- Inspeccionar instancias existentes.
- Borrar.

Con vistas, pgaccess permite:

- Definirlas salvando consultas como vistas.
- Visualizarlas, con posibilidades de ordenación y filtrado.
- Diseñar nuevas vistas.
- Borrar (drop) vistas existentes.

Con funciones , pgaccess permite:

- Definirlas.
- Inspeccionarlas.
- Borrarlas.

Con informes, pgaccess permite:

- Generar informes simples desde una tabla (beta stage).
- Cambiar fuente, tamaño y estilo de campos y etiquetas.
- Cargar y guardar informes de la base de datos.

- Previsualizar tablas, muestras de impresiones postscript.

Con formularios, pgaccess permite:

- Abrir formularios definidos por el usuario.
- Usar un módulo de diseño de formularios.
- Acceder a conjuntos de registros usando widget de consultas.

Con scripts, pgaccess permite:

- Definirlos.
- Modificarlos.
- Llamar scripts definidos por el usuario.

pgadmin

Nombre

`pgadmin` — Postgres es una herramienta de diseño y mantenimiento de bases de datos para Windows 95/98/NT

Synopsis

`pgadmin [ nombre de datosfuente [ nombre de usuario [ palabraclave ] ] ]`

Entradas

`nombre de datosfuente`

El nombre de un sistem ODBC PostgreSQL existente o Datos fuente del Usuario.

`nombre de usuario`

Un nombre de usuario válido para el especificado *`nombre de datosfuente`*.

`palabra clave`

Una palabra clave válida para el especificado *`nombre de datosfuente`* y *`nombre de usuario`*.

Resultados

Descripción

pgadmin es una herramienta de propósito general para diseñár, mantener, y administrar las bases de datos de Postgres. Funciona bajo Windows 95/98 y NT.

Características incluidas:

- Entradas SQL aleatorias.
- Pantallas de información y 'Ayudas' para bases de datos, tablas, indices, secuencias, vistas, programas de arranque, funciones y lenguajes.
- Preguntas y respuestas para configurar Usuarios, Grupos y Privilegios.
- Control de revisión con mejora de la generación de script.
- Configuración de las tablas de Microsoft MSysConf.

- ‘Ayudas´ para importar y exportar datos.
- ‘Ayuda´ para migrar Bases de datos.
- Informes predefinidos en bases de datos, tablas, índices, secuencias, lenguajes y vistas.

pgadmin se distribuye separadamente de Postgres y puede ser descargado desde la dirección http://www.pgadmin.freeserve.co.uk

pg_dump

Nombre

`pg_dump` — Extrae una base de datos Postgres a un fichero de script

Synopsis

```
pg_dump [ base_de_datos ]
pg_dump [ -h huésped ] [ -p puerto ]
    [ -t tabla ]
    [ -a ] [ -c ] [ -d ] [ -D ] [ -n ] [ -N ]
    [ -o ] [ -s ] [ -u ] [ -v ] [ -x ]
    [ base_de_datos ]
```

Entrada

pg_dump accepta los siguientes argumentos de la línea de comando:

base_de_datos

Epecifíca el nombre de la base de datos que se va a extraer. *base_de_datos* tiene como estándar el valor de la variable de entorno USER

-a

Vuelca sólo los datos, no el esquema (las definiciones).

-c

Limpia el esquema antes de crearlo.

-d

Vuelca la data como propios insertos de cadenas.

-D

Vuelca la data como insertos con nombres de atributos

-n

Suprime las dobles comillas de los identificadores, a menos que sean absolutamente necesarias. Esto puede causar problemas al cargar la misma si esta data volcada contiene palabras reservadas usadas por los identificadores. Esta era la conducta estándar en pg_dump pre-v6.4.

-N

Incluye comillas dobles en los identificadores. Este es el estándar.

-o

Vuelca los identificadores de objetos (OIDs) para cada tabla.

-s

Vuelca solo el esquema (las definiciones), no la data.

-t *tabla*

Vuelca la data para la *tabla* únicamnete.

-u

Usa autenticación por medio de clave de acceso. Pide un nombre de usuario y clave de acceso.

-v

Epecifica el modo verbose(parlanchín)

-x

Evita el volcado de ACLs (comandos grant/revoke) y la información de propiedad de la tabla.

pg_dump también acepta los siguientes argumentos de línea de comando para parámetros de conexión:

-h *huésped*

Especifica el nombre del huésped de la máquina en la cual se está ejecutando el postmaster. El estándar es usar un socket de dominio local Unix en vez de una conexión IP..

-p *puerto*

Especifica el puerto de Internet TCP/IP o extensión de archivo socket de dominio local Unix en el cual postmaster está esperando que se efectúen conexiones. En número estándar de puerto es 5432, o el valor de la variable de ambiente PGPORT (si está establecida).

-u

Usa autenticación con clave de acceso. Pide *nombre_de_usuario* y *clave_de_acceso*.

Salida

pg_dump creará un fichero o escribirá a `stdout`.

`La conexión con la base de datos 'template1' falló. connectDB() falló: ¿Está el postmaster ejecutándose y acceptando conexiones en el 'Socket de UNIX' en el puerto 'puerto'?`

pg_dump no pudo unirse al proceso postmaster en el huésped y puerto especificados. Si ve usted este mensaje, verifique que postmaster se este ejecutando en el huésped indicado, y que usted especificó el puerto correcto. Si su site usa algún sistema de autenticación, verifique que usted tiene las credenciales de autenticación requeridas.

`La conexión con la base de datos 'base_de_datos' falló. FATAL 1: SetUserId: el usuario 'nombre_de_usuario' no está en 'pg_shadow'`

Usted no posee una entrada válida en la relación `pg_shadow` y no le será permitido tener acceso a Postgres. Contacte a su administrador de Postgres.

`dumpSequence(tabla): SELECT falló`

Usted carece del permiso para leer la base de datos. Contacte a su administrador de site Postgres.

Nota: pg_dump ejecuta internamente las directivas **SELECT**. Si tiene problemas ejecutando pg_dump, verifique que puede seleccionar la información de la base de datos mediante el uso de, por ejemplo, psql.

Descripción

pg_dump es un utilitario para volcar una base de datos Postgres en un fichero de script conteniendo comandos de consulta. Los ficheros de script son en formato de texto y pueden ser usados para reconstruír la base de datos, incluso en otras máquinas y con otras arquitecturas. pg_dump producirá las consultas necesarias para regenerar todos los tipos definidos por el usuario, funciones, tablas, índices, agregados, y operadores. Adicionalmente, toda la data es copiada en formato de texto el cual puede ser nuevamente copiado, también puede ser importado a herramientas para su edición.

pg_dump es útil para verter el contenido de una base de datos que se vaya a mudar de una instalación de Postgres a otra. Después de ejecutar pg_dump, se debe examinar el script de salida a ver si contiene alguna advertencia, especialmente a la luz de las limitaciones citadas en la parte inferior.

Notas

pg_dump tiene pocas limitaciones. Las limitaciones surgen principalmente de la dificultad para extraer ciertas meta-informaciones de los catálogos del sistema.

- pg_dump no entiende los índices parciales. La razón es la misma citada anteriormente; los predicados de los índices parciales se almacenan como planos.
- pg_dump no maneja objetos grandes. Los objetos grandes son ignorados y se debe lidiar con ellos de forma manual.

Uso

Para volcar una base de datos del mismo nombre que el usuario:

```
% pg_dump > db.out
```

Para volver a cargar esta base de datos:

```
% psql -e base_de_datos < db.out
```

pg_dumpall

Nombre

`pg_dumpall` — Extrae todas las bases de datos Postgres en un archivo de script

Synopsis

```
pg_dumpall
pg_dumpall [ -h máquina ] [ -p puerto ] [ -a ] [ -d ] [ -D ] [ -O ] [ -
s ] [ -u ] [ -v ] [ -x ]
```

Entradas

pg_dumpall acepta los siguientes argumentos de la línea de órdenes:

-a

Vuelca sólo los datos, no el esquema (las definiciones).

-d

Vuelca los datos como inserciones de cadenas adecuadas.

-D

Vuelca los datos como inserciones con nombres de atributos

-n

Suprime las dobles comillas de los identificadores, a menos que sean absolutamente necesarias. Esto puede causar problemas al cargar estos datos volcados si hay palabras reservadas usadas como identificadores.

-o

Vuelca los identificadores de objetos (OIDs) de cada tabla.

-s

Vuelca sólo el esquema (las definiciones), no los datos.

-u

Usa autenticación con clave de acceso. Pide un nombre de usuario y una clave de acceso.

-v

Especifíca el modo verbose (detallado)

-x

Evita el volcado de ACLs (órdenes grant/revoke) e información del propietario de la tabla.

pg_dumpall también acepta los siguientes argumentos en la línea de órdenes como parámetros de conexión:

-h *huésped*

especifica el nombre de la máquina en la cual se está ejecutando postmaster. El estándar es usar un socket de dominio local Unix en vez de una conexión IP.

-p *puerto*

Especifica el puerto Internet TCP/IP o el fichero de dominio local Unix en el cual esté postmaster aguardando conexiones. El número estándar de puerto es 5432, o el valor de la variable de entorno PGPORT (si se ha indicado).

-u

Usa autenticación con clave de acceso. Pide *nombre_de_usuario* y *clave_de_acceso*.

Salida

pg_dumpall creará un fichero o escribirá a `stdout`.

`La conexión a la base de datos 'template1' falló. connectDB() falló: ¿Está postmaster ejecutándose y aceptando conexiones en el 'Socket UNIX' en el puerto '`*puerto*`'?`

pg_dumpall no pudo unirse al proceso postmaster en la máquina y puerto especificados. Si ve usted este mensaje, verifique que postmaster esté ejecutándose

correctamente en el huésped y puerto que usted especificó. Si su lugar de trabajo usa algún sistema de autenticación verifique que usted ha obtenido las credenciales de autenticación.

`La conexión a la base de datos 'base_de_datos' falló. FATAL 1: SetUserId: el usuario 'nombre_de_usuario' no está en 'pg_shadow'`

Usted no tiene una entrada válida en la relación `pg_shadow` y no le será permitido el acceso a Postgres. Contacte con su administrador Postgres.

`dumpSequence(tabla): SELECT falló`

No tiene permiso para leer la base de datos. Contacte a su administrador Postgres.

Nota: pg_dumpall ejecuta internamente directivas **SELECT**. Si tiene problemas ejecutando pg_dumpall, asegúrese de que puede consultar información de la base de datos usando, por ejemplo, psql.

Descripción

pg_dumpall se diseñó para volcar todas las bases de datos Postgres en un fichero. También vuelca la tabla pg_shadow, la cual es global para todas las bases de datos. pg_dumpall incluye en este archivo las órdenes correctas para crear automáticamente cada una de las bases de datos volcadas antes de cargar los datos.

pg_dumpall toma todas las opciones de pg_dump pero `-f`, `-t` y *`base_de_datos`* deberían ser omitidos.

Refiérase a *pg_dump* para más información con respecto a esta otra utilidad.

Uso

Para volcar todas las bases de datos:

```
% pg_dumpall > db.out
```

> **Sugerencia:** Puede usar la mayoría de las opciones de pg_dump con pg_dumpall.

Para volver a cargar esta base de datos:

```
% psql -e template1 < db.out
```

> **Sugerencia:** Puede usar la mayoría de las opciones de psql cuando vuelva a cargarlas.

psql

Nombre

`psql` — PostgreSQL interactive terminal

Synopsis

```
psql [ options ] [ dbname [ user ] ]
```

Summary

psql is a terminal-based front-end to PostgreSQL. It enables you to type in queries interactively, issue them to PostgreSQL, and see the query results. Alternatively, input can be from a file. In addition, it provides a number of meta-commands and various shell-like features to facilitate writing scripts and automating a wide variety of tasks.

