Estudio del

Sistema de Gestión de Base de Datos PostgreSQL

AUTOR	**Javier Novella Latorre**
DIRECTOR	**Juan Carlos Casamayor Ródenas**
FECHA	30 Septiembre 2012
TITULACIÓN	Ingeniería Informática

Contenidos

TEMA 0 – INTRODUCCIÓN

¿QUÉ ES?

PostgreSQL es un sistema de gestión de bases de datos que incorpora el modelo relacional para sus bases de datos y usa el lenguaje SQL como lenguaje de consulta. La base de datos relacional PostgreSQL es una de las aplicaciones de código abierto con más éxito de los últimos años, seguido por muchos desarrolladores y usuarios. Es una buena herramienta para crear una aplicación con grandes cantidades de información no trivial se puede beneficiar de él. PostgreSQL es una excelente implementación de una base de datos relacional, con todo tipo de funcionalidades, de código abierto y de uso gratuito.

PostgreSQL puede ser usado desde cualquiera de los lenguajes de programación más usados (C, C++, Perl, Python, Java, Tcl, PHP,…). Sigue muy de cerca los estándares de lenguajes de consulta (SQL:2008), y sigue desarrollando los siguientes estándares.

La mayoría de aplicaciones no triviales manejan grandes cantidades de información, y muchas aplicaciones están programadas para manejar la información y no para hacer cálculos. Se estima que actualmente el 80% del desarrollo de aplicaciones en el mundo está conectado de alguna forma a datos complejos almacenados en una base de datos, por lo que las bases de datos son una base importante para muchas aplicaciones.

PostgreSQL es muy competente, muy fiable, y con un buen rendimiento. Se puede ejecutar en cualquier plataforma UNIX (FreeBSD, Linux, MacOS), servidores WINDOWS (NT, 2000, 2003), o incluso en Windows XP para desarrollo. Comparándolo con otros sistemas de gestión de bases de datos, PostgreSQL contiene todas las características que se pueden encontrar tanto en sistemas comerciales como de código abierto, además de incorporar nuevas funcionalidades que sólo PostgreSQL tiene.

HISTORIA

Aunque el proyecto PostgreSQL tal y como se conoce hoy en día empezó en 1996, las bases y el trabajo en la que se asienta tienen sus comienzos en la década de los 70.

-Berkeley (1977–1985)

En la década de los 70 se desarrollan nuevos conceptos en el mundo de los gestores de las bases de datos, y es en 1973 cuando IBM empieza a trabajar con los primeros conceptos y teorías sobre las bases de datos relacionales, consiguiente la primera implementación para el lenguaje SQL con su proyecto ‘System R’. Este proyecto de IBM también desarrolló un diseño y algoritmos que influyeron posteriormente.

El ancestro de PostgreSQL es Ingres (INteractive Graphics REtrieval System), desarrollado inicialmente en la Universidad de Berkeley por Michael Stonebraker. A partir de Ingres,

Stonebraker introdujo los nuevos conceptos de IBM sobre datos relacionales. A principio de los 80, el código Ingres es comprado por Computer Associates, y estuvo compitiendo con Oracle por el liderazgo en el mundo de bases de datos relacionales y su código e implementación evolucionaron y fueron el origen de otras bases de datos relacionales como Informix, NonStop SQL y Sybase (Microsoft SQL Server fue una versión licenciada de Sybase hasta su versión 6.0).

Michael Stonebraker dejo la Universidad de Berkeley en 1982 para comercializar Ingres pero volvió a la misma en 1985 con nuevas ideas.

-Postgres (1986–1994)

Cuando Stonebraker vuelve a Berkeley en 1985 empieza a liderar un nuevo proyecto sobre un servidor de base de datos relacional de objetos llamado Postgres (después de Ingres), patrocinado por la Defense Advanced Research Projects Agency (DARPA), la Army Research Office (ARO), la National Science Foundation (NSF), y ESL, Inc. Con este proyecto y basándose en la experiencia obtenida con Ingres, Stonebraker tenía como meta mejorar lo que habían conseguido y aprendido en el desarrollo de Ingres. Y aunque se basó en muchas ideas de Ingres, no partió del código fuente del mismo.

Los objetivos iniciales de este proyecto fueron:

- Proporcionar un mejor soporte para objetos complejos.
- Proporcionar a los usuarios la posibilidad de extender los tipos de datos, operadores y métodos de acceso.
- Proporcionar los mecanismos necesarios para crear bases de datos activas (triggers, etc.).
- Simplificar el código encargado de la recuperación del sistema después de una caída del mismo.
- Hacer cambios mínimos en el modelo relacional.
- Mejorar el lenguaje de consulta QUEL heredado de Ingres (POSTQUEL).

La última versión de Postgres en este proyecto fue la versión 4.2.

-Postgres95 (1994–1995)

Dos estudiantes graduados en Berkeley, Jolly Chen y Andrew Yu, añadieron capacidades de SQL a Postgres. El proyecto resultante se llamó Postgres95. Hicieron una limpieza general del código, arreglaron errores en el mismo, e implementaron otras mejoras, entre las que destacan:

- Sustitución de POSTQUEL por un intérprete del lenguaje SQL.
- Reimplementación de las funciones agregadas.
- Se crea ‘psql’ para ejecutar consultas SQL.
- Se revisa la interfaz de objetos grandes (large object).
- Se crea un tutorial sobre Postgres.
- Postgres se pudo empezar a compilar con ‘GNU make’ y ’GCC’ sin parchear.

La versión 1.0 de Postgre95 vio la luz en 1995, el código era 100% ANSI C, un 25% más corto en relación con la versión 4.2 y un 30-50% más rápido. El código fue publicado en la web y liberado bajo una licencia BSD, y más y más personas empezaron a utilizar y a colaborar en el proyecto.

-PostgreSQL (1995–1996)

En 1996, Andrew Yu y Jolly Chen ya no tenían tanto tiempo para dirigir y desarrollar Postgres95. Los dos dejaron Berkeley pero Chen continuó manteniendo Postgres95. Algunos de los usuarios habituales de las listas de correo del proyecto decidieron hacerse cargo del mismo y crearon el llamado "PostgreSQL Global Development Team". En un principio este equipo de desarrolladores al cargo de la organización del proyecto estuvo formado por Marc Fournier en Ontario, Canada, Thomas Lockhart en Pasadena, California, Vadim Mikheev en Krasnoyarsk, Rusia y Bruce Momjian en Philadelphia, Pennsylvania. En el verano de 1996, había una gran demanda por un servidor de base de datos SQL de código abierto. Fournier ofreció un host para una lista de correo y para almacenar el código, y mil suscriptores se añadieron a la nueva lista de correo. El servidor se configuró para que unos pocos usuarios pudieran conectarse para arreglar los distintos fallos, usando el sistema de control de versiones CVS. Jolly Chen declaró: “Este proyecto necesita poca gente con mucho tiempo, no mucha gente con poco tiempo”, dado que se alcanzaron las 250.000 líneas en código C.

Al añadir propiedades de SQL al proyecto, el nombre fue cambiado de Postgres95 a PostgreSQL y lanzaron la versión 6.0 en enero de 1997. Se crearon módulos independientes con las distintas funcionalidades, y salían nuevas versiones cada pocos meses.

-PostgreSQL (1996 – actualidad)

Hoy en día el grupo central (core team) de desarrolladores está formado por 6 personas, existen 38 desarrolladores principales y más 21 desarrolladores habituales. En total alrededor de 65 personas activas, contribuyendo con el desarrollo de PostgreSQL. Existe también una gran comunidad de usuarios, programadores y administradores que colaboran activamente en numerosos aspectos y actividades relacionadas con el proyecto. Informes y soluciones de problemas, tests, comprobación del funcionamiento, aportaciones de nuevas ideas, discusiones sobre características y problemas, documentación y fomento de PostgreSQL son solo algunas de las actividades que la comunidad de usuarios realiza.

También es importante señalar que existen muchas empresas que colaboran con dinero y/ó con tiempo/personas en mejorar PostgreSQL. Muchos desarrolladores y nuevas características están muchas veces patrocinadas por empresas privadas. En los últimos años los trabajos de desarrollo se han concentrado mucho en la velocidad de proceso y en características demandadas en el mundo empresarial.

-Resumen

Durante los años de existencia del Proyecto PostgreSQL, el tamaño del mismo, tanto en número de desarrolladores como en líneas de código, funciones y complejidad del mismo, ha

ido aumentando año a año. La siguiente tabla muestra aproximadamente la evolución de PostgreSQL:

Año	Versión	Líneas de código
1996	6.0	173K
1997	6.1, 6.2	224K
1998	6.3, 6.4	312K
1999	6.5	355K
2000	7.0	419K
2001	7.1	451K
2002	7.2, 7.3	520K
2003	7.4	590K
2004	8.0	634K
2005	8.1	675K
2006	8.2	736K
2007	-	-
2008	8.3	814K
2009	8.4	897K
2010	9.0	969K

RESUMEN GENERAL DE CARACTERÍSTICAS

-Generales

- Es una base de datos 100% ACID
- Integridad referencial
- Tablespaces
- Transacciones anidadas
- COMMIT y ROLLBACK
- PITR
- Copias de seguridad
- Juegos de caracteres internacionales
- Múltiples métodos de autentificación
- Acceso encriptado vía SSL
- Actualizaciones integradas
- Completa documentación
- Licencia BSD
- Disponible para Linux y UNIX en todas sus variantes y Windows 32/64bits

-Programación / Desarrollo

- Funciones/procedimientos almacenados en numerosos lenguajes de programación, entre otros PL/pgSQL (similar al PL/SQL de Oracle), PL/Perl, PL/Python y PL/Tcl
- Bloques anónimos de código de procedimientos
- Numerosos tipos de datos y posibilidad de definir nuevos tipos

- Soporta el almacenamiento de objetos binarios grandes (gráficos, videos, sonido, ...)
- APIs para programar en C/C++, Java, .Net, Perl, Python, Ruby, Tcl, ODBC, PHP, Lisp, Scheme, Qt y muchos otros.

-SQL

- SQL92, SQL99, SQL2003, SQL2008
- Claves primarias (PK) y claves ajenas (FK)
- Restricciones
- Restricciones de unicidad diferidas
- Columnas auto-incrementales
- Índices
- Subconsultas
- Consultas recursivas
- Funciones 'Windows'
- Uniones de tablas
- Vistas
- Triggers
- Reglas
- Herencia de tablas

-Limitaciones

Límite	Valor
Máximo tamaño base de dato	Ilimitado
Máximo tamaño de tabla	32 TB
Máximo tamaño de fila	1.6 TB
Máximo tamaño de campo	1 GB
Máximo numero de filas por tabla	Ilimitado
Máximo numero de columnas por tabla	250 - 1600 (dependiendo del tipo)
Máximo numero de indices por tabla	Ilimitado

VERSIONES

Cada versión que PostgreSQL lanza está controlada con mucho detalle, y cada versión beta tiene al menos un mes de prueba. Con una gran comunidad y acceso abierto a su código, los fallos se corrigen rápidamente.

El Proyecto PostgreSQL tiene como objetivo mantener y dar soporte a cada versión de PostgreSQL durante 5 años desde el momento de su lanzamiento. El resumen de las versiones está en la tabla siguiente:

Versión	Versión menor	Soportada	Lanzamiento	Soporte
9.2	9.2.0	Sí	Septiembre 2012	Septiembre 2017

9.1	9.1.5	Sí	Septiembre 2011	Septiembre 2016
9.0	9.0.9	Sí	Septiembre 2010	Septiembre 2015
8.4	8.4.13	Sí	Julio 2009	Julio 2014
8.3	8.3.20	Sí	Febrero 2008	Febrero 2013
8.2	8.2.23	No	Diciembre 2006	Diciembre 2011
8.1	8.1.23	No	Noviembre 2005	Noviembre 2010
8.0	8.0.26	No	Enero 2005	Octubre 2010
7.4	7.4.30	No	Noviembre 2003	Octubre 2010
7.3	7.3.21	No	Noviembre 2002	Noviembre 2007
7.2	7.2.8	No	Febrero 2002	Febrero 2007
7.1	7.1.3	No	Abril 2001	Abril 2006
7.0	7.0.3	No	Mayo 2000	Mayo 2005
6.5	6.5.3	No	Junio 1999	Junio 2004
6.4	6.4.2	No	Octubre 1998	Octubre 2003
6.3	6.3.2	No	Mar 1998	Mar 2003

La última versión de PostgreSQL, la 9.2, contiene muchas nuevas características y mejoras en el rendimiento profesional diseñadas para facilitar el uso a gran escala. La diferencia más notable para su uso para grandes bases de datos es que se puede ejecutar una instancia sobre 64 núcleos de un procesador (en la versión anterior 9.1 el límite era de 16). Añade también nuevas funcionalidades fuera del estándar SQL, como es el nuevo acceso a los datos mediante el formato de intercambio de datos JSON (JavaScript Object Notation). Respecto al rendimiento en la última versión, y según los desarrolladores, PostgreSQL puede responder hasta 350.000 consultas sencillas por segundo, cuatro veces más que en la versión anterior, y realizar 14.000 escrituras por segundo, cinco veces más que en la versión anterior. La nueva funcionalidad de escaneo 'index-only' puede acelerar las consultas hasta 20 veces, según el grupo de desarrollo de PostgreSQL.