Description

Connecting To A Database

psql is a regular PostgreSQL client application. In order to connect to a database you need to know the name of your target database, the hostname and port number of the server and what user name you want to connect as. psql can be told about those parameters via command line options, namely `-d`, `-h`, `-p`, and `-U` respectively. If an argument is found that does not belong to any option it will be interpreted as database name as well. Not all these options are required, defaults do apply. If you omit the host name psql will connect via domain sockets to a server on the local host. The default port number is compile-time determined. Since the database server uses the same default, chances are you don’t have to specify the port in most settings. The default user name is your Unix username, the same with the database. Note that you can’t just connect to any database under any username. Your database administrator should have informed you about your access rights. To save you some typing you can also set the environment variables PGDATABASE, PGHOST, PGPORT, PGUSER, respectively to appropriate values.

If the connection could not be made for any reason (e.g., insufficient privileges, postmaster is not running on the server, etc.), psql will return an error and terminate.

Entering Queries

In normal operation, psql provides a prompt with the name of the database that psql is currently connected to followed by the string "=>". For example,

```
$ psql testdb
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

testdb=>
```

At the prompt, the user may type in SQL queries. Ordinarily, input lines are sent to the backend when a query-terminating semicolon is reached. An end of line does not terminate a query! Thus queries can be spread over serveral lines for clarity. If the query was sent and without error, the query results are displayed on the screen.

Whenever a query is executed, psql also polls for asynchronous notification events generated by *LISTEN* and *NOTIFY*.

psql Meta-Commands

Anything you enter in psql that begins with an unquoted backslash is a psql meta-command that is processes by psql itself. These commands are what makes psql interesting for administration or scripting. Meta-commands are more commonly called slash or backslash commands.

The format of a psql command is the backslash, followed immediately by a command verb, then any arguments. The arguments are separated from the command verb and each other by any number of white space characters.

To include whitespace into an argument you must quote it with a single quote. To include a single quote into such an argument, preceed it by a backslash. Anything contained in single quotes is furthermore subject to C-like substitutions for `\n` (new line), `\t` (tab), `\`*`digits`*, `\0`*`digits`*, and `\0x`*`digits`* (the character with the given decimal, octal, or hexadecimal code).

If an unquoted argument begins with a colon (`:`), it is taken as a variable and the value of the variable is taken as the argument instead.

Arguments that are quoted in "backticks" (`‘`) are taken as a command line that is passed to the shell. The output of the command (with a trailing newline removed) is taken as the argument value. The above escape sequences also apply in backticks.

Some commands take the name of an SQL identifier (such as a table name) as argument. These arguments follow the syntax rules of SQL regarding double quotes: an identifier without double quotes is coerced to lower-case. For all other commands double quotes are not special and will become part of the argument.

Parsing for arguments stops when another unquoted backslash occurs. This is taken as the beginning of a new meta-command. The special sequence `\\` (two backslashes) marks the end of arguments and continues parsing SQL queries, if any. That way SQL and psql commands can be freely mixed on a line. But in any case, the arguments of a meta-command cannot continue beyond the end of the line.

The following meta-commands are defined:

`\a`

If the current table output format is unaligned, switch to aligned. If it is not unaligned, set it to unaligned. This command is kept for backwards compatibility. See **pset** for a general solution.

`\C` [*`title`*]

Set the title of any tables being printed as the result of a query or unset any such title. This command is equivalent to `\pset title` *`title`*. (The name of this command derives from "caption", as it was previously only used to set the caption in an HTML table.)

`\connect` (or `\c`) [*`dbname`* [*`username`*]]

Establishes a connection to a new database and/or under a user name. The previous connection is closed. If *`dbname`* is `-` the current database name is assumed.

If *`username`* is omitted the current user name is assumed.

As a special rule, **\connect** without any arguments will connect to the default database as the default user (as you would have gotten by starting psql without any arguments).

If the connection attempt failed (wrong username, access denied, etc.) the previous connection will be kept if and only if psql is in interactive mode. When executing a non-interactive script, processing will immediately stop with an error. This distinction was chosen as a user convenience against typos on the one hand, and a safety mechanism that scripts are not accidentally acting on the wrong database on the other hand.

`\copy` *`table`* [`with oids`] { `from` | `to` } *`filename`* | stdin | stdout [`with delimiters '`*`characters`*`' ] [ with null as '`*`string`*`' ]`

Performs a frontend (client) copy. This is an operation that runs an SQL *COPY* command, but instead of the backend reading or writing the specified file, and consequently requiring backend access and special user privilege, as well as being bound to the file system accessible by the backend, psql reads or writes the file and routes the data to or from the backend onto the local file system.

The syntax of the command is in analogy to the SQL **COPY** command, see its description for the details. Note that because of this, special parsing rules apply to the **\copy** command. In particular, the variable substitution rules and backslash escapes do not apply.

> **Sugerencia:** This operation is not as efficient as the SQL **COPY** command because all data must pass through the client/server IP or socket connection. For large amounts of data the other technique may be preferable.

> **Nota:** Note the difference in interpretation of `stdin` and `stdout` between frontend and backend copies: In a frontend copy these always refer to psql's input and output stream. On a backend copy `stdin` comes from whereever the **COPY** itself came from (for example, a script ran with the `-f`) option, and `stdout` refers to the query output stream (see **\o** meta-command below).

`\copyright`

Shows the copyright and distribution terms of PostgreSQL.

`\d` *`relation`*

Shows all columns of *`relation`* (which could be a table, view, index, or sequence), their types, and any special attributes such as `NOT NULL` or defaults, if any. If the relation is, in fact, a table, any defined indices are also listed. If the relation is a view, the view definition is also shown.

The command form `\d+` is identical, but any comments associated with the table columns are shown as well.

> **Nota:** If **\d** is called without any arguments, it is equivalent to **\dtvs** which will show a list of all tables, views, and sequences. This is purely a convenience measure.

`\da [` *`pattern`* `]`

Lists all available aggregate functions, together with the data type they operate on. If *`pattern`* (a regular expression) is specified, only matching aggregates are shown.

`\dd [` *`object`* `]`

Shows the descriptions of *`object`* (which can be a regular expression), or of all objects if no argument is given. ("Object" covers aggregates, functions, operators, types, relations (tables, views, indices, sequences, large objects), rules, and triggers.) For example:

```
=> \dd version
              Object descriptions
  Name   |   What   |        Description
------+-------+-----------------
 version | function | PostgreSQL version string
(1 row)
```

Descriptions for objects can be generated with the **COMMENT ON** SQL command.

> **Nota:** PostgreSQL stores the object descriptions in the pg_description system table.

`\df [` *`pattern`* `]`

Lists available functions, together with their argument and return types. If *`pattern`* (a regular expression) is specified, only matching functions are shown. If the form `\df+` is used, additional information about each function, including language and description is shown.

`\distvS [` *`pattern`* `]`

This is not the actual command name: The letters i, s, t, v, S stand for index, sequence, table, view, and system table, respectively. You can specify any or all of them in any order to obtain a listing of them, together with who the owner is.

If *pattern* is specified, it is a regular expression restricts the listing to those objects whose name matches. If one appends a “+” to the command name, each object is listed with its associated description, if any.

`\dl`

This is an alias for **\lo_list**, which shows a list of large objects.

`\do [ pattern ]`

Lists available operators with their operand and return types. If *pattern* is specified, only operators with that name will be shown. (Since this is a regular expression, be sure to quote all special characters in you operator name with backslashes. To prevent interpretation of the backslash as a new command, you might also wish to quote the argument.)

`\dp [ pattern ]`

This is an alias for **\z** which was included for its greater mnemonic value (“display permissions”).

`\dT [ pattern ]`

Lists all data types or only those that match *pattern*. The command form `\dT+` shows extra information.

`\edit` (or `\e`) [*filename*]

If *filename* is specified, the file is edited and after the editor exit its content is copied back to the query buffer. If no argument is given, the current query buffer is copied to a temporary file which is then edited in the same fashion.

The new query buffer is then re-parsed according to the normal rules of psql, where the whole buffer is treated as a single line. (Thus you cannot make “scripts” this way, use **\i** for that.) This means also that if the query ends with (or rather contains) a semicolon, it is immediately executed. In other cases it will merely wait in the query buffer.

> **Sugerencia:** psql searches the environment variables PSQL_EDITOR, EDITOR, and VISUAL (in that order) for an editor to use. If all of them are unset, `/bin/vi` is run.

`\echo` *`text`* [...]

Prints the arguments to the standard output, separated by one space and followed by a newline. This can be useful to intersperse information in the output of scripts. For example:

```
=> \echo 'date'
Tue Oct 26 21:40:57 CEST 1999
```

If the first argument is an unquoted `-n` the the trailing newline is not written.

> **Sugerencia:** If you use the **\o** command to redirect your query output you may wish to use **\qecho** instead of this command.

`\encoding` [*`encoding`*]

Sets the client encoding, if you are using multibyte encodings. Without an argument, this command shows the current encoding.

`\f` [*`string`*]

Sets the field separator for unaligned query output. The default is "`|`" (a "pipe" symbol). See also **\pset** for a generic way of setting output options.

`\g` [{ *`filename`* | `|`*`command`* }]

Sends the current query input buffer to the backend and optionally saves the output in *`filename`* or pipes the output into a separate Unix shell to execute

command. A bare \g is virtually equivalent to a semicolon. A \g with argument is a "one-shot" alternative to the **\o** command.

\help (or \h) [*command*]

Give syntax help on the specified SQL command. If *command* is not specified, then psql will list all the commands for which syntax help is available. If *command* is an asterisk ("*"), then syntax help on all SQL commands is shown.

> **Nota:** To simplify typing, commands that consists of several words do not have to be quoted. Thus it is fine to type **\help alter table**.

\H

Turns on HTML query output format. If the HTML format is already on, it is switched back to the default aligned text format. This command is for compatibility and convenience, but see **\pset** about setting other output options.

\i *filename*

Reads input from the file *filename* and executes it as though it had been typed on the keyboard.

> **Nota:** If you want to see the lines on the screen as they are read you must set the variable ECHO to all.

\l (or \list)

List all the databases in the server as well as their owners. Append a "+" to the command name to see any descriptions for the databases as well. If your

PostgreSQL installation was compiled with multibyte encoding support, the encoding scheme of each database is shown as well.

`\lo_export` *`loid filename`*

Reads the large object with OID *`loid`* from the database and writes it to *`filename`*. Note that this is subtly different from the server function `lo_export`, which acts with the permissions of the user that the database server runs as and on the server's file system.

Sugerencia: Use **\lo_list** to find out the large object's OID.

Nota: See the description of the LO_TRANSACTION variable for important information concerning all large object operations.

`\lo_import` *`filename`* [*`comment`*]

Stores the file into a PostgreSQL "large object". Optionally, it associates the given comment with the object. Example:

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

The response indicates that the large object received object id 152801 which one ought to remember if one wants to access the object ever again. For that reason it is recommended to always associate a human-readable comment with every object. Those can then be seen with the **\lo_list** command.

Note that this command is subtly different from the server-side `lo_import` because it acts as the local user on the local file system, rather than the server's user and file system.

> **Nota:** See the description of the LO_TRANSACTION variable for important information concerning all large object operations.

`\lo_list`

Shows a list of all PostgreSQL “large objects” currently stored in the database along with their owners.

`\lo_unlink` *`loid`*

Deletes the large object with OID *`loid`* from the database.

> **Sugerencia:** Use **\lo_list** to find out the large object’s OID.

> **Nota:** See the description of the LO_TRANSACTION variable for important information concerning all large object operations.