CÓDIGO ABIERTO

El término 'código abierto' tiene un significado muy específico cuando se usa en programas. Significa que el programa se distribuye junto con el código. Puede tener algún tipo de condición sobre el uso del programa, y tiene licencias para un uso determinado. Una licencia de código abierto permite usar el programa, modificarlo, y distribuirlo sin tener que pagar licencias, incluso para usos comerciales.

Al disponer del código, es posible corregirlo, incluso algunas empresas ofrecen soporte para algunas aplicaciones de código abierto. Hay distintas versiones de licencias de código abierto, aunque todas tienen en común que disponen el código fuente con el programa, y que permiten una redistribución.

La licencia más liberal es la Berkeley Software Distribution (BSD), que es la que tiene PostgreSQL, y tiene como lema "se concede permiso para usar, copiar, modificar y distribuir

este programa y su documentación para cualquier propósito, sin pagos, y sin un acuerdo por escrito, siempre que el aviso de copyright aparezcan en todas las copias"

PREMIOS

- 2008: Developer.com, producto del año, herramienta para bases de datos.
- 2006: Linux Journal Editors' Choice Awards, mejor base de datos.
- 2005: Linux Journal Editors' Choice Awards, mejor base de datos.
- 2004: Linux Journal Editors' Choice Awards, mejor base de datos.
- 2004: Ars Technica, mejor aplicación de servidor.
- 2004: Linux New Media, mejor base de datos.
- 2003: Linux Journal Editors' Choice Awards, mejor base de datos.
- 2002: Linux New Media, mejor base de datos.
- 2000: Linux Journal Editors' Choice Awards, mejor base de datos.
- 1999: LinuxWorld, mejor base de datos.

BIBLIOGRAFÍA

- [1] "PostgreSQL Introduction and Concepts"
- [2] "Beggining Databases with PostgreSQL: From Novice to Professional, Second Edition"
- [3] http://www.postgresql.org/about/awards/
- [3] http://www.postgresql.org.es/sobre_postgresql
- [12] http://www.computerworld.com.au/article/435977/postgres_9_2_invites_large-scale_enterprise_deployment/

TEMA 1 – SISTEMA DE GESTIÓN DE BASES DE DATOS

BASES DE DATOS

Según la RAE, una base de datos es un "conjunto de datos organizado de tal modo que permita obtener con rapidez diversos tipos de información". Un sistema de gestión de bases de datos (SGBD o en inglés DBMS) es un conjunto de librerías, aplicaciones y utilidades que permiten a un desarrollador no tener que preocuparse por detalles del almacenamiento y manejo de la información. También ofrece utilidades para la búsqueda y actualización de registros.

MODELOS DE BASES DE DATOS

Durante las décadas de los 60 y 70, los desarrolladores crearon bases de datos para solucionar las repeticiones de información de distintas formas, y esos métodos crearon los términos para los modelos de los sistemas de bases de datos. IBM desarrolló muchas de las bases para estos modelos, que aun están en uso a día de hoy.

Los diseños iniciales se basaban en la eficiencia, y una forma de hacer los sistemas más eficientes era forzar registros de longitud fija en las bases de datos, o al menos tener un número fijo de elementos por registro (columnas por filas), esto evita el problema de los datos repetidos. En la vida real hay que encontrar la forma de estructurar la información de forma conveniente, y los diseñadores lo solucionaron introduciendo distintos tipos de datos en las bases de datos.

-Modelo Jerárquico:

En los últimos años de los 60, IBM introdujo el modelo jerárquico para las bases de datos en su base de datos IMS. Este modelo considera los registros de información como colecciones de otros registros, solucionando el problema de la repetición. Estos modelos aun se usan en aplicaciones actuales, como ADABAS, y optimizan el almacenamiento para hacerlo más eficiente en casos particulares.

-Modelo de Red:

El modelo de red introduce la idea de punteros, es decir, los registros pueden contener referencias a otros registros. Una base de datos con modelo de red permite obtener de forma muy rápida todos los registros de un mismo tipo que están relacionados con un registro dado de otro tipo, siguiendo los punteros desde el registro inicial. Por otra parte tiene algunas desventajas, por ejemplo al buscar alguna condición se tiene que seguir todos los punteros de todos los registros, consiguiendo una búsqueda extremadamente lenta en bases de datos grandes. Además crear aplicaciones que usen un modelo de red puede ser tedioso, ya que la aplicación debe crear y mantener los punteros mientras se actualizan o borran registros.

-Modelo Relacional:

La teoría de los sistemas de gestión de bases de datos tomó un gran impulso en 1970 con la publicación de "Un modelo relacional de datos para grandes bancos de datos compartidos", de Edgar Frank Codd. Este trabajo introdujo la idea revolucionaria de las relaciones y mostró cómo podrían usarse tablas para representar hechos relacionados con objetos reales, y mantener la información sobre ellos. La idea inicial de priorizar la eficiencia en las bases de datos ya no era tan importante como mantener la integridad de la información, y el modelo relacional enfatizaba en la integridad mucho más que modelos anteriores.

BASE DE DATOS RELACIONAL (TABLAS)

El propósito de toda base de datos es almacenar y obtener una cantidad grande de datos. Hoy en día la gran mayoría de los sistemas de bases de datos son "bases de datos relacionales". Tienen una base matemática, pero en la práctica una base de datos relacional significa un almacenamiento con estructura uniforme de información de distinto tipo. Las tablas son el fundamento de los sistemas de gestión de bases de datos relacionales, y en ellas se almacenan los datos que forman la información. Cada una de las tablas tiene un nombre que la identifica y un número de registros (o columnas) con distinta información. Cada columna está definida por un tipo de datos (caracteres, números, fechas,…) haciendo una estructura consistente, rápida y fiable.

Un ejemplo es una tabla Amigo y sus posibles registros podrían ser Nombre, Apellidos, Ciudad y Edad. Por tanto cada amigo sería una fila y cada fila en la tabla mostraría información de un único amigo.

AMIGO	Nombre	Apellidos	Ciudad	Edad
1	Miguel	Muñoz	Zaragoza	19
2	Sara	Aguilar	Madrid	23
3	Samuel	Jiménez	A Coruña	41

La integridad referencial se refiere a mantener la información con sentido en cualquier momento. Los registros en una tabla se llaman tuplas, que son grupos ordenados de componentes o atributos, y cada componente está definido por un tipo. Diversas reglas importantes definen un sistema de gestión de bases de datos. Todas las tuplas deben seguir el mismo patrón en el que todas deben tener el mismo número y tipo de componentes. El conjunto de tuplas forma una tabla. Además en una tabla no debe haber tuplas duplicadas y cada atributo debe ser atómico (una única información). El atributo para distinguir un registro de otro se llama clave primaria (PK).

Para crear las tablas correctamente se pueden seguir las siguientes reglas:

-Regla 1 – Separar la información en columnas:

Hay que poner cada dato de información por separado. En general es una tarea sencilla, pero hay que tener claro la información que se va a querer obtener durante el mantenimiento de la base de datos, y el tipo de información de cada columna.

-Regla 2 – Tener un identificador único para cada fila:

Para evitar las repeticiones y poder identificar cada fila aunque la información se haya ido modificando con el tiempo. Para esto se crea la PK comentada anteriormente.

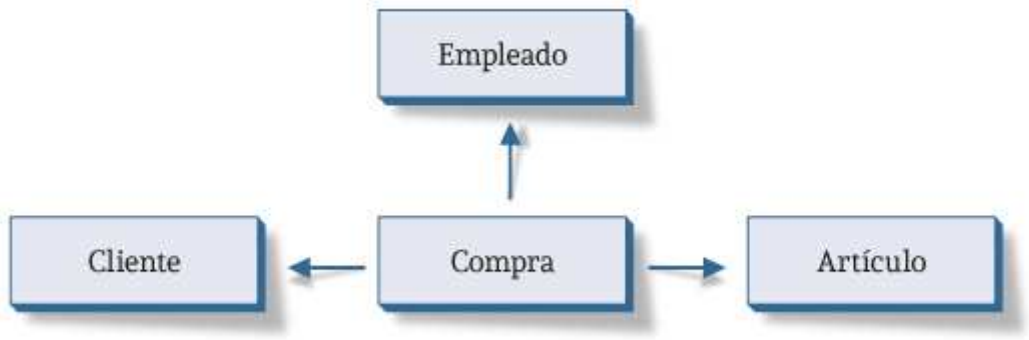
-Regla 3 – Eliminar información repetida:

Cuando cierta información se repita en distintas filas de la misma tabla, y ante la actualización de una haya que actualizar las demás de la misma forma (relación de una a muchas), es porque realmente esa información debe estar en otra tabla. Posteriormente será sencillo poder sacar información de distintas tablas y mostrarla como información compacta.

-Regla 4 – Elegir correctamente los nombres:

Los nombres de las tablas y las columnas deben ser cortos, con significado y con sentido. Si alguna columna es difícil de nombrar, posiblemente esté incorrectamente separada (regla 1). Cada diseñador de bases de datos puede tener su forma de hacerlo, pero es importante seguir un mismo modelo, como por ejemplo usar plurales para las tablas y singulares para las columnas, o usar el prefijo 'id_' para las columnas que sean PK de la tabla

Una vez diseñadas las tablas, hay que diseñar la base de datos, por ejemple mediante un diagrama:



En el diagrama las flechas van desde "compra" a las demás tablas, esto es porque cada compra tendrá asociada uno o más clientes, uno o más empleados y uno o más artículos. Para que fuese más completo, el diagrama puede completarse con la información de las columnas de cada tabla, las PK, las relaciones exactas entre las columnas de cada tabla,…. Si por ejemplo cada artículo pudiese ser vendido únicamente por uno o varios empleados, habría que añadir otra relación (flecha) desde artículo hasta empleado.

TIPOS DE DATOS BÁSICOS

Aunque PostgreSQL admite muchos tipos de datos, hay ciertos tipos de datos que son comunes a la gran mayoría de sistemas de gestión de bases de datos:

CATEGORÍA	TIPO	DESCRIPCIÓN
Texto	CHAR(longitud)	Cadena de caracteres de longitud fija
	VARCHAR(longitud)	Cadena de caracteres de tamaño variable
Número	INTEGER	Entero con rango +/- 2 billones
	FLOAT	Número en punto flotante con 15 dígitos de precisión
	NUMERIC(precisión, decimal)	Número con precisión y decimales definido por el usuario
Fecha/Tiempo	DATE	Fecha
	TIME	Tiempo (horas, minutos, segundos,…)
	TIMESTAMP	Fecha y tiempo

-Integer:

Un número entero.

-Serial:

Un número entero, pero asignado automáticamente a un número único para cada fila. El propósito de este tipo es no tener que asignar a mano los identificadores o PK.

-Char:

Una cadena de caracteres de tamaño fijo, con el tamaño mostrado entre paréntesis tras el tipo.

-Varchar:

Una cadena de caracteres de tamaño variable, con el tamaño máximo posible mostrado entre paréntesis tras el tipo.

-Date:

Una fecha almacenada como año, mes, día, hora, minuto y segundos. La precisión más allá de los segundos depende de cada sistema.

-Numeric:

Un número con un número específico de dígitos (el primer número entre paréntesis), y con un número fijo de decimales (el segundo número entre paréntesis).