`\o [ {`*`filename`*`| |`*`command`*`} ]`

Saves future query results to the file *`filename`* or pipe future results into a separate Unix shell to execute *`command`*. If no arguments are specified, the query output will be reset to `stdout`.

“Query results” includes all tables, command responses, and notices obtained from the database server, as well as output of various backslash commands that query the database (such as **\d**), but not error messages.

> **Sugerencia:** To intersperse text output in between query results, use **\qecho**.

`\p`

Print the current query buffer to the standard output.

`\pset` *`parameter`* [*`value`*]

This command sets options affecting the output of query result tables. *`parameter`* describes which option is to be set. The semantics of *`value`* depend thereon.

Adjustable printing options are:

`format`

Sets the output format to one of `unaligned`, `aligned`, `html`, or `latex`. Unique abbreviations are allowed. (That would mean one letter is enough.)

"Unaligned" writes all fields of a tuple on a line, separated by the currently active field separator. This is intended to create output that might be intended to be read in by other programs (tab-separated, comma-separated). "Aligned" mode is the standard, human-readable, nicely formatted text output that is default. The "HTML" and "LaTeX" modes put out tables that are intended to be included in documents using the respective mark-up language. They are not complete documents! (This might not be so dramatic in HTML, but in LaTeX you must have a complete document wrapper.)

`border`

The second argument must be a number. In general, the higher the number the more borders and lines the tables will have, but this depends on the particular format. In HTML mode, this will translate directly into the `border=...` attribute, in the others only values 0 (no border), 1 (internal dividing lines), and 2 (table frame) make sense.

`expanded` (or `x`)

Toggles between regular and expanded format. When expanded format is enabled, all output has two columns with the field name on the left and the data on the right. This mode is useful if the data wouldn't fit on the screen in the normal "horizontal" mode.

Expanded mode is support by all four output modes.

`null`

The second argument is a string that should be printed whenever a field is null. The default is not to print anything, which can easily be mistaken for, say, an empty string. Thus, one might choose to write `\pset null "(null)"`.

`fieldsep`

Specifies the field separator to be used in unaligned output mode. That way one can create, for example, tab- or comma-separated output, which other programs might prefer. To set a tab as field separator, type `\pset fieldsep "\t"`. The default field separator is "`|`" (a "pipe" symbol).

`recordsep`

Specifies the record (line) separator to use in unaligned output mode. The default is a newline character.

`tuples_only` (or `t`)

Toggles between tuples only and full display. Full display may show extra information such as column headers, titles, and various footers. In tuples only mode, only actual table data is shown.

`title` [*`text`*]

Sets the table title for any subsequently printed tables. This can be used to give your output descriptive tags. If no argument is given, the title is unset.

> **Nota:** This formerly only affected HTML mode. You can now set titles in any output format.

`tableattr` (or `T`) [*`text`*]

Allows you to specify any attributes to be places inside the HTML table tag. This could for example be `cellpadding` or `bgcolor`. Note that you probably don't want to specify `border` here, as that is already taken care of by `\pset border`.

`pager`

Toggles the list of a pager to do table output. If the environment variable PAGER is set, the output is piped to the specified program. Otherwise `more` is used.

In any case, psql only uses the pager if it seems appropriate. That means among other things that the output is to a terminal and that the table would normally not fit on the screen. Because of the modular nature of the printing routines it is not always possible to predict the number of lines that will actually be printed. For that reason psql might not appear very discriminating about when to use the pager and when not to.

Illustrations on how these different formats look can be seen in the *Examples* section.

> **Sugerencia:** There are various shortcut commands for **\pset**. See **\a**, **\C**, **\H**, **\t**, **\T**, and **\x**.

> **Nota:** It is an error to call **\pset** without arguments. In the future this call might show the current status of all printing options.

`\q`

Quit the psql program.

`\qecho` *`text`* [...]

This command is identical to **\echo** except that all output will be written to the query output channel, as set by **\o**.

`\r`

Resets (clears) the query buffer.

`\s` [*`filename`*]

Print or save the command line history to *`filename`*. If *`filename`* is omitted, the history is written to the standard output. This option is only available if psql is configured to use the GNU history library.

> **Nota:** As of psql version 7.0 it is no longer necessary, in fact, to save the command history as that will be done automatically on program termination. The history is then also automatically loaded every time psql starts up.

`\set` [*`name`* [*`value`* [...]]]

Sets the internal variable *`name`* to *`value`* or, if more than one value is given, to the concatenation of all of them. If no second argument is given, the variable is just set with not value. To unset a variable, use the **\unset** command.

Valid variable names can contain characters, digits, and underscores. See the section about psql variables for details.

Although you are welcome to set any variable to anything you want to, psql treats several variables special. They are documented in the section about variables.

Nota: This command is totally separate from the SQL command *SET*.

`\t`

Toggles the display of output column name headings and row count footer. This command is equivalent to `\pset tuples_only` and is provided for convenience.

`\T` *`table_options`*

Allows you to specify options to be placed within the table tag in HTML tabular output mode. This command is equivalent to `\pset tableattr` *`table_options`*.

`\w` {*`filename`* | `|`*`command`*}

Outputs the current query buffer to the file *`filename`* or pipes it to the Unix command *`command`*.

`\x`

Toggles extended row format mode. As such it is equivalent to `\pset expanded`.

`\z` [*`pattern`*]

Produces a list of all tables in the database with their appropriate access permissions listed. If an argument is given it is taken as a regular expression which limits the listing to those tables which match it.

```
test=> \z
Access permissions for database "test"
 Relation |           Access permissions
```

```
-------+------------------------
 my_table | {"=r","joe=arwR", "group staff=ar"}
(1 row )
```

Read this as follows:

- `"=r"`: `PUBLIC` has read (**SELECT**) permission on the table.
- `"joe=arwR"`: User `joe` has read, write (**UPDATE**, **DELETE**), "append" (**INSERT**) permissions, and permission to create rules on the table.
- `"group staff=ar"`: Group `staff` has **SELECT** and **INSERT** permission.

The commands *GRANT* and *REVOKE* are used to set access permissions.

`\!` [*`command`*]

Escapes to a separate Unix shell or executes the Unix command *`command`*. The arguments are not further interpreted, the shell will see them as is.

`\?`

Get help information about the slash ("\") commands.

Command-line Options

If so configured, psql understands both standard Unix short options, and GNU-style long options. The latter are not available on all systems.

-a, –echo-all

Print all the lines to the screen as they are read. This is more useful for script processing rather than interactive mode. This is equivalent to setting the variable ECHO to `all`.

-A, –no-align

Switches to unaligned output mode. (The default output mode is otherwise aligned.)

-c, –command *`query`*

Specifies that psql is to execute one query string, *`query`*, and then exit. This is useful in shell scripts.

`query` must be either a query string that is completely parseable by the backend (i.e., it contains no psql specific features), or it is a single backslash command. Thus you cannot mix SQL and psql meta-commands. To achieve this you could pipe the string into psql, like so: `echo "\x \\ select * from foo;" | psql`.

-d, –dbname *`dbname`*

Specifies the name of the database to connect to. This is equivalent to specifying *`dbname`* as the first non-option argument on the command line.

-e, –echo-queries

Show all queries that are sent to the backend. This is equivalent to setting the variable ECHO to `queries`.

-E, –echo-hidden

Echos the actual queries generated by \d and other backslash commands. You can use this if you wish to include similar functionality into your own programs. This is equivalent to setting the variable ECHO_HIDDEN from within psql.

-f, –file *filename*

Use the file *filename* as the source of queries instead of reading queries interactively. After the file is processed, psql terminates. This in many ways equivalent to the internal command **\i**.

Using this option is subtly different from writing `psql < filename`. In general, both will do what you expect, but using `-f` enables some nice features such as error messages with line numbers. There is also a slight chance that using this option will reduce the startup overhead. On the other hand, the variant using the shell's input redirection is (in theory) guaranteed to yield exactly the same output that you would have gotten had you entered everything by hand.

-F, –field-separator *separator*

Use *separator* as the field separator. This is equivalent to **\pset fieldsep** or **\f**.

-h, –host *hostname*

Specifies the host name of the machine on which the postmaster is running. Without this option, communication is performed using local Unix domain sockets.

-H, –html

Turns on HTML tabular output. This is equivalent to `\pset format html` or the **\H** command.

-l, –list

Lists all available databases, then exits. Other non-connection options are ignored. This is similar to the internal command **\list**.

-o, –output *filename*

Put all query output into file *filename*. This is equivalent to the command **\o**.

-p, –port *`port`*

Specifies the TCP/IP port or, by omission, the local Unix domain socket file extension on which the postmaster is listening for connections. Defaults to the value of the PGPORT environment variable or, if not set, to the port specified at compile time, usually 5432.

-P, –pset *`assignment`*

Allows you to specify printing options in the style of **\pset** on the command line. Note that here you have to separate name and value with an equal sign instead of a space. Thus to set the output format to LaTeX, you could write `-P format=latex`.

-q

Specifies that psql should do its work quietly. By default, it prints welcome messages and various informational output. If this option is used, none of this happens. This is useful with the `-c` option. Within psql you can also set the QUIET variable to achieve the same effect.

-R, –record-separator *`separator`*

Use *`separator`* as the record separator. This is equivalent to the **\pset recordsep** command.

-s, –single-step

Run in single-step mode. That means the user is prompted before each query is sent to the backend, with the option to cancel execution as well. Use this to debug scripts.

-S, –single-line

Runs in single-line mode where a newline terminates a query, like a semicolon would do.

> **Nota:** This mode is provided for those who insist on it, but you are not necessarily encouraged to use it. In particular, if you mix SQL and

meta-commands on a line the order of execution might not always be clear to the unexperienced user.

-t, –tuples-only

Turn off printing of column names and result row count footers, etc. It is completely equivalent to the **\t**.

-T, –table-attr *`table_options`*

Allows you to specify options to be placed within the HTML table tag. See **\pset** for details.

-u

Makes psql prompt for the user name and password before connecting to the database.

This option is deprecated, as it is conceptually flawed. (Prompting for a non-default user name and prompting for a password because the backend requires it are really two different things.) You are encouraged to look at the `-U` and `-W` options instead.

-U, –username *`username`*

Connects to the database as the user *`username`* instead of the default. (You must have permission to do so, of course.)

-v, –variable, –set *`assignment`*

Performs a variable assignment, like the **\set** internal command. Note that you must separate name and value, if any, by an equal sign on the command line. To unset a variable, leave off the equal sign. These assignments are done during a

very early state of startup, so variables reserved for internal purposes might get overwritten again.

-V, –version

Shows the psql version.

-W, –password

Requests that psql should prompt for a password before connecting to a database. This will remain set for the entire session, even if you change the database connection with the meta-command **\connect**.

As of version 7.0, psql automatically issues a password prompt whenever the backend requests password authentication. Because this is currently based on a “hack”, the automatic recognition might mysteriously fail, hence this option to force a prompt. If no password prompt is issued and the backend requires password authentication the connection attempt will fail.

-x, –expanded

Turns on extended row format mode. This is equivalent to the command **\x**.

-?, –help

Shows help about psql command line arguments.

Advanced features

Variables

psql provides variable substitution features similar to common Unix command shells. This feature is new and not very sophisticated, yet, but there are plans to expand it in

the future. Variables are simply name/value pairs, where the value can be any string of any length. To set variables, use the psql meta-command **\set**:

```
testdb=> \set foo bar
```

sets the variable “foo” to the value “bar”. To retrieve the content of the variable, precede the name with a colon and use it as the argument of any slash command:

```
testdb=> \echo :foo
bar
```

> **Nota:** The arguments of **\set** are subject to the same substitution rules as with other commands. Thus you can construct interesting references such as `\set :foo 'something'` and get “soft links” or “variable variables” of Perl or PHP fame, respectively. Unfortunately (or fortunately?), there is not way to do anything useful with these constructs. On the other hand, `\set bar :foo` is a perfectly valid way to copy a variable.

If you call **\set** without a second argument, the variable is simply set, but has no value. To unset (or delete) a variable, use the command **\unset**.

psql’s internal variable names can consist of letters, numbers, and underscores in any order and any number of them. A number of regular variables are treated specially by psql. They indicate certain option settings that can be changed at runtime by altering the value of the variable or represent some state of the application. Although you can use these variables for any other purpose, this is not recommended, as the program behavior might grow really strange really quickly. By convention, all specially treated variables consist of all upper-case letters (and possibly numbers and underscores). To ensure maximum compatibility in the future, avoid such variables. A list of all specially treated variables follows.

DBNAME

The name of the database you are currently connected to. This is set everytime you connect to a database (including program startup), but can be unset.

ECHO

If set to "`all`", all lines entered or from a script are written to the standard output before they are parsed or executed. To specify this on program startup, use the switch `-a`. If set to "`queries`", psql merely prints all queries as they are sent to the backend. The option for this is `-e`.

ECHO_HIDDEN

When this variable is set and a backslash command queries the database, the query is first shown. This way you can study the PostgreSQL internals and provide similar functionality in your own programs. If you set the variable to the value "noexec", the queries are just shown but are not actually sent to the backend and executed.

ENCODING

The current client multibyte encoding. If you are not set up to use multibyte characters, this variable will always contain "SQL_ASCII".

HISTCONTROL

If this variable is set to `ignorespace`, lines which begin with a space are not entered into the history list. If set to a value of `ignoredups`, lines matching the previous history line are not entered. A value of `ignoreboth` combines the two options. If unset, or if set to any other value than those above, all lines read in interactive mode are saved on the history list.

Nota: This feature was shamelessly plagiarized from bash.

HISTSIZE

The number of commands to store in the command history. The default value is 500.

Nota: This feature was shamelessly plagiarized from bash.

HOST

The database server host you are currently connected to. This is set everytime you connect to a database (including program startup), but can be unset.

IGNOREEOF

If unset, sending an EOF character (usually Control-D) to an interactive session of psql will terminate the application. If set to a numeric value, that many EOF characters are ignored before the application terminates. If the variable is set but has no numeric value, the default is 10.

Nota: This feature was shamelessly plagiarized from bash.

LASTOID

The value of the last affected oid, as returned from an **INSERT** or **lo_insert** commmand. This variable is only guaranteed to be valid until after the result of the next SQL command has been displayed.

LO_TRANSACTION

If you use the PostgreSQL large object interface to specially store data that does not fit into one tuple, all the operations must be contained in a transaction block.

(See the documentation of the large object interface for more information.) Since psql has no way to keep track if you already have a transaction in progress when you call one of its internal commands **\lo_export**, **\lo_import**, **\lo_unlink** it must take some arbitrary action. This action could either be to roll back any transaction that might already be in progress, or to commit any such transaction, or to do nothing at all. In the latter case you must provide you own **BEGIN TRANSACTION**/**COMMIT** block or the results will be unpredictable (usually resulting in the desired action not being performed in any case).

To choose what you want to do you set this variable to one of “rollback”, “commit”, or “nothing”. The default is to roll back the transaction. If you just want to load one or a few objects this is fine. However, if you intend to transfer many large objects, it might be advisable to provide one explicit transaction block around all commands.

ON_ERROR_STOP

By default, if non-interactive scripts encounter an error, such as a malformed SQL query or internal meta-command, processing continues. This is has been the traditional behaviour of psql but it is sometimes not desirable. If this variable is set, script processing will immediately terminate. If the script was called from another script it will terminate in the same fashion. If the outermost script was not called from an interactive psql session but rather using the `-f` option, psql will return error code 3, to distinguish this case from fatal error conditions (error code 1).

PORT

The database server port you are currently connected to. This is set everytime you connect to a database (including program startup), but can be unset.

PROMPT1, PROMPT2, PROMPT3

These specify what the prompt psql issues is supposed to look like. See “*Prompting*” below.

QUIET

This variable is equivalent to the command line option `-q`. It is probably not too useful in interactive mode.

SINGLELINE

This variable is set be the command line options `-S`. You can unset or reset it at run time.

SINGLESTEP

This variable is equivalent to the command line option `-s`.

USER

The database user you are currently connected as. This is set everytime you connect to a database (including program startup), but can be unset.

SQL Interpolation

An additional useful feature of psql variables is that you can substitute ("interpolate") them into regular SQL statements. The syntax for this is again to prepend the variable name with a colon (`:`).

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

would then query the table `my_table`. The value of the variable is copied literally, so it can even contain unbalanced quotes or backslash commands. You must make sure that it makes sense where you put it. Variable interpolation will not be performed into quoted SQL entities.

A popular application of this facility is to refer to the last inserted OID in subsequent statement to build a foreign key scenario. Another possible use of this mechanism is to

copy the contents of a file into a field. First load the file into a variable and then proceed as above.

```
testdb=> \set content '\" `cat my_file.txt` '\"
testdb=> INSERT INTO my_table VALUES (:content);
```

One possible problem with this approach is that `my_file.txt` might contain single quotes. These need to be escaped so that they don't cause a syntax error when the third line is processed. This could be done with the program sed:

```
testdb=> \set content `sed -e "s/'/\\\\\\'/g" < my_file.txt`
```

Observe the correct number of backslashes (6)! You can resolve it this way: After psql has parsed this line, it passes `sed -e "s/'/\\\'/g" < my_file.txt` to the shell. The shell will do it's own thing inside the double quotes and execute `sed` with the arguments `-e` and `s/'/\\'/g`. When sed parses this it will replace the two backslashes with a single one and then do the substitution. Perhaps at one point you thought it was great that all Unix commands use the same escape character. And this is ignoring the fact that you might have to escape all backslashes as well because SQL text constants are also subject to certain interpretations. In that case you might be better off preparing the file externally.

Since colons may legally appear in queries, the following rule applies: If the variable is not set, the character sequence "colon+name" is not changed. In any case you can escape a colon with a backslash to protect it from interpretation. (The colon syntax for variables is standard SQL for embedded query languages, such as ecpg. The colon syntax for array slices and type casts are PostgreSQL extensions, hence the conflict.)

Prompting

The prompts psql issues can be customized to your preference. The three variables PROMPT1, PROMPT2, and PROMPT3 contain strings and special escape sequences that describe the appearance of the prompt. Prompt 1 is the normal prompt that is issued when psql requests a new query. Prompt 2 is issued when more input is expected during query input because the query was not terminated with a semicolon or a quote

was not closed. Prompt 3 is issued when you run an SQL **COPY** command and you are expected to type in the tuples on the terminal.

The value of the respective prompt variable is printed literally, except where a percent sign (“%”) is encountered. Depending on the next character, certain other text is substituted instead. Defined substitutions are:

`%M`

The hostname of the database server (or “.” if Unix domain socket).

`%m`

The hostname of the database server truncated after the first dot.

`%>`

The port number at which the database server is listening.

`%n`

The username you are connected as (not your local system user name).

`%/`

The name of the current database.

`%~`

Like `%/`, but the output is “~” (tilde) if the database is your default database.

`%#`

If the current user is a database superuser, then a “#”, otherwise a “>”.

`%R`

In prompt 1 normally “=”, but “^” if in single-line mode, and “!” if the session is disconnected from the database (which can happen if **\connect** fails). In prompt 2 the sequence is replaced by “-”, “*”, a single quote, or a double quote, depending on whether psql expects more input because the query wasn’t terminated yet,

because you are inside a `/* ... */` comment, or because you are inside a quote. In prompt 3 the sequence doesn't resolve to anything.

`%`*`digits`*

If *`digits`* starts with `0x` the rest of the characters are interpreted at a hexadecimal digit and the character with the corresponding code is subsituted. If the first digit is `0` the characters are interpreted as on octal number and the corresponding character is substituted. Otherwise a decimal number is assumed.

`%:`*`name`*`:`

The value of the psql, variable *`name`*. See the section "*Variables*" for details.

`%‘`*`command`*`‘`

The output of *`command`*, similar to ordinary "back-tick" substitution.

To insert a percent sign into your prompt, write `%%`. The default prompts are equivalent to `'%/%R%# '` for prompts 1 and 2, and `'» '` for prompt 3.

Nota: This feature was shamelessly plagiarized from tcsh.

Miscellaneous

psql returns 0 to the shell if it finished normally, 1 if a fatal error of its own (out of memory, file not found) occurs, 2 if the connection to the backend went bad and the session is not interactive, and 3 if an error occurred in a script and the variable ON_ERROR_STOP was set.

Before starting up in interactive mode, psql attempts to read and execute commands from the file `$HOME/.psqlrc`. It could be used to set up the client or the server to taste (using the **\set** and **SET** commands).

GNU readline

psql supports the readline and history libraries for convenient line editing and retrieval. The command history is stored in a file named `.psql_history` in your home directory and is reloaded when psql starts up. Tab-completion is also supported, although the completion logic makes no claim to be an SQL parser. When available, psql is automatically built to use these features. If for some reason you do not like the tab completion, you can turn if off by putting this in a file named `.inputrc` in your home directory:

```
$if psql
set disable-completion on
$endif
```

(This is not a psql but a readline feature. Read its documentation for further details.)

If you have the readline library installed but psql does not seem to use it, you must make sure that PostgreSQL's top-level `configure` script finds it. `configure` needs to find both the library `libreadline.a` (or `libreadline.so` on systems with shared libraries) *and* the header files `readline.h` and `history.h` (or `readline/readline.h` and `readline/history.h`) in appropriate directories. If you have the library and header files installed in an obscure place you must tell `configure` about them, for example:

```
$ ./configure -with-includes=/opt/gnu/include -with-libs=/opt/gnu/lib  ...
```

Then you have to recompile psql (not necessarily the entire code tree).

The GNU readline library can be obtained from the GNU project's FTP server at ftp://ftp.gnu.org.

Examples

> **Nota:** This section only shows a few examples specific to psql. If you want to

> learn SQL or get familiar with PostgreSQL, you might wish to read the Tutorial that is included in the distribution.

The first example shows how to spread a query over several lines of input. Notice the changing prompt.

```
testdb=> CREATE TABLE my_table (
testdb->  first integer not null default 0,
testdb->  second text
testdb-> );
CREATE
```

Now look at the table definition again:

```
testdb=> \d my_table
             Table "my_table"
 Attribute |  Type   |      Modifier
-------+------+-------------
 first     | integer | not null default 0
 second    | text    |
```

At this point you decide to change the prompt to something more interesting:

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

Let's assume you have filled the table with data and want to take a look at it:

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-----+-----
     1 | one
     2 | two
     3 | three
     4 | four
(4 rows)
```

Notice how the int4 colums in right aligned while the text column in left aligned. You can make this table look differently by using the **\pset** command.

```
peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM my_table;
+-----+-----+
| first | second |
+-----+-----+
|     1 | one    |
|     2 | two    |
|     3 | three  |
|     4 | four   |
+-----+-----+
(4 rows)

peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM my_table;
first second
--- ----
    1 one
    2 two
    3 three
    4 four
(4 rows)

peter@localhost testdb=> \pset border 1
Border style is 1.
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep ","
Field separator is ",".
peter@localhost testdb=> \pset tuples_only
Showing only tuples.
peter@localhost testdb=> SELECT second, first FROM my_table;
one,1
```

```
two,2
three,3
four,4
```

Alternatively, use the short commands:

```
peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

Appendix

Bugs and Issues

- In some earlier life psql allowed the first argument to start directly after the (single-letter) command. For compatibility this is still supported to some extent but I

am not going to explain the details here as this use is discouraged. But if you get strange messages, keep this in mind. For example

```
testdb=> \foo
Field separator is "oo".
```

is perhaps not what one would expect.