NULOS

En algunos casos algunos campos no tienen información, bien porque se desconoce, bien porque no se desea incluir, o bien porque se va a rellenar posteriormente. En el ejemplo de la tabla ‘amigo’, se podría desconocer el apellido en una de las filas, pero aun así querer meter la información. Una solución sería poner una cadena que dejara clara la situación, por ejemplo

'desconocido', pero haría falta conocer la base de datos en profundidad para conocer estos casos. Además, si el valor no fuera alfanumérico, se debería crear un valor en cada tipo de datos que se quisiera incluir valores nulos.

Para este caso particular, los sistemas de bases de datos dan soporte a un valor especial llamado NULL, que significa 'desconocido', y se trata de forma especial, ya que cualquier valor de cualquier tipo puede tomar el valor NULL (a no ser que un campo se diseñe para que expresamente tenga valor definido).

Para tipos de datos numéricos hay que diferenciar el cero y el NULL, ya que el primero sí es un valor conocido, y el segundo no. De la misma forma para tipos de datos alfanuméricos hay que diferenciar la cadena vacía '' y el NULL, por la misma razón. Por ejemplo si se calcula la media entre distintos números, el valor cero se tiene en cuenta para la media y el NULL no (la media de (0,2) es 1, y la media de (NULL, 2) es 2).

Hay otra forma de entender el valor NULL, y es cuando realmente no puede existir valor. Por ejemplo si se quiere guardar el segundo apellido de un amigo inglés, al no tener no se podrá poner valor.

SISTEMA DE GESTIÓN DE BASES DE DATOS

Un sistema de gestión de bases de datos es un conjunto de programas que permiten la construcción de bases de datos, y de las aplicaciones que los usan. Entre las responsabilidades que tiene que cumplir están:

-Crear la base de datos:

Algunos sistemas manejan un gran archivo y crean uno o más bases de datos en él. Otros pueden usar ficheros de sistemas operativos o usar directamente particiones de disco. Los usuarios no deben preocuparse sobre la estructura a bajo nivel de estos archivos, por lo que el sistema de gestión debe proporcionar todos los accesos necesarios para desarrolladores y usuarios.

-Proporcionar utilidades para consulta y actualización:

Un sistema de gestión de bases de datos debe tener los métodos para consultar información que cumple unos criterios, como por ejemplo los pedidos de un cliente en una empresa. Esta propiedad está ligada al estándar SQL.

-Multitarea:

Si una base de datos se usa en varias aplicaciones, o si es accedida a la vez por distintos usuarios, el sistema debe asegurarse que cada solicitud se procesa sin interferir en las demás. Esto significa que los usuarios necesitan esperar sólo si otra solicitud está escribiendo en la misma información que está consultando (o modificando). Es posible tener varias consultas

(lecturas) al mismo tiempo. En la práctica, diferentes bases de datos soportan diferentes grados de multitarea, así como grados de configuración.

-Mantener un seguimiento:

El sistema debe mantener un log de todos los cambios de la información en un periodo de tiempo. Esto se puede usar para investigar errores, y lo que es más importante, para reconstruir información ante un fallo del sistema, como puede ser un apagado.

-Manejar la seguridad de la base de datos:

El sistema proporciona controles de acceso para que únicamente los usuarios autorizados manejen la información de la base de datos y la estructura de los datos (atributos, tablas e índices). Generalmente existe una jerarquía de usuarios, desde un superusuario que puede cambiar todo, pasando por usuarios que pueden añadir y borrar información, hasta unos usuarios que sólo pueden ver la información. Los sistemas deben proporcionar las herramientas para añadir y borrar usuarios, y para especificar qué pueden hacer.

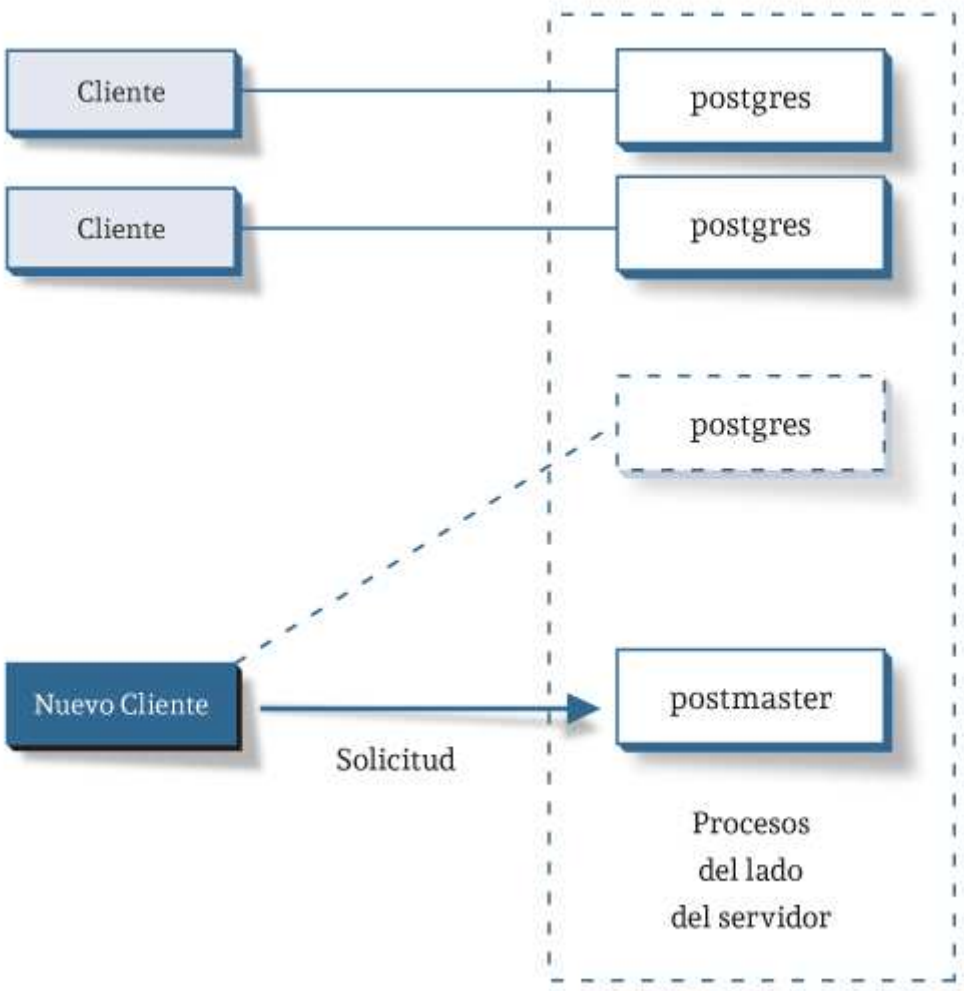
-Mantener la integridad referencial:

Muchos sistemas proporcionan herramientas para mantener la información de forma correcta, e informan cuando una consulta o modificación intenta romper esta integridad.

ARQUITECTURA

Uno de los puntos fuertes de PostgreSQL es su arquitectura. En común con los sistemas comerciales usa un entorno cliente/servidor que aporta beneficios tanto a los usuarios como a los desarrolladores. El punto central de una instalación de PostgreSQL es el proceso de servidor de la base de datos. Se ejecuta en un único servidor y las aplicaciones que necesitan acceder a la información almacenada en la base de datos requieren acceder pasando por el proceso. Estos programas clientes no pueden acceder a la información directamente aunque se estén ejecutando en el mismo ordenador que el proceso de servidor.

Esta separación entre clientes y servidores permite a las aplicaciones estar distribuidas, por ejemplo para implementar una base de datos en UNIX y crear programas clientes en Windows. La siguiente figura muestra una típica aplicación distribuida de PostgreSQL:



En la figura se ven los distintos clientes conectándose al servidor a través de una red, que PostgreSQL crea usando una red TCP/IP. Cada cliente se conecta al proceso de servidor de la base de datos general (postmaster), que es el que crea los nuevos procesos de servidor para dar servicio a cada cliente.

Una sesión en PostgreSQL consiste en los siguientes componentes que interactúan entre ellos:

- Un proceso demonio supervisor (postmaster).
- Un programa cliente sobre la que trabaja el usuario (frontend), como son pgAdmin III y psql.
- Un programa servidor de base de datos en segundo plano (postgres).

Un único postmaster controla una colección de bases de datos almacenados en un único host (equipo anfitrión). En un host se ejecuta solamente un proceso postmaster y múltiples procesos postgres. Los clientes pueden ejecutarse en el mismo sitio o en equipos remotos conectados por TCP/IP. Una colección de bases de datos se suele llamar una instalación. Es posible restringir el acceso a usuarios o a direcciones IP modificando las opciones del archivo 'pg_hba.conf'. _Este archivo junto con PostgreSQL.conf son particularmente importantes porque algunos de sus parámetros de configuración por defecto provocan multitud de

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

Un proceso servidor postgres puede atender exclusivamente a un solo cliente, es decir, hacen falta tantos procesos servidor postgres como clientes haya. El proceso postmaster es el encargado de ejecutar un nuevo servidor para cada cliente que solicite una conexión. Las aplicaciones de frontend que quieren acceder a una determinada base de datos dentro de una instalación hacen llamadas a la librería, y ésta envía peticiones de usuario a través de la red al postmaster (estableciendo una conexión), el cual en respuesta inicia un nuevo proceso en el servidor (backend) y conecta el proceso de frontend al nuevo servidor. A partir de este punto, el proceso de frontend y el servidor en backend se comunican sin la intervención del postmaster. Aunque el postmaster está siempre ejecutándose, esperando peticiones, tanto los procesos de frontend como los de backend se comunican entre ellos. La librería 'libpq' permite a un único proceso en frontend realizar múltiples conexiones a procesos en backend. Aunque la aplicación frontend todavía es un proceso en un único hilo. Conexiones multihilo entre el frontend y el backend no están soportadas de momento en libpq. Una implicación de esta arquitectura es que el postmaster y el proceso backend siempre se ejecutan en la misma máquina (el servidor de la base de datos), mientras que la aplicación en frontend puede ejecutarse desde cualquier sitio. Esto debe tenerse en cuenta porque los archivos de la máquina del cliente pueden no ser accesibles (o sólo pueden se accesibles usando un nombre de archivo diferente) desde la máquina del servidor de base de datos.

Los servidores del postmaster y PostgreSQL se ejecutan usando el identificado de usuario del superusuario de PostgreSQL, y todos los archivos relacionados con la base de dados deben pertenecer a este superusuario.

Concentrando la manipulación de información en un servidor, en vez de controlar a muchos clientes accediendo al mismo almacén de datos en un directorio común, permite a PostgreSQL mantener eficientemente la integridad incluso con muchos usuarios simultáneos. Los programas cliente se conectan usando un protocolo de mensajes específico para PostgreSQL. Sin embargo es posible instalar programas en el cliente que proporcionen una interfaz estándar para que la aplicación trabaje, como el estándar Open Database Connectivity (ODBC) o el estándar Java Database Connectivity (JDBC). El uso de ODBC permite que muchas aplicaciones existentes usen PostgreSQL como base de datos, incluyendo productos de Microsoft Office como Excel o Access.

La arquitectura cliente/servidor de PostgreSQL permite una división de tareas. Un servidor con un buen almacenamiento y acceso a grandes cantidades de información puede usarse como un repositorio seguro de información. Aplicaciones gráficas sofisticadas se pueden desarrollar para clientes. Alternativamente, un aplicación web se puede crear para acceder a la información y devolver los resultados como páginas web, sin aplicaciones adicionales.

Al ser PostgreSQL una herramienta gratuita, posibilita al cliente invertir más recursos económicos en el hardware donde se instalará. Debe existir un balance entre tres componentes del cliente son el procesador, la memoria y los discos.

-Procesador:

Actualmente se pueden distinguir dos tipos, los que tienen 2 o más núcleos en su procesador, y los que sólo tienen un núcleo pero con una frecuencia mayor. Dependiendo de cada base de datos será mejor una opción u otra. Si en la base de datos hay un pequeño número de procesos ejecutándose entonces la mejor opción es elegir procesadores rápidos, que es lo que suele ocurrir cuando se tienen consultas complejas ejecutándose continuamente. Pero si el procesador necesita muchos procesos simultáneos entonces la mejor opción es elegir procesadores de varios núcleos, que es lo que suele ocurrir cuando se tienen aplicaciones con muchos usuarios accediendo.

PostgreSQL no permite dividir una consulta en varios núcleos del procesador (lo que en otros sistemas se llama consulta en paralelo), por lo que si se tiene una consulta muy compleja que se va a ejecutar muchas veces, conviene tener un procesador rápido, y tener varios núcleos no aportará ningún beneficio.