- psql only works smootly with servers of the same version. That does not mean other combinations will fail outright, but subtle and not-so-subtle problems might come up.

pgtclsh

Nombre

`pgtclsh` — Postgres Cliente para shell TCL

Synopsis

```
pgtclsh [ base_de_datos ]
```

Entrada

`base_de_datos`

El nombre de una base de datos existente a la que se quiera tener acceso.

Salida

Descripción

pgtclsh proporciona un interfaz al shell TCL para Postgres.

Otra manera de tener acceso a Postgres por medio de tcl es usando *pgtksh* o *pgaccess*.

pgtksh

Nombre

`pgtksh` — Postgres Shell gráfico para TCL/TK

Synopsis

```
pgtksh [ base_de_datos ]
```

Entrada

`base_de_datos`

El nombre de una base datos existente a la que se quiera tener acceso.

Salida

Descripción

pgtksh proporciona un interfaz gráfico al shell TCL/TK para Postgres.

Otra manera de tener acceso a Postgres por medio de TCL es usando *pgtclsh* o *pgaccess*.

vacuumdb

Nombre

`vacuumdb` — Limpia y analiza una base de datos PostgreSQL

Synopsis

```
vacuumdb [ opciones de conexión ] [ -analyze | -z ] [ -alldb | -a ] [ -
verbose | -v ]
         [ -table 'tabla [ ( columna [,...] ) ]' ] [ [-d] nombre_bd ]
```

Entradas

vacuumdb acepta los siguientes argumentos en la línea de comandos:

[-d, –dbname] *nombre_bd*

Especifica el nombre de la base de datos que de be ser limpiada o analizada.

-z, –analyze

Calcula estadísticas sobre la base de datos para ser usadas por el optimizador.

-a, –alldb

Limpia todas las bases de datos.

-v, –verbose

Imprime información detallada durante el proceso.

-t, –table *tabla* [(*columna* [,...])]

Limpia o analiza únicamente la *tabla* indicada. Se pueden especificar nombres de columnas únicamente cuando se usa la opción `-analyze`.

> **Sugerencia:** Si usted da el nombre de columnas que deben ser analizadas, probablemente tendrá que usar caracteres de escape de la shell para los paréntesis.

vacuumdb también acepta los siguientes argumentos de línea de comandos, para parámetros de conexión:

-h, –host *anfitrión*

Especifica el nombre de la máquina anfitriona en la cual se está ejecutando el postmaster.

-p, –port *puerta*

Especifica la puerta de Internet TCP/IP o el fichero Unix de extensión de dominio local de conexión en que el postmaster recibe conexiones.

-U, –username *nombre*

Nombre de usuario que se debe usar para conectar.

-W, –password

Obliga el pedido de contraseña antes de ejecutar.

-e, –echo

Escribe una copia de los comandos que vacuumdb genera y envía al servidor.

-q, –quiet

No muestre la respuesta.

Mensajes de Resultados

`VACUUM`

Todo corrió bien.

`vacuumdb: La limpieza falló.`

Algo ha fallado. vacuumdb es apenas un guión de interfaz. Consulte *VACUUM* y *psql* para un discusión detallada de los mensajes de error y posibles problemas.

Descripción

vacuumdb es un utilitario para limpiar una base de datos PostgreSQL. vacuumdb también produce estadísticas internas usadas por el optimizador de búsquedas de Postgres.

vacuumdb es un guión que envuelve al comando *VACUUM* de PostgreSQL, por medio del terminal interactivo*psql*. No existe diferencia efectiva entre la limpieza de bases de datos usando este u otros métodos. El guión deberá lograr encontrar a psql y deberá existir un servidor de bases de datos en ejecución en el anfitrión usado. Serán usadas cualquier configuración y variables de estado de psql y de la librería de interfaz libpq.

Uso

Para limpiar la base de datos `prueba`:

```
$ vacuumdb prueba
```

Para analizar para el optimizador una base de datos llamada `bdgrande`:

```
$ vacuumdb -analyze bdgrande
```

Para analizar para el optimizador una única columna `cual` en la tabla `tal` de una base de datos llamada `xyzzy`:

```
$ vacuumdb -analyze -verbose -table 'tal(cual)' xyzzy
```

Capítulo 16. Aplicaciones del sistema

Esta es la información de referencia para los servidores y utilidades de soporte de Postgres.

initdb

Nombre

`initdb` — Crea una nueva instalación de la base de datos de PostgreSQL

Synopsis

```
initdb [ -pgdata|-D dbdir ]
       [ -sysid|-i sysid ]
       [ -pwprompt|-W ]
       [ -encoding|-E encoding ]
       [ -pglib|-L libdir ]
       [ -noclean | -n ] [ -debug | -d ] [ -template | -t ]
```

Inputs

–pgdata=*dbdir*
-D *dbdir*
PGDATA

Esta opción especifica en que parte del sistema de archivos será almacenada la base de datos. Ésta es la única información requerida por el initdb, pero podemos omitirla estableciendo la variable de entorno PGDATA lo que puede ser conveniente ya que el servidor de la base de datos (`postmaster`) puede encontrar el directorio de la base de datos más adelante a través de la misma variable.

–sysid=*sysid*
-i *sysid*

Selecciona él id del sistema para el super usuario (root) de la base de datos. Por omisión apunta al id de aquel usuario que este ejecutando initdb. Realmente no es importante cuál sea él id del sistema para el super usuario, ya que uno podría elegir comenzar la numeración con cualquier número como 0 o 1.

–pwprompt
-W

Ocasiona que el initdb pregunte por el password del super usuario (root) de la base de datos. Si uno no planea usar la autentificación a través de passwords, entonces realmente no es importante. De cualquier manera uno no podrá utilizar la autentificación a través de passwords hasta que haya establecido un password.

–encoding=*encoding*
-E *encoding*

Selecciona la codificación multibyte para la base de datos modelo (o plantilla). De hecho esta también será la codificación por defecto para cualquier base de datos que uno cree más adelante, a menos que usted la cambie. Para utilizar la característica de codificación multibyte, se debe especificar durante el tiempo de

construcción (creación de la BD), en cuyo caso uno también selecciona el valor por defecto para esta opción.

Otros parámetros utilizados menos comúnmente están también disponibles:

–pglib=*`libdir`*
-l *`libdir`*

initdb necesita algunos archivos de entrada para poder inicializar la base de datos. Esta opción indica dónde encontrarlos. Normalmente, uno no tiene que preocuparse por esto puesto que el initdb conoce los esquemas de instalación más comunes y encontrará, normalmente, los archivos por sí mismo. Se le dirá si usted necesita especificar su ubicación explícitamente. Si sucede esto, uno de los ficheros se llama `global1.bki.source` y está instalado tradicionalmente junto con los otros archivos en el directorio de bibliotecas (por ejemplo, /usr/local/pgsql/lib) `/usr/local/pgsql/lib`).

–template
-t

Replace the `template1` Substituye la base de datos template1 en un sistema de base de datos existente, y no toca otra cosa. Esto es útil cuando se necesita actualizar el `template1`de la base de datos usando el initdb de una versión más nueva de PostgreSQL, o cuando el `template1`de la base de datos se ha corrompido por algún problema del sistema. Normalmente el contenido del `template1` se mantendrá constante a través de la vida del sistema de base de datos. No se puede destruir cualquier otra cosa ejecutando el initdb con la opción `-template`.

–noclean
-n

Por defecto, cuando initdb determina que un error evita que se cree totalmente el sistema de base de datos, remueve cualquier archivo que pudo haber creado, antes

de determinar que no puede acabar el trabajo. Esta opción inhibe "tidying-up" y es por lo tanto, útil para depurar.

–debug
-d

Imprime la salida de depuración de la "carga inicial backend" y algunos otros mensajes de poco interés para el público en general. La "carga inicial backend" es la aplicación que el initdb usa para crear las tablas del catálogo. Esta opción genera una enorme cantidad de salida.

Salidas

initdb creará los ficheros en el área de datos especificada que son las tablas del sistema y el marco de trabajo para una instalación completa.

Descripción

initdb crea un nuevo sistema de base de datos de PostgreSQL database system. Un sistema de base de datos es una colección de bases de datos que son todas administradas por el mismo usuario de UNIX y manejadas por un solo postmaster

Crear un sistema de base de datos consiste en crear los directorios en los cuales los datos de la base de datos serán almacenados. generar las tablas de catálogo compartidas (son tablas que no pertenecen a ninguna base de datos determinada). crear el `template1` de la base de datos. Cuando usted crea una nueva base de datos, todo el `template1`de la base de datos se copia. Contiene las tablas de catálogo llenas para cosas como los tipos interconstruidos

No se debe ejecutar el initdb como root. Esto se debe ya que uno no puede ejecutar el servidor de la base de datos ni siquiera como root, pero el servidor necesita tener acceso a los archivos que initdbcrea. Además, durante la fase de la inicialización,

cuando no hay usuarios y ningún control de acceso instalado, postgres solamente se conectará con el nombre de usuario actual de UNIX, así que uno debe iniciar una sesión bajo la cuenta que poseerá el proceso del servidor.

Aunque initdb procurará crear el directorio de datos respectivo, lo cierto es que no tendrá el permiso para hacerlo. Por lo tanto, es una buena idea crear el directorio de datos antes de ejecutar initdb *y* entregar la propiedad de él al super usuario de la base de datos.

initlocation

Nombre

`initlocation` — Crea un área de almacenamiento secundario para la base de datos de PostgreSQL

Synopsis

`initlocation` *`directory`*

Entradas

`directory`

¿Dónde deseas almacenar las bases de datos alternativas dentro del sistema de archivos UNIX?

Salidas

initlocation creará directorios en el lugar especificado .

Descripción

initlocation crea una nueva área de almacenamiento secundario para la base de datos PostgreSQL Vea la discusión en *CREATE DATABASE* sobre cómo manejar y utilizar áreas de almacenamiento secundario. Si el argumento no contiene un slash (/) y es inválido como vía (path), entonces se asume que es una variable de entorno (ambiente), la cual es referenciada. Vea los ejemplos al final.

Para poder utilizar este comando usted debe haber entrado (con “su”, por ejemplo) a la base de datos como super usuario (root).

Uso

Para crear una base de datos en una posición alterna, usando una variable de entorno:

```
$ export PGDATA2=/opt/postgres/data
$ initlocation PGDATA2
$ createdb ’testdb’ -D ’PGDATA2/testdb’
```

Alternativamente, si usted permite vías (paths) absolutas entonces podría escribir:

```
$ initlocation /opt/postgres/data
$ createdb ’testdb’ -D ’/opt/postgres/data/testdb’
```

ipcclean

Nombre

`ipcclean` — Limpia la memoria compartida y los semáforos de "backends" abortados.

Synopsis

```
ipcclean
```

Entradas

Ninguna.

Salidas

Ninguna.

Descripción

ipcclean limpia la memoria compartida y el espacio de semáforos de "backends" abortados, borrando todas las instancias que son propiedad del usuario `postgres`. Solamente el DBA (DataBase Administrator - Administrador de la Base de Datos) debe ejecutar este programa ya que puede causar algún tipo de comportamiento extraño (es decir, caídas) si se ejecuta durante una ejecución multiusuario. Este programa se debe ejecutar si aparecen mensajes como por ejemplo `semget: No queda espacio`

`libre en el dispositivo` al ejecutar el proceso postmaster o el servidor "backend".

Si se ejecuta esta órden mientras el proceso postmaster está corriendo, se eliminará la memoria compartida y los semáforos almacenados por el postmaster. Esto puede provocar el fallo general de los servidores "backend" iniciados por ese postmaster.

Este script es un "hack", pero en los muchos años desde que fué escrito, nadie ha venido con una solución igualmente eficaz y portable. Cualquier sugerencia será bienvenida.

El script hace una suposición sobre el formato de salida de la utilidad ipcs, suposición que puede no ser cierta en todos los sistemas operativos, por lo que puede fallar en su SO particular.

pg_passwd

Nombre

`pg_passwd` — manipula el fichero plano de passwords.

Synopsis

`pg_passwd` *`filename`*

Descripción

pg_passwd es una herramienta para manipular la funcionalidad del fichero plano de passwords de Postgres. Este estilo de autentificación de passwords no se *requiere* en

una instalación, pero es uno de los diversos mecanismos utilizados en la seguridad.

Especifique el archivo de passwords en el mismo estilo que autentificación `Ident` en: `$PGDATA/pg_hba.conf`:

```
host  unv     133.65.96.250   255.255.255.255 password passwd
```

Donde la línea anterior permite el acceso desde 133,65,96,250 usando los passwords listados en `$PGDATA/passwd`. El formato del archivo de passwords sigue el formato de `/etc/passwd` y `/etc/shadow`. El primer campo es el nombre de usuario, y el segundo campo es el password cifrado. El resto es totalmente ignorado. Así las tres líneas siguientes de ejemplo especifican el mismo par de nombre de usuario y password:

```
pg_guest:/nB7.w5Auq.BY:10031::::::
pg_guest:/nB7.w5Auq.BY:93001:930::/home/guest:/bin/tcsh
pg_guest:/nB7.w5Auq.BY:93001
```

Provea del fichero de passwords al comando pg_passwd. En el caso descrito anteriormente, después de cambiar el directorio de trabajo a PGDATA, la ejecución siguiente del comando especifica el nuevo password para `pg_guest`:

```
% pg_passwd passwd
Username: pg_guest
Password:
Re-enter password:
```

Donde la petición `Password:` y `Re-enter password:` requieren el mismo password de entrada pero no se visualizarán en la terminal. El archivo original de passwords se renombra como `passwd.bk`.

psql utiliza la opción `-u` para invocar este estilo de autentificación.

Las líneas siguientes muestran ejemplos de uso de la opción:

```
% psql -h hyalos -u unv
Username: pg_guest
Password:
Bienvenido al monitor interactivo de PostgreSQL:
  Lea por favor el archivo COPYRIGHT para los términos de derechos de au-
tor del tipo de PostgreSQL.
   Escriba \? para la ayuda en comandos slash (/)
   Escriba \q para salir
   Escriba \g o terminar con punto y coma para ejecutar la consulta
Usted está conectado actualmente con la base de datos: unv
unv =>
```

La autentificación de Perl5 utiliza el nuevo estilo de `Pg.pm` como esto:

```
$conn = Pg::connectdb("host=hyalos dbname=unv
                       user=pg_guest password=xxxxxxx");
```

Para más detalles, refiérase a `src/interfaces/perl5/Pg.pm`.

La autentificación Pg{tcl, tk}sh utiliza el comando `pg_connect` con la opción `-conninfo` por lo tanto:

```
% set conn [pg_connect -conninfo \\
            "host=hyalos dbname=unv \\
             user=pg_guest password=xxxxxxx "]
```

Se pueden enumerar todas las claves para la opción ejecutando el comando siguiente:

```
% puts [ pg_conndefaults]
```

pg_upgrade

Nombre

`pg_upgrade` — permite la actualización de una versión anterior sin tener que volver a recargar los datos.

Synopsis

```
pg_upgrade [ -f filename ] old_data_dir
```

Descripción

pg_upgrade es una utilidad para actualizar una versión anterior de PostgreSQL sin la necesidad de recargar todos los datos. No todas las transiciones de versiones de Postgres se pueden manejar de esta manera. Verifique las notas de la versión para saber si hay detalles en su instalación.

Actualización de Postgres with pg_upgrade

1. Respalde su directorio de datos existente, preferiblemente haciendo un vaciado completo con el pg_dumpall.
2. Luego realice:

   ```
   % pg_dumpall -s >db.out
   ```

para vaciar la antigua tabla de definiciones de la base de datos sin ningún dato.

3. Detenga el antiguo postmaster y todos los "backends".
4. Renombre (usando mv) su antiguo directorio pgsql `data/` a `data.old/`.
5. Ejecute

   ```
   % make install
   ```

   para instalar los nuevos binarios.
6. Ejecute initdb para crear una nueva base de datos template1 que contenga las tablas del sistema para la nueva versión.
7. Inicie el nuevo postmaster. (Nota: es de suma importancia que ningún usuario se conecte a la base de datos hasta que la actualización esté completada. Quizás desee iniciar el postmaster sin la opción - i y/o alterar `pg_hba.conf` temporalmente.)
8. Cambie su directorio de trabajo hacia el directorio principal del pgsql, y ejecute:

   ```
   % pg_upgrade -f db.out data.old
   ```

   El programa hará algunas verificaciones para cerciorarse de que todo esta configurado correctamente, y ejecutará el script db.out para volver a reconstruir todas las bases de datos y tablas que uno tenía, pero sin datos. Entonces moverá físicamente los archivos de datos que no contienen tablas del sistema y los índices desde `data.old/` hacia los subdirectorios indicados debajo de `data.old/` sustituyendo los archivos de datos vacíos creados durante la ejecución del script db.out.
9. Restablezca si es necesario su antiguo archivo `pg_hba.conf` para permitir conexiones a los usuarios.
10. Detenga y vuelva a iniciar el postmaster.
11. Examine *cuidadosamente* el contenido de la base de datos actualizada. Si encuentra algún problema, entonces necesitará recuperar sus datos restableciendo su respaldo completo pg_dump. Puede eliminar el directorio `data.old/` cuando se encuentre satisfecho con los resultados obtenidos.

12. La base de datos actualizada se encontrará en un estado no limpio. Probablemente deseará ejecutar un **VACUUM ANALYZE** antes de que comience el trabajo de producción.

postgres

Nombre

`postgres` — Ejecuta un proceso Postgres de usuario único

Synopsis

```
postgres [ dbname ]
postgres [ -B nBuffers ] [ -C ] [ -D DataDir ] [ -E ] [ -F ]
   [ -O ] [ -Q ] [ -S SortSize ] [ -d [ DebugLevel ] ] [ -e ]
   [ -o ] [ OutputFile ] [ -s ] [ -v protocol ] [ dbname ]
```

Entradas

postgres acepta los siguientes argumentos en la línea de comandos:

dbname

El argumento opcional *dbname* especifica el nombre de la base de datos a acceder. *dbname* toma por defecto el valor de la variable de entorno USER.

-B *nBuffers*

Si el motor de datos se está ejecuntando bajo el postmaster, *nBuffers* es el número de búfers de memoria compartida que el postmaster tiene reservados para los procesos servidores que arranca. Si el motor de datos se ejecuta como un proceso independiente, especifica el número de búfers a reservar. Este valor es por defecto de 64 búfers, de 8k cada uno (o el valor que BLCKSZ tenga asignado en config.h)

-C

No mostrar el número de versión del servidor.

-D *DataDir*

Especifica el directorio a usar como raíz para el árbol de directorios de las bases de datos. Si -D no se especifica, el nombre del directorio de datos por defecto es el valor de la variable de entorno PGDATA. Si PGDATA no tiene un valor asignado, entonces el directorio usado es `$POSTGRESHOME/data`. Si ni la variable de entorno ni esta opción de línea de comandos están asignadas, se usa el directorio por defecto indicado durante la compilación.

-E

Muestra todas las consultas.

-F

Desactiva la ejecución automática de `fsync()` después de cada transacción. Esta opción mejora el rendimiento, pero una caída del sistema durante una transacción en curso puede provocar la pérdida de los últimos datos introducidos. Sin la llamada a `fsync()` los datos son almacenados temporalmente por el sistema operativo, y escritos en disco más tarde.

-O

Ignorar las restricciones que impiden la modificación de la estructura de las tablas de sistema. Estas tablas son tipicamente las que incluyen «pg_» al principio del nombre.

-Q

Especifica el modo «silencioso».

-S *SortSize*

Especifica la cantidad de memoria a usar por ordenaciones internas y hashes antes de reordenar en ficheros temporales en disco. El valor se indica en kilobytes, y su valor por defecto es de 512 kilobytes. Nótese que para una consulta compleja, varias ordenaciones y/o hashes deben ejecutarse en paralelo, y cada una puede utilizar hasta *SortSize* kilobytes antes de empezar a poner datos en ficheros temporales.

-d [*DebugLevel*]

El argumento opcional *DebugLevel* determina el volumen de información de depuración que el servidor producirá. Si *DebugLevel* es uno, el postmaster registrará todo el trafico de conexión y nada más. Para valores 2 y mayores, la depuración es activada en el proceso del motor de datos y el postmaster muestra más información, incluyendo su entorno y tráfico de proceso. Nótese que si no se especifica un archivo para almacenar la información de depuración la misma aparecerá en la consola del proceso padre postmaster.

-e

Esta opción controla cómo son interpretadas las fechas en la entrada y salida de la base de datos. Si la opción `-e` es incluida, las fechas pasadas a y desde los procesos de aplicación asumirán el formato «Europeo» (`DD-MM-YYYY`), En caso contrario se asume que las fechas están en formato «Americano» (`MM-DD-YYYY`). Las fechas se aceptan por el motor de datos en una amplia variedad de formatos, y para la entrada de fechas, este parámetro afecta a la interpretación de casos ambiguos. Véase *Data Types* para más información.

-o *OutputFile*

Envía los mensajes de error y la información de depuración a *OutputFile*. Si el motor de datos se ejecuta bajo el postmaster, los mensajes de error se envían también a la aplicación además de a *OutputFile*, pero la información de

depuración se envía a la consola del postmaster (puesto que sólo un descriptor de archivo puede enviarse a un fichero real).

-s

Muestra información de tiempo y otras estadísticas al final de cada consulta. Esto es útil para medidas de rendimiento o para su uso en el ajuste del número de búfers.

-v *protocol*

Especifica el número del protocolo a emplear entre la aplicación y el motor de datos en esta sesión en particular.

Hay otras varias opciones que pueden especificarse, usadas principalmente con propósitos de depuración. Se listan aquí únicamente para su uso por desarrolladores de Postgres. *Se desaconseja claramente el uso de cualquiera de estas opciones.* Además, cualquiera de estas opciones puede desaparecer o cambiar en cualquier momento.

Estas opciones para casos especiales son:

-A n|r|b|Q\fIn\fP|X\fIn\fP

Esta opción genera un tremendo volumen de información.

-L

Desactiva el sistema de bloqueos.

-N

Desactiva el uso del carácter de nueva línea como un delimitador de consultas.

-f [s | i | m | n | h]

Prohibe el uso de métodos particulares de escaneo y reuniones: `s` y `i` desactivan scaneos secuenciales y de índices, respectivamente, mientras `n`, `m`, y `h` lo hacen

con reuniones de bucles enlazados, merge y hash.

Nota: Ni los scaneos secuenciales ni las uniones de bucles enlazados pueden ser desactivados completamente; Neither sequential scans nor nested-loop joins can be disabled completely; las opciones `-fs` y `-fn` simplemente impiden al optimizador usar estos tipos de planes si hay cualquier otra alternativa.

-i

Previene la ejecución de la consulta. En su lugar muestra el árbol del plan de ejecución.

-p *dbname*

Indica al motor de datos que ha sido iniciado por un postmaster y hace diferentes suposiciones sobre la gestión de los búfers, descriptores de ficheros, etc. Los parámetros posteriores a -p están restringidos a los considerados «seguros».