-Memoria:

Priorizar la cantidad de memoria depende de la cantidad de datos que maneja la base de datos sobre la aplicación, o al menos la cantidad de datos que maneja en las operaciones más comunes. Generalmente añadir memoria RAM proporciona una notable mejora del rendimiento, excepto en ciertas situaciones:

- Cuando el conjunto de datos que se consultan es lo suficientemente pequeño como para caber en una RAM de poca capacidad, añadir RAM más no ayudará. En este caso la solución sería posiblemente añadir un procesador más rápido.

- Cuando se ejecutan aplicaciones que buscan en tablas con datos tan grandes que es imposible que quepan sin dividirse en una RAM, como en muchas situaciones de almacenamiento de archivos donde la mejor solución sería conseguir discos más rápidos.

La situación más común es que el conjunto de datos consultado quepa en la capacidad de la RAM pero sin ser demasiado pequeños, por lo añadir memoria beneficia al rendimiento.

-Discos:

Posiblemente el cuello de botella más común sea el disco, especialmente si el sistema tiene sólo uno o dos discos. Hace unos años las dos opciones para los discos duros eran los baratos ATA y los más potentes SCSI, pero hoy en día las dos opciones han avanzado y para un servidor se puede elegir entre SATA (Serial ATA) o SAS (Serial Attached SCSI), ambos con características muy similares.

La última tecnología para discos son los discos sólidos o SSD (Solid State Drives) que proporcionan almacenamiento de memoria permanente y son muy rápidos en la búsqueda de disco. Pero hay tres razones principales por las que no se usan aun para bases de datos:

- La capacidad máxima de almacenamiento es pequeña, y las bases de datos suelen ser grandes.
- El coste por almacenamiento es muy alto.
- La mayoría de diseños no tienen una buena caché de escritura.

ACCESO A LA INFORMACIÓN

PostgreSQL es una base de datos basada en un servidor, como se ha comentado anteriormente, y una vez configurado aceptará solicitudes desde clientes a través de la red, aunque el cliente esté en la misma máquina que el servidor.

Con PostgreSQL se puede acceder a la información de distintas formas:

- Usando una aplicación que interprete sentencias SQL.
- Usando aplicaciones que tengan SQL integrado.
- Usando llamadas a funciones (APIs) que se encargan de preparar y ejecutar sentencias SQL, captar resultados, y montar modificaciones para una gran variedad de lenguajes de programación.
- Accediendo a la información indirectamente usando un estándar como OCBC o JDBC, o usando librerías estándares como DBI (de Perl).

Para usuarios de Windows, está disponible el estándar ODBC (Open DataBase Connectivity), que permite el acceso a bases de datos desde cualquier aplicación. ODBC logra esto al insertar una capa intermedia (CLI) entre la aplicación y PostgreSQL. El propósito de esta capa es traducir las consultas de datos de la aplicación en comandos que PostgreSQL entienda.

-Acceso Multiusuario:

PostgreSQL se encarga automáticamente de que los posibles conflictos entre usuarios no creen errores en la información. Los usuarios tienen acceso a la información como si únicamente ellos estuvieran accediendo a ella, pero realmente PostgreSQL monitoriza los cambios y previene conflictos en las modificaciones.

Esta propiedad de permitir a muchos usuarios leer y escribir simultáneamente mientras se mantiene la consistencia, es una característica muy importante en las bases de datos. Cuando un usuario modifica una fila, otro usuario podrá consultar la fila antes del cambio, y después del cambio, pero nunca la verá a mitad de la modificación. Como se verá más adelante, esto se consigue con el 'aislamiento' de la fila.

CONFIGURACIÓN DEL SISTEMA EN POSTGRESQL

En este punto se pretende explicar el sistema de ficheros de PostgreSQL y las principales opciones de configuración del sistema. El diseño del sistema de archivos es esencialmente el mismo en Windows y en Linux.

Dependiendo del sistema PostgreSQL usa un directorio u otro:

Sistema Operativo	Directorio
Windows **OS X**	C:\Program Files\PostgreSQL\R.r\
Debian **Ubuntu**	/var/lib/postgresql/R.r/main
Red Hat RHEL **CentOS** **Fedora**	/var/lib/pgSQL/

Bajo el directorio base se encuentran entre otros siete subdirectorios importantes (dependiendo también de las opciones seleccionadas en la instalación):

- bin
- data
- doc
- include
- lib
- man (en Linux)
- share
- pgAdmin III (en Windows)

-Directorio 'bin':

Contiene un gran número de archivos ejecutables:

Programa	Descripción
postgres	Servidor interno de base de datos
postmaster	Proceso en espera de base de datos (el mismo ejecutable que 'postgres')
psql	Herramienta de líneas de comandos para PostgreSQL
initdb	Utilidad para inicializar la base de datos
pg_ctl	Control de PostgreSQL (iniciar, parar y reiniciar el servidor)
createuser	Utilidad para crear un usuario de base de datos
dropuser	Utilidad para borrar un usuario de base de datos
createdb	Utilidad para crear una base de datos
dropdb	Utilidad para borrar una base de datos
pg_dum	Utilidad para copia de seguridad una base de datos
pg_dumpall	Utilidad para copia de seguridad de todas las bases de datos en una instalación
pg_restore	Utilidad para restaurar una base de datos a partir de una copia de seguridad
vacuumdb	Utilidad para ayudar a optimizar la base de datos
ipclean	Utilidad para borrar segmentos de memoria compartida después de una caída

pg_config	Utilidad para hacer un informe de la configuración de PostgreSQL
createlang	Utilidad para añadir soporte para extensiones de idiomas
dropland	Utilidad para borrar el soporte de idiomas
ecpg	Compilador SQL integrado

-Directorio 'data':

Contiene subdirectorios con archivos de datos para la instalación base y también los archivos de registro que PostgreSQL usa internamente. También tiene varios archivos de configuración que contienen importantes configuraciones que se pueden modificar. Los archivos accesibles por el usuario más importantes son:

Programa	Descripción
pg_hba.conf	Configura las opciones de autenticación del cliente
pg_ident.conf	Configura el sistema operativo para el mapeo de nombres de autenticación de PostgreSQL cuando se usa autenticación basada en identidades
PG_VERSION	Contiene el número de versión de la instalación
postgresql.conf	Archivo de configuración principal para la instalación de PostgreSQL
postmaster.opts	Proporciona las opciones de línea de comandos por defecto al programa 'postmaster'
postmaster.pid	Contiene el proceso ID del proceso 'postmaster' y una identificación del directorio de datos principal

- Archivo pg_hba.conf: El archivo hba (host based authentication: autenticación basada en host) le dice al servidor PostgreSQL cómo autenticar usuarios, basado en una combinación de su localización, tipo de autenticación, y la base de datos que desea acceder. Un requerimiento típico es querer añadir líneas de configuración para permitir acceso a alguna o varias bases de datos desde máquinas remotas. La configuración por defecto es bastante segura, previniendo el acceso desde cualquier máquina remota. Cada línea en el archivo corresponde con una simple regla de permiso o denegación. Las reglas se procesan en el orden en el que aparecen. Cada línea tiene las siguientes cinco partes:

  - TYPE: Para máquinas locales vale 'local', y para máquinas remotas vale 'host'.

  - DATABASE: Lista separada por comas de las bases de datos en las que se aplica la regla. Si se aplica para todas su valor es 'all'.

  - USER: Lista separada por comas de usuarios para los que se aplica la regla. Si se aplica para todos su valor es 'all', o '+groupname' si se aplica para los usuarios de un grupo.

  - CIDR-ADDRESS: Lista de las direcciones en las que se aplica la regla, a menudo con una máscara de bits.

  - METHOD: Cómo han de autenticarse los usuarios que encajan con las condiciones previas. Sus valores incluyen un amplio rango de valores, entre los más comunes:

Método	Descripción
trust	El usuario tiene acceso, sin necesidad de introducir una clave. Generalmente esta opción sólo se utiliza en sistemas experimentales, aunque es una buena opción cuando la seguridad no es un objetivo.
reject	El usuario es rechazado. Esto puede ser útil para prevenir acceso desde un rango de máquinas remotas, o para modificar alguna condición de una regla más arriba en el archivo.
md5	El usuario debe proporcionar una clave encriptada mediante MD5. Es una buena opción en muchos casos.
crypt	Este método es similar al anterior, pero para instalaciones previas a la 7.2.
password	El usuario debe proporcionar una clave en texto plano. Esta opción no es muy segura, pero es útil cuando se trata de identificar problemas de logeado.
ident	El usuario se autentica usando el nombre de cliente del usuario del sistema operativo del host. Está relacionado con el archivo pg_ident.conf.

Una línea estándar de configuración sería:

TYPE	DATABASE	USER	CIDR-ADDRESS	METHOD
local	all	all	127.0.0.1/32	md5

- Archivo pg_ident.conf:
  Este archivo se usa conjuntamente con la opción 'ident' del archivo 'pg_hba.conf'. Su función es determinar el nombre de usuario en la máquina del cliente, y configurar ese nombre a un nombre de usuario de PostgreSQL. Está basado en el Identification Protocol (estándar RFC 1413). No está considerado un método muy seguro de control de acceso.

- Archivo postgresql.conf:
  Es el principal archivo de configuración que determina cómo funciona PostgreSQL. El archivo consiste en un gran número de líneas, cada una de la forma:

```
nombre_opcion = valor
```

Esto configura el comportamiento de cada opción. Las principales opciones son:

Opción	Valor y significado
listen_addresses	Configura la dirección en la que PostgreSQL acepta conexiones. Normalmente será 'localhost', pero para máquinas con IP múltiples, se puede especificar una dirección IP específica.
port	Configura el puerto en el que PostgreSQL está en espera. Por defecto el puerto es el 5432
max_connections	Configura el número de conexiones concurrentes permitidas. En la mayoría de sistemas operativos es 100. Si se incrementa este número, también se incrementa la sobrecarga de recursos del sistema, y más en particular la cantidad de memoria compartida.
superuser_reserved_connections	Configura el número de conexiones desde el máximo que

	está reservado para superusuarios. Por defecto es 2. Se puede incrementar para asegurar que los superusuarios tienen acceso aunque estén muchos usuarios ordinarios conectados.
authentication_timeout	Define cuánto tiempo un cliente tiene para completar la autenticación antes de ser desconectado automáticamente. Por defecto son 60 segundos. Es aconsejable disminuirlo si muchas personas no autorizadas están intentando acceder.
shared_buffers	Configura el número de buffers que están siendo usados por PostgreSQL. Un valor típico sería 1000. Disminuyendo este valor evita al sistema sobrecarga de recursos. Incrementándolo se mejora el rendimiento en un sistema de producción pesado.
work_men	Señala a PostgreSQL cuánta memoria puede usar antes de incrementar archivos temporales para procesar resultados intermedios. El valor por defecto es 1MB. Si se tienen tablas muy grandes y con muchos datos, incrementar este valor mejorará el rendimiento.
log_destination	Determina dónde PostgreSQL registra los mensajes del servidor, proporcionando una lista separada por comas de nombres de archivos.
log_min_messages	Configura el nivel de mensaje que se registra en el log. Las opciones, desde la que más se registra a la que menos son: 'debug5', 'debug4', 'debug3', 'debug2', 'debug1', 'info', 'notice', 'warning', 'error', 'log', 'fatal', y 'panic'. Por defecto se usa 'notice'.
log_error_verbosity	Configura la cantidad de detalle escrita en el log. El valor por defecto es 'default'. Las otras opciones son 'terse' para reducirlo, y 'verbose' para aumentar la información que se escribe.
log_connections	Registra las conexiones a la base de datos. Es 'false' por defecto, pero en una base de datos segura debería cambiarse por 'true'.
log_disconnections	Registra las desconexiones de la base de datos.
search_path	Controla la orden por la que se busca en los esquemas. El valor por defecto es '$user, public'.
default_transaction_isolation	Configura el nivel de aislamiento de transacciones por defecto. El valor por defecto es 'read commited'.
deadlock_timeout	Configura la cantidad de tiempo antes de que el sistema compruebe interbloqueos cuando está esperando por un bloqueo de una tabla. Por defecto es 1000 milisegundos. Se puede incrementar en un sistema con mucha carga.
statement_timeout	Configura el tiempo máximo, en milisegundos, que una sentencia tiene para ejecutarse. Por defecto es 0, lo que deshabilita esta opción.
stats_start_collector	Si se configura a 'true', PostgreSQL recolecta estadísticas internas, que se podrán usar desde 'pg_stat_activity' y otras vistas estadísticas.