-t pa[rser] | pl[anner] | e[xecutor]

Muestra estadísticas de tiempo para cada consulta relacionadas con cada uno de los módulos principales del sistema. No puede ser usada con `-s`.

Salidas

Del infinito número de mensajes de error que pueden verse cuando se ejecuta directamente el motor de datos, probablemente los más comunes son:

`semget: No space left on device`

Si se muestra este mensaje, debería ejecutarse ipcclean Una vez hecho esto pruébese a iniciar postmaster de nuevo. Si todavía no funciona, probablemente es

necesario configurar el núcleo para emplear memoria compartida y semáforos, como se describe en las notas de instalación. Si se cuenta con un núcleo con memoria compartida particularmente pequeña y/o límites al uso de semáforos, será necesario reconfigurarlo para inrementar uno o los otros.

Sugerencia: Es posible posponer la reconfiguración del núcleo reduciendo -B para minimizar el uso de memoria compartida por Postgres.

Descripción

El motor de datos de Postgres puede ser ejecutado directamente desde un shell de usuario. Esto debería hacerse únicamente para tareas de depuración por el DBA, y no mientras otros motores de datos están siendo gestionados por un postmaster en este conjunto de bases de datos.

Algunos de los parámetros descritos aquí pueden pasarse al motor de datos a través del campo «opciones de base de datos» de una petición de conexión, y por lo tanto pueden referirse a un motor particular sin necesidad de reiniciar el postmaster. Esto es particularmente práctico para parámetros relacionados con la depuración.

El argumento opcional *dbname* especifica el nombre de la base de datos a acceder. *dbname* toma por defecto el valor de la variable de entorno USER.

Notes

Existen utilidades para resolver problemas de memoria compartida como ipcs(1), ipcrm(1), e ipcclean(1). Ver también *postmaster*.

postmaster

Nombre

`postmaster` — Ejecuta el servidor (backend)multiusuario de Postgres

Synopsis

```
postmaster [ -B nBuffers ] [ -D
DataDir ] [ -i ] [ -l ]
postmaster [ -B nBuffers ] [ -D
DataDir ] [ -N nBackends ] [ -S ]
   [ -d [ DebugLevel ] [ -i ] [
-l ] [ -o BackendOptions ] [ -p
port ]
postmaster [ -n | -s ] ...
```

Inputs

postmaster acepta los siguientes parámetros en su línea de comandos:

-B *nBuffers*

Indica el número de buffers de memoria compartida que postmasterasignar y administrará para los procesos del servidor que inice. El valor predeterminado para esta opción es 64 buffers, siendo cada buffer de 8 kilobytes (o lo que sea que esté indicado en BLCKSZ en config.h).

-D *DataDir*

Especifica el directorio a usar como raíz del arbol de directorios de bases de datos. Si no se especifica -D, el nombre de directorio predeterminado es el valor

de la variable de entorno PGDATA. Si PGDATA no está especificada, entonces se utiliza el directorio `$POSTGRESHOME/data`. Si no se especifica ni la variable de entorno ni esta opció de línea de comando, el directorio predeterminado es el utilizado al momento de la compilación.

-N *`nBackends`*

El máximo número de procesos en el servidor (backend) que postmaster tiene permitido iniciar. En la configuración predeterminada este valor está usualmente definido en 32, y puede ser fijado hasta un valor máximo de 1024 si su sistema puede soportar esa cantidad de procesos. Tanto el valor predeterminado como el máximo puede modificarse cuando se compila Postgres (vea el archivo src/include/config.h)

-S

Indica que el proceso de postmaster debe iniciarse en modo silencioso. Esto es, anulará la vinculación con la terminal del usuario (que tiene el control) e iniciará su propio grupo de proceso. Esta opción no debería utilizarse en conjunto con las opciones de depuración ya que cualquier mensaje enviado a la salida estándar y a la salida de error estándar serán descartados.

-d [*`DebugLevel`*]

Este argumento *`DebugLevel`* determina la cantidad de información de depuración que producirá el servidor. Si *`DebugLevel`* es uno, postmaster rastreará todo el tráfico de conexión y nada más. Para niveles iguales o mayores a 2 se activa la depuración y el proceso del servidor y postmaster muestran más información, incluyendo el entorno del servidor y tráfico de proceso. Note que si no se especifica ningún archivo para que los servidores del backend envíen su información, esta información será exhibida en la terminal de su proceso postmaster padre.

-i

Esta opción habilita las comunicaciones mediante TCP/IP o mediante el socket de dominio Internet. Sin esta opción solamente es posible la comunicación a través

del socket de dominio Unix local.

-l

Este parámetro habilita la comunicación mediante el socket SSL. También es necesario especificar la opción `-i`. Además, debió habilitarse SSL en el momento de la compilación.

-o *`BackendOptions`*

Las opciones de `postgres` que se especifican en *`BackendOptions`* son pasadas a todos los procesos iniciados en el servidor por este postmaster. are passed to all backend server processes started by this postmaster. Si la cadena de opciones contiene espacios, entonces debe encerrársela entre comillas.

-p *`port`*

Especifica el puerto TCP/IP o la extensión de archivo del socket del dominio Unix local en el cual postmaster deberá esperar por conexiones solicitadas desde las aplicaciones del lado del cliente. El valor predeterminado es el especificado en la variable de entorno PGPORT o, si PGPORT no fue especificada, se toma como predeterminado el valor establecido cuando Postgres fue compilado (normalmente 5342). Si se especifica un puerto distinto del predeterminado, a todas las aplicaciones cliente (incluyendo psql) deberá especificárseles el mismo puerto ya sea mediante las opciones de línea de comando o utilizando la variable de entorno PGPORT.

Existen algunas opciones de línea de comandos disponibles para realizar depuraciones en caso de que un proceso en el servidor termine de forma anormal. Estas opciones controlan el comportamiento de postmaster en estas situaciones, y *ninguna de ellas está pensada para ser utilizada en situaciones normales*.

La estrategia usual para esta situación es notigicar a todos los demás procesos en el servidor que deben terminar y reinicializar la memoria y semáforos compartidos. Esto es así debido a que un proceso de servidor que funcione de manera errática podría corromper alguno de estos recursos compartidos antes de terminar.

Estas opciones especiales son:

-n

postmaster no reinicializaráa las estructuras compartidas. Un programador podría luego analizarlas con el programa shmemdoc y examinar la memoria compartida y lo estados de los semáforos.

-s

postmaster detendrá todos los demás procesos del servidor enviándoles la señal `SIGSTOP`, pero no hará que terminene. Estos permite a los programadores del sistema realizar vuelcos de núcleo a mano para todos los procesos del servidor.

Salidas

`semget: No space left on device`

Si aparece este mensaje, debería ejecutar el comando ipcclean. Una vez hecho esto, pruebe iniciar postmaster nuevamente. Si aun no funciona, probablemente necesite configurar el núcleo (kernel) de su sistema para que pueda utilizar memoria compartida y semáforos, tal como se describe en las notas de instalación. Si ejecuta múltiples instancias de postmaster en un sólo host, o tiene un kernel con muy poca memoria compartida o un límite de semáforos muy pequeño, tal vez deba reconfigurarlo su kernel para incrementar sus parámetros de memoria comapartida y semáforos.

Sugerencia: Tal vez pueda posponer la reconfiguración del kernel disminuyendo lo especificado con -B para reducir la utilización de memoria compartida por parte de Postgres, o disinuyendo lo especificado con -N para reducir la cantidad de semáforos que utiliza Postgres.

`StreamServerPort: cannot bind to port`

Si se encuentra con este mensaje, debe asegurarse de que no existen otros procesos de postmaster ejecutándose en el momento. La manera más fácil de determinar esto es mediante el comando

```
% ps -ax | grep postmaster
```

en sistemas basados en BSD, o

```
% ps -e | grep postmast
```

en sistemas tipo System V o compatibles con POSIX como ser HP-UX.

Si está seguro de que no existen otros procesos de postmaster en ejecución, y aun así sigue recibiendo este error, intente especificar un puerto diferente utilizando la opción `-p`. También puede obtener este mensaje de error si finaliza postmaster y lo vuelve a iniciar inmediatamente utilizando el mismo número de puerto; simplemente espere unos segundos hasta que el sistema operativo cierre el puerto antes de intentar nuevamente. Finalmente, puede que obtenga este mensaje de error si especifica un número de puerto que su sistema operativo considere reservado. Por ejemplo, muchas versiones de Unix consideran que los puertos con número menor a 1024 deben ser confiables y solo permite al superusuario tner acceso a ellos.

`IpcMemoryAttach: shmat() failed: Permission denied`

Una explicación plausible es que otro usuario intentó iniciar un proceso postmaster en el mismo puerto el cual ha adquirido recursos compartidos y luego ha finalizado. Dado que las claves de memoria compartidas de Postgres se basan en el número de puerto asignado al proceso postmaster, estos conflictos tiene más probabilidad de ocurrir si existe más de una instalación en un mismo servidor. Si no hay otros procesos postmaster en ejecución (vea más arriba), ejecute ipcclean e intente nuevamente. Si existen otros postmaster ejecutándose, deberá contactar a los propietarios de estos procesos para coordinar la asignación de puertos y/o la remoción de los segmentos de memoria compartida no utilizados.

Description

postmaster administra la comunicación entre los procesos del cliente y del servidor, así como la asignación de buffers compartidos y semáforos SysV (en máquinas que no tengan intstrucciones del tipo test-and-set). postmaster no interactúa directamente con el usuario y debe ser iniciado como un proceso en segundo plano.

Sólo un postmaster debe estar ejecutándose a la vez para una instalación Postgres dada. Aquí una instalación significa un directorio de base de datos y un número de puerto de postmaster Se puede ejecutar más de un postmaster en una misma máquina si cada uno de ellos tiene un directorio y un número de puerto diferente.

Notes

Siempre que se posible *evite* utilizar `SIGKILL` para forzar la finalizació de postmaster. En su lugar debería utilizarse `SIGHUP`, `SIGINT`, o `SIGTERM` (la señal predeterminada para kill(1))". La utilización

```
% kill -KILL
```

o su forma alternativa

```
% kill -9
```

impedirá que postmaster pueda liberar los recursos del sistema (memoria compartida y semáforos) que poseía antes de finalizar. En cambio, si postmaster logra liberar los recursos en su poder, le evitará a usted tener que lidiar con los problemas de memoria compartida que se describieron anteriormente.

Existen varias utilidades para resolver problemas de memoria compartida, entre las cuales se encuentran ipcs(1), ipcrm(1), y ipcclean(1).

Utilización

Para iniciar postmaster utilizando los valors predeterminados, escriba:

```
% nohup postmaster >logfile 2>&1 &
```

Este comando iniciará postmaster en el puerto predeterminado (5432). Esta es la manera más simple, y la más común, de iniciar postmaster.

Para iniciar postmaster con un número de puerto específico y un nombre de ejecutable:

```
% nohup postmaster -p 1234 &
```

Este comando ejecutará postmaster comunicándose a través del puerto 1234. Para poder conectarse a este postmaster utilizando psql, necesitará ejecutarlo del siguiente modo

```
% psql -p 1234
```

o fijar la variable de entorno PGPORT:

```
% setenv PGPORT 1234
% psql
```

Apéndice UG1. ayuda de fecha/hora

UG1.1. Zonas horarias

Postgresdebe tener información tabular interna para decodificar la zona horaria, desde que no hay un sistema estandar de interface *nix para proveer acceso a lo general, información de zona de tiempo cruzada. El SO subyacente *es* usado para proveer información de zona de tiempo para*salidas*.