stats_command_string	Si se configura a 'true', permite la recolección de estadísticas de comandos que se están ejecuntado.
datestyle	Configura el estilo de 'date' por defecto. El valor por defecto es 'iso, mdy'.
timezone	Configura la zona horaria por defecto. Por defecto, está configurado como 'unknown', desconocido, lo que significa que PostgreSQL debe usar la zona horaria del sistema.
default_with_oids	Controa si el comando 'CREATE TABLE' crea por defecto tablas con OIDs. Por defecto está a 'true'

- Archivo postmaster.opts:
  Este archivo sirve para configurar las opciones de invocación por defecto para el programa 'postmaster', que es el principal programa de PostgreSQL. Contiene la ruta completa de 'postmaster'. Las opciones que tiene son:

Opción	Descripción
-B nbufs	Configura el número de buffers de memoria compartida a 'nbufs'
-d level	Configura el nivel de información de depuración (entre 1 y 5) que se escribe en el log de registro de servidor
-D dir	Configura el directorio de la base de datos '/data' a 'dir'. No hay valor por defecto, y si esta opción no está configurada se usa el valor de la variable de en torno PGDATA
-i	Permite conexiones remotas TCP/IP a la base de datos
-l	Permite conexiones seguras a base de datos usando el protocolo SSL (Secure Sockets Layer). Requiere la opción '-i' (acceso a red) y soporte a SSL para que sea compilado en el servidor
-N cons	Configura el número máximo de conexiones simultáneas que el servidor acepta
-p port	Configura el número de puerto de TCP que usará el servidor para esperar respuesta
--help	Obtiene una lista de opciones útiles

Desde Windows, un típico archivo sería (en una única línea):

```
C:/Program Files/PostgreSQL/8.0.0/bin/postmaster.exe "-D"
        "C:/Program Files/PostgreSQL/8.0.0/data"
```

-Otros subdirectorios:

- Directorio 'doc': Contiene la documentación online, y puede contener documentación adicional.

- Directorios 'include' y 'lib': Contienen la cabecera y librerías necesarias para crear y ejecutar aplicaciones cliente para PostgreSQL.

- Directorio 'main': En Linux únicamente, contiene el manual.

- Directorio 'share': Contiene una mezcla de ficheros de ejemplos de configuración, material adicional, y archivos de zonas horarias. También contiene un listado de características del estándar SQL soportados por la versión actual de PostgreSQL.

INICIALIZACIÓN DE LA BASE DE DATOS

Después de una instalación de PostgreSQL, se tiene que montar la base de datos a crear. Es importante inicializar la base de datos de PostgreSQL correctamente, ya que la seguridad de la base de datos se aplica a partir de los permisos de usuarios sobre los directorios de datos. Se necesitan los siguientes pasos para asegurarse que la base de datos es segura:

- Crear un usuario dueño de la base de datos, por defecto es el usuario 'postgres'.
- Crear el directorio ('data') para almacenar los archivos de la base de datos.
- Asegurar que el usuario 'postgres' es el propietario del directorio.
- Ejecutar 'initdb', con el usuario 'postgres' para inicializar la base de datos.

Lo normal es que estos pasos se ejecuten automáticamente durante la instalación, pero si se necesita hacer algo de forma diferente, o instalarlo manualmente, se han de seguir los pasos. La utilidad 'initdb' tiene varias opciones, las más comunes son:

Opción	Descripción
-D dir, --gdata=dir	Especifica la localización del directorio de datos para la base de datos
-W, --pwprompt	Consigue que 'initdb' solicite una contraseña para el superusuario de base de datos. Se requiere una contraseña para habilitar la autenticación por contraseña.

La instalación de base de datos creada por defecto por 'initdb' contiene información sobre la cuenta del superusuario, una plantilla llamada 'template1', y otros objetos de la base de datos. Esta plantilla inicial es importante ya que se usa como la plantilla para las siguientes creaciones de bases de datos.

Para crear bases de datos adicionales, hay que conectarse al sistema de base de datos y solicitar la creación de una nueva base de datos. Se puede usar la utilidad 'createdb', o más comúnmente, desde dentro de la base de datos una vez iniciada la sesión. Una conexión requiere un nombre de usuario, posiblemente con contraseña, y un nombre para la base de datos. En la instalación inicial, sólo se tiene un usuario ('postgres') y se puede conectar con una única base de datos. Antes de poder conectarse al sistema de base de datos, el proceso en servidor debe estar ejecutándose, como se especifica en el siguiente apartado.

CONTROL DE SERVIDOR

El servidor de base de datos de PostgreSQL se ejecuta como un proceso en escucha en UNIX y Linux, y como un servicio de sistema en Windows. El proceso de servidor se llama 'postmaster'

y debe ejecutarse para que las aplicaciones del cliente puedan conectarse y utilizar la base de datos.

Si se desea, se puede inicializar el proceso automáticamente en Linux. Pero en Windows se tiene que usar la aplicación 'Servicios' del panel de control, seleccionar el servicio 'PostgreSQL Database' e inicializarlo.

CONFIGURACIÓN INTERNA DE POSTGRESQL

-Métodos de configuración:

Generalmente la configuración se tiene que realizar con un usuario administrador, que es 'postgres' por defecto. Hay tres formas de configurar objetos internamente en PostgreSQL:

- Comandos SQL: Usando SQL, con un gran número de sentencias dedicadas a mantener la configuración. Muchos de éstos son sentencias estándares SQL (DDL: Data Definition Language), utilizable en un amplio rango de bases de datos, pero la mayoría de bases de datos tienen sus propios elementos SQL propietarios. Manejar bien SQL para configurar las bases de datos es importante ya que ayuda a entender qué está ocurriendo exactamente. También es importante conocerlas porque las herramientas gráficas pueden no existir, o el ancho de banda o conexión disponibles son limitados.

- Herramientas gráficas: Usando entre otros pgAdmin III, la principal herramienta gráfica para PostgreSQL, muy sencilla de utilizar.

- Versiones de línea de comandos: Algunas opciones de configuración, como son las que crean usuarios y bases de datos, tienen disponible una versión de línea de comandos. Aunque pueden ser prácticas, no se suele usar para configurar PostgreSQL

-Configuración de usuario:

Es importante crear una estructura de cuentas de usuario para poder manejar los cambios de personal y los cambios de roles. Los usuarios se manejan con los comandos CREATE USER, ALTER USER y DROP USER:

- Crear usuario:

```
CREATE USER nombre_usuario
        [ WITH
        | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'contrasenya'
        | CREATEDB | NOCREATEDB
        | CREATEUSER | NOCREATEUSER
        | IN GROUP nombre_grupo [, ...]
        | VALID UNTIL 'fecha' ]
```

Generalmente se le da una contraseña al usuario. Con la opción CREATEUSER el usuario será opciones de administrador, pudiendo crear otros usuarios. Con la opción CREATEBD el usuario podrá crear bases de datos. Con la opción IN GROUP se podrá asignar al usuario uno o más grupos. Con la opción VALID UNTIL se expresa que la cuenta de usuario expirará en la fecha señalada.

- Utilidad 'createuser':

```
createuser [ opciones... ] username
```

Es una utilidad que permite la creación de usuarios en PostgreSQL mediante la línea de comandos del sistema operativo.

Las opciones permiten especificar el servidor de base de datos para el que se quiere crear un usuario y para configurar privilegios de usuario. Las opciones son:

Opción	Descripción
-h nombre_host **--host nombre_host**	Especifica el host del servidor de base de datos. Por defecto la máquina local
-p puerto **--port puerto**	Especifica el puerto. Por defecto el puerto de escucha estándar de PostgreSQL: 5432
-U usuario **--username=usuario**	Especifica el usuario que va a conectarse al servidor
-q **--quiet**	No imprime respuesta
-d **--createdb**	Permite al usuario crear bases de datos
-a **--adduser**	Permite al usuario crear nuevos usuarios
-P **--pwprompt**	Solicita una contraseña que le asigna al nuevo usuario
-i **--sysid=numero_ID**	Especifica el número ID del usuario. Generalmente no se usa esta opción para que se le asigne un valor por defecto
-e **--echo**	Imprime el comando enviado al servidor para crear el usuario
--help	Imprime un mensaje de uso

- Modificar usuario:

```
ALTER USER nombre_usuario
        [ WITH
        | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'contrasenya'
        | CREATEDB | NOCREATEDB
        | CREATEUSER | NOCREATEUSER
        | VALID UNTIL 'fecha' ]
```

Este comando tiene las mismas opciones que CREATE USER, y tiene una variante para renombrar usuarios:

```
ALTER USER nombre_usuario RENAME TO nuevo_nombre_usuario
```

- Listado de usuarios: Se pueden consultar los usuarios en la base de datos usando una vista del sistema llamado 'pg_user':

```
SELECT usesysid, usename, usecreatedb, usesuper, valuntil
FROM pg_user;
```

- Eliminación de usuarios:

```
DROP USER nombre_usuario;
```

Este comando es sencillo y sólo necesita el nombre del usuario. Una alternativa en línea de comandos sería 'dropuser':

```
dropuser [ opciones ... ] nombre_usuario
```

Las opciones de 'dropuser' son las opciones de conexión de servidor que tiene también 'createuser', más la opción '-i' para solicitar confirmación de borrado.

- Gestión de usuarios con pgAdmin III: Todas las opciones anteriores se pueden realizar con pgAdmin III. Con el botón derecho sobre la parte 'Usuarios' se puede crear un usuario, modificar uno seleccionado, o borrar uno seleccionado.

-Configuración de grupos:

Los grupos son una ventaja para la configuración, una forma útil de agrupar usuarios para propósitos administrativos. Igual que en el caso de usuarios, existen comandos para gestionarlos, y se puede usar pgAdmin III para lo mismo.

- Creando grupos:

```
CREATE GROUP nombre_grupo [ WITH USER usuarios_separados_por_comas ]
```

- Modificando grupos:

```
ALTER GROUP nombre_grupo ADD USER nombre_usuario
ALTER GROUP nombre_grupo DROP USER nombre_usuario
ALTER GROUP nombre_grupo RENAME TO nuevo_nombre_grupo
```

- Listado de grupos: con la utilidad 'pg_group':

```
SELECT * FROM pg_group;
```

- Eliminación de grupos:

```
DROP GROUP nombre_grupo
```

-Configuración de Tablespaces:

Una de las características principales de administración introducidas en PostgreSQL es el concepto de tablespaces. Éstos facilitan a los administradores de la base de datos controlar cómo los datos se almacenan en las tablas a través del sistema de archivos, que es útil para tareas como gestionar tablas grandes y mejorar el rendimiento distribuyendo la carga en distintas unidades de disco.

Un tablespace es un objeto de PostgreSQL que corresponde con una localización física en el sistema operativo del host. Los tablespaces sólo pueden ser creados por usuarios administrativos con los privilegios de CREATE USER.

Antes de crear un tablespace, hay que crear una ubicación del disco físico en el que mapear el tablespace.

- Creación de tablespaces:
  Primero se tiene que crear la ubicación y asignarle como propietario del directorio el mismo que el sistema operativo tenía para la instalación de PostgreSQL (‘postgres’). Después ya se puede crear un tablespace PostgreSQL asociado con el directorio. Se tiene que realizar mediante el programa ‘psql’. Los directorios a asociar deben estar vacíos antes de la asociación. El comando para la creación de tablespaces es simple:

```
CREATE TABLESPACE nombre_tablespace [OWNER nombre_propietario] LOCATION 'directorio'
```

  Si no se especifica propietario, toma como valor por defecto el de la persona que ejecuta el comando. Se puede consultar los tablespaces con la vista ‘pg_tablespace’:

```
SELECT * FROM pg_tablespace;
```

- Modificación de tablespaces: No es posible modificar la localización física, únicamente el propietario y el nombre:

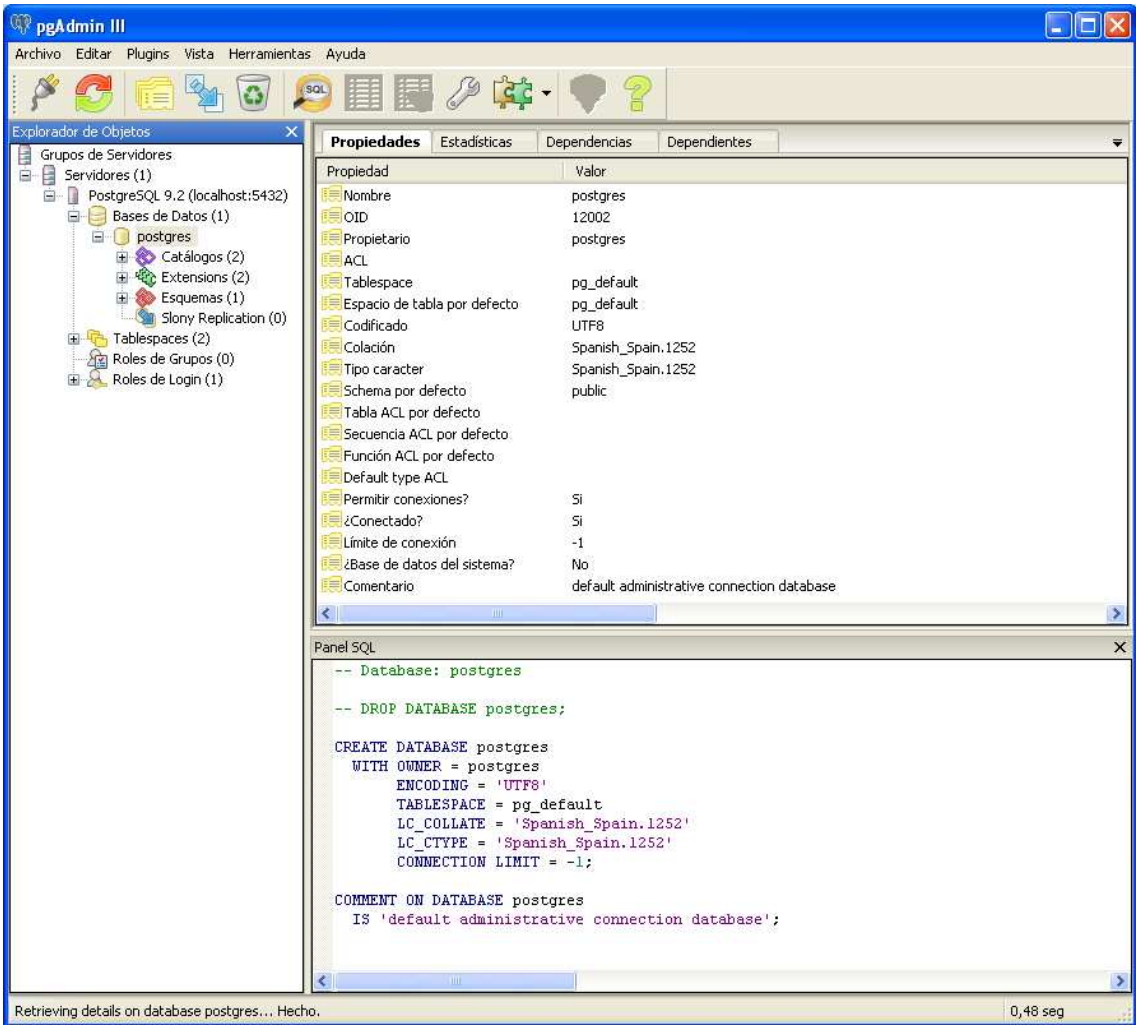
```
ALTER TABLESPACE nombre_tablespace OWNER TO nuevo_propietario
ALTER TABLESPACE nombre_tablespace RENAME TO nuevo_nombre_tablespace
```

- Eliminación de tablespaces: Se permite eliminar el tablespace siempre que se eliminen previamente los objetos que se encuentran en él.

```
DROP TABLESPACE nombre_tablespace
```

-Gestión de bases de datos:

Los elementos clave en cualquier instalación de bases de datos son las propias bases de datos, es decir, los objetos en los que se almacenan las tablas y los datos. Otros sistemas de bases de datos gestionan las bases de datos de forma muy distinta, pero PostgreSQL añade herramientas para hacerlo de forma simple. Cada instalación del servidor de PostgreSQL puede manejar y dar servicio muchas bases de datos individuales. La instalación incluye tablespaces, nombres de usuarios, y grupos.



- Creación de bases de datos:

```
CREATE DATABASE nombre_bd
        [ [ WITH ] [ OWNER [=] propietario ]
        [ TEMPLATE [=] plantilla ]
        [ ENCODING [=] codificación ]
        [ TABLESPACE [=] tablespace ] ]
```

La base de datos debe ser única en la instalación. El OWNER permite crear una base de datos con un propietario. La opción TABLESPACE permite especificar cuál de los tablespaces se usará para almacenar los datos, si no se especifica los archivos usarán el tablespace por defecto 'pg_default', creado automáticamente en la instalación. Las opciones TEMPLATE y ENCODING especifican el diseño de la base de datos y la codificación multibyte escogida. Estas opciones se suelen omitir.

- Modificación y listado de bases de datos: Se puedes modificar el nombre y el propietario de una base de datos:

```
ALTER DATABASE nombre_bd RENAME TO nuevo_nombre_bd
```

```
ALTER DATABASE nombre_bd OWNER TO nuevo_propietario
```

- Eliminación de bases de datos:

```
DROP DATABASE nombre_bd
```

No se permite eliminar una base de datos con conexiones abiertas, teniendo que acceder desde otra base de datos.

- Creación y eliminación de bases de datos desde la línea de comandos:

```
createdb [ opciones... ] nombre_bd [ descripción ]
dropdb [ opciones... ] nombre_bd
```

Las opciones para estas dos utilidades son similares a las de 'createuser' y 'dropuser':

Opción	Descripción
-h **--host= nombre_host**	Especifica el host del servidor de base de datos o directorio del socket
-p **--port=puerto**	Especifica el puerto del servidor
-U **--username=usuario**	Especifica el usuario que va a conectarse
-W **--password**	Solicita una contraseña
-D **tablespace=tablespace_elegido**	Configura el tablespace por defecto para la nueva base de datos
-E **--enconding=codificación**	Configura la codificación para la nueva base de datos
-O **--owner=propietario**	Especifica el usuario de base de datos que será propietario de la nueva base de datos
-T **template=plantilla**	Especifica la plantilla de base de datos para copiarlo para la nueva base de datos
-e **--echo**	Imprime el comando enviado al servidor
-q **--quiet**	No imprime respuesta
--help	Imprime un mensaje de uso
--version	Imprime información de la versión, y después sale

-Gestión de esquemas:

Dentro de cada base de datos hay un nivel por encima del nivel de las tablas: el esquema, que es un agrupado de objetos relacionados de la base de datos. Por defecto PostgreSQL crea un esquema llamado 'public' y coloca las tablas dentro de él. Para la mayoría de los usos no hace falta crear otros esquemas, y con utilizar 'public' es suficiente. Los esquemas tienen dos propósitos:

- Ayudar a gestionar el acceso a distintos usuarios a una única base de datos.
- Permitir tablas adicionales que están asociadas con la base de datos estándar, pero manteniéndolas separadas.

- Creación de esquemas:

```
CREATE SCHEMA nombre_esquema [ AUTHORIZATION propieario_esquema ]
```

Es necesario estar conectado a la base de datos en la que se desea crear el esquema. Aunque el comando de creación no tiene opciones, es posible añadirle comentarios:

```
COMMENT ON SCHEMA nombre_esquema IS 'texto de ayuda'
```

- Listado de esquemas:
  Con pgAdmin III se ves de forma clara los esquemas, pero se puede usar el comando '\dn' en 'psql'. Este listado mostrará también los esquemas internos de PostgreSQL de 'pg_catalogue' y 'pg_toast'.

- Eliminación de esquemas:

```
DROP SCHEMA nombre_esquema [CASCADE]
```

La eliminación de un esquema con el comando anterior permite la opción CASCADE, que le informa a PostgreSQL que borre todos los objetos que están dentro del esquema. En general es más seguro eliminar las tablas primero, y después eliminar el esquema vacío, para evitar un borrado accidental de un esquema.

- Creación de tablas en un esquema: Para crear una tabla en un esquema, sólo se ha de usar como prefijo de la tabla el nombre del esquema:

```
CREATE TABLE nombre_esquema.nombre_tabla
(
        column definiciones
);
```

- Configuración de la ruta de búsqueda del esquema: Es posible controlar la forma en que PostgreSQL busca los diferentes nombres de esquemas especificando el 'search_path' del esquema:

```
SHOW ruta_búsqueda;
SET ruta_búsqueda TO esquema, public;
```

- Listado de tablas en un esquema: No hay un comando específico en 'psql' para listar las tablas de un esquema, pero es posible consultar sobre el catálogo del sistema 'pg_tables':

```
SELECT schemaname, tablename, tableowner
FROM pg_tables
WHERE schemaname = 'schema1';
```

-Gestión de privilegios:

PostgreSQL controla los accesos a la base de datos usando un sistema de privilegios que se conceden y revocan con los comandos GRANT Y REVOKE respectivamente, y también desde pgAdmin III. Por defecto los usuarios no pueden guardar datos en tablas que no han creado. El comando para crearlo es:

```
GRANT privilegio [, ...] ON objeto [, ...]
TO { PUBLIC | GROUP nombre_grupo | nombre_usuario } [ WITH GRANT OPTION ]
```

La opción WITH GRANT OPTION permite al usuario con privilegios conceder esos privilegios a otros. El objeto debe ser el nombre de una tabla, una vista, un tablespace o un grupo. La opción PUBLIC es para cuando se quiere asignar a todos los usuarios. Los posibles privilegios son:

Privilegio	Descripción
SELECT	Permite leer filas
INSERT	Permite crear nuevas filas
DELETE	Permite borrar filas
UPDATE	Permite modificar filas ya existentes
RULE	Permite crear reglas para una tabla o vista
REFERENCES	Permite crear de restricciones de claves ajenas (debe tener permiso también en la tabla relacionada)
TRIGGER	Permite crear triggers en una tabla
EXCECUTE	Permite ejecutar procedimientos almacenados
ALL	Concede todos los privilegios

Y el comando para la eliminación del privilegio es:

```
REVOKE privilegio [, ...]
ON objeto [, ...]
FROM { PUBLIC | GROUP nombre_grupo | nombre_usuario }
```

REPRESENTACIÓN DEL MODELO RELACIONAL

En el diseño de una base de datos existen ciertos patrones que ocurren una y otra vez. Es útil reconocer estos patrones porque generalmente se tratan de la misma forma.

-Muchos-a-muchos:

Cuando se tienen dos entidades que aparentemente tienen una relación de muchos a muchos entre ellos, se debe romper de alguna forma esta relación, ya que no es correcta esta relación en la base de datos física.

La solución es crear una tabla adicional de enlace entre las dos tablas, que tengan la información relacionada. En un ejemplo de una tabla ‘autor’ y otra ‘libro’, cada autor tiene muchos libros, y cada libro tiene uno o más autores. La solución sería es insertar una tabla entre las dos llamada ‘libro_autor’ con la información de la relación entre libros y autores. De

esta forma cada autor tiene una relación uno-a-muchos con la tabla 'libro_autor', y lo mismo pasa con cada libro.

-Jerarquía:

Otro patrón que se repite es la jerarquía, que puede aparecer de distintas formas. En un ejemplo de una tabla 'tienda' en la que tiene un atributo 'país' y otro 'región', y además existen las tablas 'país' y 'región'. Almacenar en la tienda tanto el país como la región está violando la forma normal tercera, ya que la región depende del país y no únicamente de la clave primaria de la tienda. La solución sería que la tienda únicamente tuviera información de la región, y que en la tabla 'región' se añadiese una columna 'país' relacionada con la tabla 'país'.