Tabla UG1-1. Zonas de tiempo reconocidas porPostgres

Zona de Tiempo	fuera de UTC	descripción
NZDT	+13:00	Hora de luz del día de nueva Zelanda
IDLE	+12:00	Fecha internacional lineal, Este
NZST	+12:00	Hora Std de Nueva Zelanda
NZT	+12:00	Hora de Nueva Zelanda
AESST	+11:00	Hora de verano Std de Australia del este
ACSST	+10:30	Hora de verano Std de Australia Central
CADT	+10:30	Hora de luz del día de Australia
SADT	+10:30	Hora de luz del día de Australia del sur
AEST	+10:00	Hora Std de Australia del este
EAST	+10:00	Hora Std de Australia del Este

Zona de Tiempo	fuera de UTC	descripción
GST	+10:00	Hora de Guam Std, Zona 9 de USSR
LIGT	+10:00	Melbourne, Australia
ACST	+09:30	Hora Std de Australia Central
CAST	+09:30	Hora Std de Australia Central
SAT	+9:30	Hora Std de Australia del sur
AWSST	+9:00	Hora Std de verano de Austrialia del oeste
JST	+9:00	Hora Std de Japón, Zona 8 de USSR
KST	+9:00	Hora estandart de Korea
WDT	+9:00	Hora de luz del día del Oste de Australia
MT	+8:30	Hora de Moluccas
AWST	+8:00	Hora Std de Australia del oeste
CCT	+8:00	Hora de la costa de China
WADT	+8:00	Hora de luz del día del oeste de australia
WST	+8:00	Hora Std del Oeste de Australia
JT	+7:30	Hora de Java
WAST	+7:00	Hora Std del Oeste de Australia
IT	+3:30	Hora de Irán
BT	+3:00	Hora de Baghdad

Zona de Tiempo	fuera de UTC	descripción
EETDST	+3:00	Hora de luz del día en Europa del este
CETDST	+2:00	Hora de luz del día en Europa Central
EET	+2:00	Europa del Este,Zona 1 de USSR
FWT	+2:00	Hora de invierno Frances
IST	+2:00	Hora Std de Israel
MEST	+2:00	Hora de verano de Europa del centro
METDST	+2:00	Hora de luz del día en Europa del centro
SST	+2:00	Hora de verano de Suecia
BST	+1:00	Hora de verano de Inglaterra
CET	+1:00	Hora de Europa central
DNT	+1:00	Hora normal de Dansk
DST	+1:00	Hora estandart de Dansk(?)
FST	+1:00	Hora de verano Francesa
MET	+1:00	Hora de Europa del Centro
MEWT	+1:00	Hora de invierno de Europa del Centro
MEZ	+1:00	Zona de Europa del Centro
NOR	+1:00	Hora estandart de Norway
SET	+1:00	Hora de Seychelles
SWT	+1:00	Hora de invierno de Suecia
WETDST	+1:00	Hora de luz del día del Oeste de Europa
GMT	0:00	Hora principal de Greenwish
WET	0:00	Europa del Oeste

Zona de Tiempo	fuera de UTC	descripción
WAT	-1:00	Hora del oeste de Africa
NDT	-2:30	Hora de luz del día de Newfoundland
ADT	-03:00	Hora de luz del día de Atlantic
NFT	-3:30	Hora estandart de Newfoundland
NST	-3:30	Hora estandart de Newfoundland
AST	-4:00	Hora Std de Atlantic(Canada)
EDT	-4:00	Hora de luz del día del este
ZP4	-4:00	GMT +4 hours
CDT	-5:00	Hora de luz del día Central
EST	-5:00	Hora estandart del este
ZP5	-5:00	GMT +5 hours
CST	-6:00	Hora Std Central
MDT	-6:00	Hora de luz del día de la Montaña
ZP6	-6:00	GMT +6 hours
MST	-7:00	Hora estandart de la montaña
PDT	-7:00	Hora de luz del día del Pacífico
PST	-8:00	Hora Std del Pacífico
YDT	-8:00	Hora de luz del día de Yukon
HDT	-9:00	Hora de luz del día en Hawaii/Alaska
YST	-9:00	Hora estandart de Yukon

Zona de Tiempo	fuera de UTC	descripción
AHST	-10:00	Hora Std de Alaska-Hawaii
CAT	-10:00	Hora de Alaska Central
NT	-11:00	Hora Nome
IDLW	-12:00	Linea de Fecha Internacional, Oeste

UG1.1.1. Zonas Horarias Australianas

Las zonas horarias Australianas y sus variantes de denominación cuentan con un curato de la totalidad de las zonas horarias de la tabla de búsqueda de las zonas horarias de Postgres. Hay dos conflictos de denominación con zonas horarias en común definidas en los Estados Unidos, `CST` y `EST`.

Si la opción del compilador USE_AUSTRALIAN_RULES esta activa entonces `CST` y `EST` se interpretaran siguiendo los convenios Australianos.

Tabla UG1-2. Zonas Horarias Australianas de Postgres

Zona Horaria	Desplazamiento desde UTC	Descripción
CST	+10:30	Tiempo Estándar Central de Australia
EST	+10:00	Tiempo Estándar Oriental de Australia

UG1.1.2. Interpretación de las entradas de Fecha/tiempo

Los tipos de fecha/tiempo son todos decodificados usando un conjunto de rutinas

comunes.

Interpretación de las entradas de Fecha/tiempo

1. Partiendo la cadena de entrada en muestras y clasificando cada uno de las marcas como cadena, tiempo, zona horaria, o número.

    a. Si la muestra contiene dos puntos (":"), esto es una cadena de tiempo.

    b. si la muestra contiene un guión ("-"), barra ("/"), o un punto ("."), esto es una cadena de fecha que puede tener el nombre del mes.

    c. Si la muestra es solamente numérica, entonces es cualquiera de estas opciones un campo sencillo un fecha concatenada ISO-8601 (p.e. "19990113" para 13 Enero del 1999) o tiempo (p. e. 141516 para 14:15:16).

    d. Si la muestra comienza con un mas ("+") o un menos ("-"), entonces es o una zona horaria o un campo especial.

2. Si la muestra es una cadena de texto, compara con posibles cadenas.

    a. Hacer un búsqueda binaria en la tabla de consulta de la muestra para cada cadena especial (p. e. `today`), day (p. e. `Thursday`), month (p. e. `January`), o noise word (p. e. `on`).

    Pone los valores del campo y la mascara de bit para los campos. Por ejemplo, pone año, mes, día para `today`, y adicionalmente hora, minutos, segundos para `now`.

    b. Si no lo encuentra, hace una búsqueda binaria similar en la tabla de consulta para encontrar la muestra a la zona horaria.

    c. Si no lo encuentra, lanza un error.

3. La muestra es un número o un campo numérico.

    a. Si hay más de 4 dígitos, y si no se ha leído con posterioridad otro campo de tipo fecha, entonces lo interpretará como un "fecha concatenada" (e.g.

`19990118`). Con 8 y 6 dígito se interpreta como año, mes, y día, mientras que con 7 y 5 dígitos se interpreta como año, día del año, respectivamente.

b. Si la muestra tiene 3 dígitos y un año ha sido decodificado, entonces se interpreta como día del año.

c. Si es más largo que dos dígitos, entonces se interpreta como el año.

d. Si está en modo fecha Europea, y si el campo día no ha sido leído todavía, y si el valor es más pequeño o igual a 31, entonces se interpreta como un día.

e. Si el campo mes no ha sido leído todavía, y si el valor es más pequeño o igual que 12, entonces se interpreta como un mes.

f. Si el campo día no ha sido leído todavía, y si el valor es más pequeño o igual que 31, entonces se interpreta como un día.

g. Si no, se interpreta como un año.

4. Si se ha especificado AC, anula el año y desplaza uno al almacenado interno (no hay año cero en el calendario Gregoriano, pero numéricamente 1AC es el año cero).

5. Si no se ha especificado, y si el campo año tiene dos dígitos de longitud, entonces ajustamos el año a 4 dígitos. Si el campo no es más pequeño que 70, entonces sumamos 2000; si no, sumamos 1900.

Sugerencia: Los años Gregorianos 1-99AD pueden ser introducidos usando 4 dígitos precedidos por ceros (p. e. 0099 es 99AD). Los tres dígitos también son aceptados como un año bajo muchas circunstancias, sin embargo dependiendo de la posición la cadena numérica puede ser interpreta en lugar de un día.

UG1.2. Historia

Nota: Contrido por José Soares (jose@sferacarta.com).

El día Juliano fue inventado por erudito francés Joseph Justus Scaliger (1540-1609) y probablemente coge su nombre del padre Scaliger, el erudito italiano Julius Caesar Scaliger (1484-1558). los astrónomos tienen que usar el periodo Juliano para asignar un único número cada día desde 1 de Enero de 4713 AC. Esto es el llamado Día Juliano (JD). JD 0 designa 24 horas, del mediodía UTC del 1 de Enero de 4713 AC hasta el mediodía UTC del 2 Enero 4713 AC.

“Día Juliano” es diferente que “Fecha Juliana”. El calendario Juliano fue introducido por Julius Caesar en 45 AC. Fue usado comúnmente hasta el 1582, donde países empezaron a cambiarse al calendario Gregoriano. en el calendario Juliano, el año tropical es aproximadamente como 365 1/4 días = 365.25 días. Esto da un error de un día en aproximadamente 128 días. El error acumulado del calendario movió al Papa Gregorio XIII ha reformar el calendario acorde con las instrucciones del Concilio de Trento.

En el calendario Gregoriano, el año tropical es aproximadamente 365 + 97 / 400 días = 365.2425 días. Así coge aproximadamente 3300 años para el año tropical se desplace un día con respecto al calendario Gregoriano.

La aproximación 365+97/400 esta lograda mediante 97 años bisiestos cada 400 años, usando las siguientes reglas:

Cada año divisible por 4 es un año bisiesto.
Sin embargo, cada año divisible por 100 no es un año bisiesto.
Sin embargo, cada año divisible por 400 es un año bisiesto después de todo.

De este modo, 1700, 1800, 1900, 2100, y 2200 no son años bisiestos. pero 1600, 2000, y 2400 son años bisiestos. Por el contrario, en el viejo calendario Juliano sólo los años divisibles por 4 son años bisiestos.

La bula papal de Febrero del 1582 decretó que se debía quitar 10 días a Octubre de

1582, así que el 15 de Octubre debe seguir inmediatamente después del 4 de Octubre. Esto se observó en Italia, Polonia, Portugal, y España. Los otros países Católicos lo siguieron poco después, pero los países Protestantes se resistieron al cambio, y los países ortodoxos griegos no cambiaron hasta que no empezó este siglo. La reforma fue observada por Gran Bretaña y sus Colonias (incluido lo que ahora es USA) en 1752. Así el 2 Septiembre 1752 fue seguido por el 14 Septiembre 1752. Esto es lo que tiene cal de los sistemas UNIX produciendo lo siguiente:

```
% cal 9 1752
   Septiembre 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

> **Nota:** SQL92 dista que “Dentro de la definición del fecha/tiempo literal , los valores fecha/tiempo están restringidos por las reglas naturales para las fechas y los tiempos acorde con el calendario Gregoriano ”. Las Fechas entre 1752-09-03 y 1752-09-13, aunque se han eliminado en algunos países por el "fiat" Papal, conforme a las “reglas naturales” y son por lo tanto fechas validas.

Diferentes calendarios han sido desarrollados en varios lugares del mundo, muchos proceden del sistema Gregoriano. Por ejemplo, Los principios del calendario Chino pueden remontarse hasta el siglo 14 AC. La leyenda dice que el Emperador Huangdi inventó el calendario en 2637 AC. La gente de la República de China usa el calendario Gregoriano para uso civil. El calendario Chino es utilizado para las fiestas.

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.