-Relaciones recursivas:

Este patrón no es tan común como los anteriores, pero ocurre cuando una tabla tiene una relación consigo misma. Un ejemplo es una tabla de 'empleado' donde se refleja la jerarquía de una empresa, y por tanto cada empleado tiene un atributo que lo relaciona con su jefe, y a su vez ese jefe tiene otro jefe. Una solución es crear un atributo 'jefe' que haga referencia a la clave primaria del jefe en la misma tabla. En este ejemplo tendrá que haber uno o más casos donde el atributo 'jefe' esté a NULL (para el presidente de la empresa)

BIBLIOGRAFÍA

- [2] "Beggining Databases with PostgreSQL: From Novice to Professional, Second Edition"
- [4] http://lema.rae.es/drae/?val=base+de+datos
- [5] http://es.wikipedia.org/wiki/Open_Database_Connectivity
- [6] "PostgreSQL 9.0 High Performance"
- [7] "PostgreSQL 9 Administration Cookbook"

TEMA 2 – TRANSACCIONES

LENGUAJES DE CONSULTA

Los sistemas de gestión de bases de datos relacionales sirven para añadir y actualizar información, pero su función más potente es permitir a los usuarios preguntar información sobre la información almacenada, mediante consultas (queries). En estos sistemas las relaciones definen conjuntos, y estos conjuntos pueden ser gestionados matemáticamente. Las consultas usan una rama de la lógica teórica llamada lógica de predicados, y es lo que los lenguajes de consulta utilizan como base. Los sistemas modernos de bases de datos, entre ellos PostgreSQL, ocultan toda esta parte matemática detrás de un sencillo lenguaje de consulta.

SQL

SQL son las siglas de Structured Query Language (lenguaje de consultas estructurado), un lenguaje declarativo desarrollado por IBM, es el sistema más utilizado para comunicarse con los servidores de bases de datos, y está considerado el estándar para lenguajes de consulta. Los orígenes del SQL están ligados a los de las bases de datos relacionales. En 1970 E. F. Codd propone el modelo relacional y asociado a éste un sublenguaje de acceso a los datos basado en el cálculo de predicados. Basándose en estas ideas, los laboratorios de IBM definen el lenguaje SEQUEL (Structured English Query Language) que más tarde sería ampliamente implementado por el sistema de gestión de bases de datos experimental System R, desarrollado en 1977 por IBM. Sin embargo, fue Oracle quien lo introdujo por primera vez en 1979 en un programa comercial. SEQUEL terminaría siendo el predecesor de SQL, siendo este una versión evolucionada del primero. El SQL pasa a ser el lenguaje más usado en los diversos sistemas de gestión de bases de datos relacionales surgidos en los años siguientes, y por eso es estandarizado en 1986 por el ANSI, dando lugar a la primera versión estándar de este lenguaje, el "SQL-86" o "SQL1". Al año siguiente este estándar es también adoptado por la ISO. Sin embargo, este primer estándar no cubre todas las necesidades de los desarrolladores e incluye funcionalidades de definición de almacenamiento que se consideró suprimirlas. Por eso en 1992 se lanzó un nuevo estándar ampliado y revisado del SQL llamado "SQL-92" o "SQL2". En la actualidad el SQL es el estándar de facto de la inmensa mayoría de los SGBD comerciales, aunque las distintas implementaciones del lenguaje es amplia, el soporte al estándar SQL-92 es general y muy amplio. La última versión es el SQL:2008, del año 2008.

El lenguaje SQL comprende tres tipos de comandos:

- Lenguaje de manipulación de datos (DML: Data Manipulation Language): Es la parte de SQL más usada, y está formada por los comandos de inserción, borrado, actualización y selección de información de la base de datos.

- Lenguaje de definición de datos (DDL: Data Definition Language): Son los comandos para crear tablas, definir relaciones, y controlar otros aspectos de la base de datos más estructurales.

- Lenguaje de control de datos (DCL: Data Control Language): Es un conjunto de comandos que generalmente controlan los permisos sobre la información, tal como la definición de derechos de acceso. La mayoría de usuarios de bases de datos nunca llegan a usar estos comandos.

ESTRUCTURA DE SQL

No es objetivo de este proyecto tratar en profundidad el lenguaje SQL, pero a continuación se dan ejemplos de las distintas funcionalidades:

-Crear tablas:

Indicando el nombre de la tabla, las columnas y el tipo de dato (y longitud máxima) de cada columna.

```
CREATE TABLE cliente
(
        id_cliente INTEGER,
        nombre CHAR(30) NOT NULL,
        teléfono CHAR(20),
        dirección CHAR(40),
        código_postal INTEGER,
        país CHAR(20)
);
```

La tabla necesita un identificador, que actúa como clave primaria (PK), y que puede ser generado automáticamente. En este caso es id_cliente, de tipo INTEGER, por lo que cada cliente tendrá un número entero que lo identifique, y que este número será independiente del cliente. El valor 'nombre' será un VARCHAR (cadena alfanumérica) con límite 30 caracteres, y que no podrá ser nulo.

-Añadir información:

Indicando todos o algunos de los datos de la tabla para insertar nuevas filas.

```
INSERT INTO cliente VALUES
(1,'Miguel', '686323582', 'Pl. Reina 3, Valencia', 46002, 'España');
```

Cada valor deberá tener el tipo de dato que corresponde con su columna, y los alfanuméricos deberán estar limitados por comillas simples (').

-Ver información:

Indicando qué información se desea mostrar, de qué tablas, y con qué condiciones.

```
SELECT * FROM cliente WHERE país = 'España';
```

Debido a la gran funcionalidad de este punto, está desarrollado más ampliamente en el apartado siguiente.

-Borrar información:

Seleccionando la fila o las filas que se quiere eliminar de la tabla.

```
DELETE FROM cliente WHERE nombre = 'Luis';
```

-Modificar información:

Seleccionando la fila o filas a modificar, y señalando la nueva información.

```
UPDATE amigo SET edad = 42 WHERE nombre = 'Samuel' AND apellido = 'Jiménez';
```

-Eliminar tablas:

Señalando únicamente el nombre de la tabla.

```
DROP TABLE amigo;
```

ACCEDIENDO A LA INFORMACIÓN

En este apartado se explica un poco más profundamente la codificación de SQL para hacer consultas sobre la base de datos. Partiendo de la base de datos de ejemplo:

cliente		**empleado**		**artículo**		**compra**	
id_cliente	serial	id_ empleado	serial	id_ articulo	serial	id_ cliente	integer
nombre	varchar	nombre	varchar	nombre	varchar	id_ empleado	integer
teléfono	varchar	fecha_ contrato	date	precio	numeric	id_ articulo	integer
dirección	varchar			peso	float	fecha_ compra	date
código_postal	integer					pago	numeric
país	varchar						

-Ver información de una tabla:

Cualquier base de datos relacional se basa en SQL para obtener la información, usando su sentencia SELECT. El esquema simple de una sentencia SELECT es:

```
SELECT <lista de columnas> FROM <tabla>
```

Hay que distinguir dos formas de hacerlo:

- Selección:

Para seleccionar un subconjunto de filas. En el ejemplo podría ser una consulta de clientes que viven en España, a lo que el sistema devolverá todas las filas de la tabla cliente donde la columna 'país' tenga como valor la cadena 'España'.

```
SELECT * FROM cliente WHERE país = 'España';
```

- Proyección:
  Para seleccionar ciertas columnas de una tabla. En el ejemplo podría ser una consulta para conocer el nombre y teléfono de todos los clientes, sin importar la demás información, a lo que el sistema devolverá una lista con la información solicitada.

```
SELECT nombre, teléfono FROM cliente;
```

Evidentemente se puede hacer una consulta juntando selección y proyección, además de añadir restricciones dentro del WHERE hasta hacer la selección una sentencia bastante compleja.

-Ver información de varias tablas:

Una de las mejoras que proporciona el lenguaje SQL es poder obtener información conjunta de varias tablas. En el ejemplo, se puede tener la tabla cliente, y otra tabla de compras:

cliente					
Id_cliente	Nombre	Teléfono	Dirección	Código_postal	País
1	Miguel	686323582	Pl. Reina 3, Valencia	46002	España
2	Laia	638258412	C. Alicante 14-8, Aldaia	46121	España

compra					
Id_compra	Id_cliente	Id_empleado	Id_articulo	Fecha_compra	Pago
1	1	3	32	01/02/2012	100
2	1	2	9	28/05/2012	350

Para poder ver la información del cliente en una compra dada, se ve el valor que tiene en el campo 'id_cliente' de la tabla, y se compara con el valor 'id_cliente' de la tabla cliente. A esto se le llama combinación de tablas.

```
SELECT * FROM cliente, compra WHERE cliente.id_cliente = compra.id_cliente;
```

Como en el ejemplo se ha puesto que se quiere ver todas las columnas (*), el resultado devolverá todas las columnas de las dos tablas. Además, como la columna para identificar el cliente se llama igual en las dos tablas, hay que especificar a qué campo se refiere en cada caso con 'cliente.id_cliente' y 'compra.id_cliente'.

-Controlar el orden:

En la base de datos no existe un orden en el que se almacenan las filas, y cada sistema usa un método para almacenaros. Generalmente los datos se muestran en el orden interno en el que están almacenados. Para señalar el orden en que se muestran los resultados de una SELECT,

hay que incorporar las palabras ORDER BY al final de la sentencia, y a continuación la columna o columnas que marcan la ordenación de la consulta, y la palabra ASC si se quiere un orden ascendente, o DESC si se quiere un orden descendiente. Si no se especifica el tipo de orden, se ordena ascendentemente.

SELECT <columnas> FROM <tabla> ORDER BY <columna> [ASC | DESC]
SELECT * FROM cliente WHERE país = 'España' ORDER BY nombre ASC;

-Evitar información repetida:

Aunque las filas no deben repetirse, en ocasiones en una SELECT se obtiene sólo cierta información de la tabla, y esa información parcial puede estar repetida. En el ejemplo, si se quiere obtener los países de los clientes con la SELECT siguiente, se obtiene una fila por cada cliente, aunque el país coincida en varios clientes.

SELECT país FROM cliente;

En estos casos, hay que especificar que no se quiere información repetida con la palabra DISTINCT.

SELECT DISTINCT país FROM cliente;

-Cálculos numéricos:

A veces la información que se necesita es una variación matemática o un cálculo de distintas columnas. Es posible introducir algunos cálculos en la SELECT. En el ejemplo, si se quiere calcular el IVA en el campo 'pago' de la tabla 'compra':

SELECT id_articulo, pago + (pago * 0.21) AS precio_iva FROM compra;

En este caso, además se especifica que la columna calculada se llame 'precio_iva', para que quede claro visualmente, y sin ningún efecto sobre la base de datos.

-Funciones agregadas:

Son funciones para producir estadísticas a partir de los resultados de una consulta, como obtener el mayor valor en una columna, el recuento de valores,…. Estas funciones son:

Función	Descripción
Count(*)	Devuelve el recuento de filas
Count(columna)	Devuelve el recuento de filas donde el valor de la columna no es NULL
Min(columna)	Devuelve el mínimo valor encontrado en la columna
Max(columna)	Devuelve el máximo valor encontrado en la columna
Sum(columna)	Devuelve la suma total de las entradas en la columna
Avg(columna)	Devuelve la media de las entradas en la columna

-Condicionando la búsqueda:

Con una SELECT se obtienen todas las filas de la tabla, pero si se quiere obtener únicamente las filas que cumplan una condición se tiene que incorporar la palabra WHERE:

SELECT <columnas> FROM <tabla> WHERE <condiciones>

```
SELECT * FROM cliente WHERE país = 'España';
```

La parte de condiciones que siguen al WHERE pueden separarse por la palaba AND o OR, según se quiera una consulta que cumpla todas las condiciones, o alguna de ellas, respectivamente. Para tratar los NULL, el valor 'desconocido' explicado en el tema anterior, ya que se trata de un tipo especial, no se podrá poner una condición "WHERE país = NULL", si no que habrá que usar la funcionalidad especial "WHERE país IS NULL", o en su negación "WHERE país IS NOT NULL".

-Agrupando la búsqueda:

Cuando se quiere obtener información agrupando los valores de alguna columna, por ejemplo, los clientes de cada país. Para esto hace falta la cláusula GROUP BY:

```
SELECT país, count(*) FROM cliente GROUP BY país;
```

-Subconsultas:

Cuando uno o más condiciones dentro de un WHERE son otras consultas enteras. Por ejemplo, si se quiere consultar los artículos que valen más que la media, la media se obtiene con:

```
SELECT avg(pago) FROM compra;
```

Y metiendo esta consulta como subconsulta:

```
SELECT * FROM articulo WHERE precio > (SELECT avg(precio) FROM articulo);
```

FORMACIÓN DE TABLAS

Ya se ha comentado en un apartado anterior cómo se forman las tablas de una forma sencilla mediante el comando CREATE TABLE, pero en este apartado se pretende desarrollar más profundamente cómo funciona, además de poder explicar funciones adicionales. La sintaxis básica para crear tablas es:

```
CREATE [TEMPORARY] TABLE nombre_tabla (
        { nombre_columna tipo [ restricción_columna ] [,...] }
        [ CONSTRAINT restricción_tabla ]
) [ INHERITS (nombre_tabla_existente) ]
```

En la primera línea se especifica el nombre de la tabla, y la palabra opcional TEMPORARY permite crear una tabla temporal. En las siguientes líneas se describen la lista de columnas de la tabla, con su nombre, su tipo, y una restricción opcional. Se pueden añadir ilimitadas columnas a una tabla, separadas por comas. La restricción opcional permite especificar reglas adicionales como por ejemplo NOT NULL para no permitir un valor desconocido. Después de la lista de columnas va una restricción de tabla opcional, que permite escribir reglas adicionales para la tabla.

Finalmente va una extensión opcional, INHERITS, que da la posibilidad que la nueva tabla creada herede las columnas de una tabla existente. La nueva tabla contiene todas las columnas que están en la tabla de la que hereda, además de las columnas especificadas como propias.

-Restricciones de columna:

Es común tener columnas con ciertas reglas. A veces se quiere imponer reglas que manejen los datos, como por ejemplo que un pago no sea inferior a 10 euros, o asegurarse que un campo es único. Al añadir estas restricciones se posibilita un diseño de comprobaciones de datos a bajo nivel para toda la aplicación. Además añade seguridad a la aplicación, ya que es independiente de la aplicación final y un uso ilegal de la misma no permitirá insertar valores no deseados en la base de datos. Las principales restricciones de columna son:

Restricción	Descripción
NOT NULL	La columna no puede tomar valor NULL
UNIQUE	El valor almacenado es diferente en cada fila, excepto el valor NULL que puede estar en varias filas
PRIMARY KEY	Es una combinación de NOT NULL y UNIQUE. Cada tabla sólo puede tener sólo una columna con PRIMARY KEY. Si se necesita una clave primaria con más de una columna, hay que crearla como restricción de la tabla
DEFAULT valor_por_defecto	Permite proporcionar un valor por defecto cuando se insertan datos
CHECK (condición)	Permite comprobar una condición cuando se insertan o actualizan datos
REFERENCES	Restringe el conjunto de valores posibles a los de la columna de otra tabla

A excepción de PRIMARY KEY, se pueden incluir todas las restricciones que hagan falta en cada columna. La restricción UNIQUE no sigue el estándar SQL, ya que en PostgreSQL se permite insertar el valor NULL en varias filas, pero el estándar SQL sólo permitiría uno.

-Restricciones de tabla:

De forma muy similar a las restricciones de columna, pero aplicadas a la tabla entera. A veces hace falta especificar restricciones, como la clave primaria, en este nivel. Las cuatro restricciones de tabla posibles son:

Restricción	Descripción
UNIQUE (lista_de_columnas)	El conjunto de valores almacenado en las columnas listadas debe ser diferente de los demás en la tabla
PRIMARY KEY (lista_de_columnas)	Es una combinación de NOT NULL y UNIQUE. Cada tabla sólo puede tener una clave primaria, bien de columna, bien de tabla
CHECK (condición)	Permite comprobar una condición cuando se inserta o se actualizan datos
REFERENCES	Restringe el conjunto de valores posibles a los de las columnas de otra tabla

Como se ve, las restricciones de tabla son parecidas a las restricciones de columna. Las diferencias son:

- Las restricciones de tabla se pueden referir a más de una columna.
- Las restricciones de tabla se listan después de listar todas las columnas.

-Modificar la estructura de una tabla:

Aunque lo óptimo es diseñar correctamente una base de datos, y una vez haya datos no modificar el diseño, a menudo los requerimientos y la implementación de la base de datos pueden cambiar, y necesitar de una modificación del diseño de una tabla. Una solución sería:

- Crear una nueva tabla intermedia con la misma estructura que la tabla vieja.
- Usar INSERT INTO para copiar toda la información de la tabla vieja a la intermedia.
- Eliminar (usando DROP) la tabla vieja.
- Crear una tabla nueva con el nombre de la tabla vieja, pero con el nuevo diseño.
- Volver a usar INSERT INTO desde la tabla intermedia para que la tabla nueva tenga todos los datos.
- Eliminar (usando DROP) la tabla intermedia.

Estos pasos realizarían una solución correcta al problema, pero haría falta copiar datos (en ciertos casos podrían ser millones de filas) entre tablas, lo que podría ser lento (más si existiesen triggers o vistas). Por eso, PostgreSQL permite añadir, eliminar, borrar y renombrar columnas en tablas ya existentes y con datos, o incluso renombrar la propia tabla. Igualmente es posible también añadir o eliminar restricciones, aunque evidentemente estos cambios deberán cumplir ciertos requisitos prácticos, por ejemplo al añadir una restricción que no se cumple para los datos actuales. Para realizar todos estos cambios, se usa el comando ALTER TABLE, con una sintaxis simple pero con muchas variantes:

ALTER TABLE nombre_tabla ADD COLUMN nombre_columna tipo_columna
ALTER TABLE nombre_tabla DROP COLUMN nombre_columna
ALTER TABLE nombre_tabla RENAME COLUMN nombre_columna_viejo TO nombre_columna_nuevo
ALTER TABLE nombre_tabla nombre_columna TYPE tipo_nuevo [USING expresión]
ALTER TABLE nombre_tabla ALTER COLUMN [SET DEFAULT valor | DROP DEFAULT]
ALTER TABLE nombre_tabla ALTER COLUMN [SET NOT NULL | DROP NOT NULL]
ALTER TABLE nombre_tabla ADD CHECK expresión
ALTER TABLE nombre_tabla ADD CONSTRAINT nombre definicion_restriccion
ALTER TABLE nombre_tabla_viejo RENAME TO nombre_tabla_nuevo

Las columnas que se creen nuevas tendrán valor NULL en las filas que ya existían.

-Eliminar tablas:

Este proceso es muy sencillo, pero tiene mucho efecto en la base de datos:

DROP TABLE nombre_tabla

-Tablas temporales:

En ciertas ocasiones se necesita una tabla para un procedimiento aislado que no se va a repetir, por el que no es necesario crear una tabla convencional, o quizá demasiado laborioso. En estos casos una solución es una tabla temporal que almacene cierto número de filas, que se crea con el comando CREATE TEMPORARY TABLE, de forma similar a como se crea una tabla convencional. La tabla se accede de la misma forma que el resto de tablas, pero la diferencia radica en que cuando la sesión o la conexión acaban, la tabla se elimina automáticamente.

TRANSACCIONES

En una situación óptima los cambios en la base de datos se realizan con sentencias declarativas de una en una. Sin embargo, en las aplicaciones reales a veces se tienen que realizar una serie de cambios sobre la base de datos, y éstos no pueden realizarse con una única sentencia SQL. Y aun así se tienen q realizar todos los cambios para que se actualice la base de datos de forma correcta. De esta forma que si ocurre cualquier problema en cualquier parte de los cambios, ninguna modificación se tiene en cuenta en la base de datos. Dicho de otra forma, se necesita llevar a cabo una única e indivisible unidad de trabajo, que requiere ejecutar diversas sentencias de SQL, de forma que o todas las sentencias se ejecutan correctamente, o ninguna se ejecuta.

El ejemplo típico de una transacción es la transferencia de dinero en un banco de una cuenta a otra, en la que el dinero está en una cuenta antes de la transacción, y está en la otra cuenta después. Si ocurre algún error, la transacción no se lleva a cabo, y la situación está igual que si no se hubiese intentado hacerla, y por tanto el dinero no puede desaparecer o aparecer duplicado en las cuentas.

En bases de datos basados en ANSI SQL, como lo es PostgreSQL, se define una transacción como la ejecución de una tarea todo-o-nada. O dicho más técnicamente, una transacción es una unidad lógica de trabajo que no puede ser particionada.

-Unidades lógicas:

Una unidad lógica es un conjunto de cambios lógicos en la base de datos, que puede ocurrir por completo, o fallar por completo. En PostgreSQL, estos cambios se controlan con 4 palabras clave:

- BEGIN: empieza una transacción
- SAVEPOINT nombre_savepoint: le pide al servidor recordar el estado actual de la transacción en un punto dado, es decir un punto de retorno. Esta sentencia se puede utilizar únicamente dentro de una transacción (después de un BEGIN, y antes de un COMMIT o ROLLBACK).
- COMMIT: señala que todos los elementos de la transacción se han completado, y a partir de este momento debe hacerse persistente y accesible al resto de transacciones.
- ROLLBACK [TO nombre_savepoint]: señala que la transacción se debe abandonar, y todos los cambios hechos sobre los datos se deben cancelar. La base de datos debe mostrarse a todos

los usuarios como si la transacción (desde el BEGIN) no se hubiese intentado nunca, y la transacción ha sido cerrada. La versión alternativa, con la adición de la cláusula TO, permite hacer un rollback al punto de retorno en que se pidió al servidor hacer un SAVEPOINT nombre_savepoint.

-Acceso multiusuario simultáneo a los datos:

Otro aspecto de las transacciones es que cualquier transacción sobre la base de datos está aislada del resto de transacciones que ocurren en la base de datos al mismo tiempo. En una situación óptima, cada transacción debería comportarse como si tuviese acceso exclusivo a los datos de la base de datos. En una situación real, donde distintos usuarios están accediendo simultáneamente, conseguir un buen funcionamiento significa determinar cuáles son las posibles situaciones.

En el ejemplo de una compra online de un billete de avión, dos personas pueden consultar la última plaza, pero hasta que uno no finalice la compra, esa plaza se muestra como libre. Si las dos personas inician la compra, habrán empezado dos transacciones intentando actualizar la misma información sobre la base de datos. Una solución al ejemplo sería volver a comprobar que el asiento está libre justo en el momento del pago, pero igualmente existirían unos instantes (el proceso de pago) en el que otro usuario visualizaría el asiento como libre. Otra solución sería extremar las precauciones, y permitir que sólo una persona acceda a la compra del billete, pero la aplicación se haría lenta y liosa para usuarios que sólo quieren comprobar que existen plazas libres (aunque el funcionamiento sería totalmente correcto).

En términos de la aplicación, este proceso es una sección crítica de código (una pequeña sección de código que necesita acceso exclusivo a algunos datos). Se podría programar para que se manejase el acceso como un semáforo, o alguna técnica similar, lo que requeriría que cada aplicación que accediese a la base de datos utilizase el semáforo. Sin embargo, más que programar la solución de forma lógica, es más sencillo que la propia base de datos controle esta situación.

En términos de base de datos, esta situación es una transacción, es decir, el conjunto de manipulaciones de datos que ocurren como una única unidad de trabajo.

-Reglas ACID:

Las siglas ACID (en inglés) se usan frecuentemente para describir las cuatro propiedades que debe tener una transacción:

- Atomicidad (Atomic): Una transacción, incluso si es un grupo de acciones individuales sobre la base de datos, debe ocurrir como una única unidad. Una transacción debe ocurrir una única vez, sin subconjuntos y sin repeticiones inintencionadas de la acción. En el ejemplo de la transferencia bancaria, el movimiento del dinero debe ser atómico, es decir, sacar el dinero de una cuenta y meter el dinero en la otra cuenta debe ocurrir como una única acción, aunque se requieren varias sentencias SQL.

- Consistencia (Consistent): Al final de una transacción, el sistema debe quedarse en un estado consistente. Es decir, si al acabar la transacción existe en la base de datos alguna restricción que no se cumple, se debe volver al estado previo a la transacción. En el ejemplo, al acabar la transferencia las dos cuentas reflejan correctamente el movimiento del dinero.

- Aislamiento (Isolated): Cada transacción, sin importar cuántas transacciones se estén llevando a cabo en el mismo instante, deben actuar independientemente de las demás. En el ejemplo, si dos transferencias se están realizando al mismo tiempo, cada una debe actuar como si tuviese uso exclusivo de la base de datos, aunque realmente no lo tenga. En la práctica esto no es sencillo, y hay que marcar el comportamiento de la base de datos para cada caso. Esto se describe con más profundidad en el apartado de transacciones con múltiples usuarios.

- Durabilidad (Durable): Una vez la transacción se ha completado, debe permanecer completada. En el ejemplo, el dinero ha sido correctamente transferido entre las cuentas, y debe permanecer transferido aunque falle el sistema. En PostgreSQL, como en la mayoría de bases de datos relacionales, esto se logra usando un archivo de registro (log), descrito en el apartado de logs de transacciones.

-Logs de transacciones:

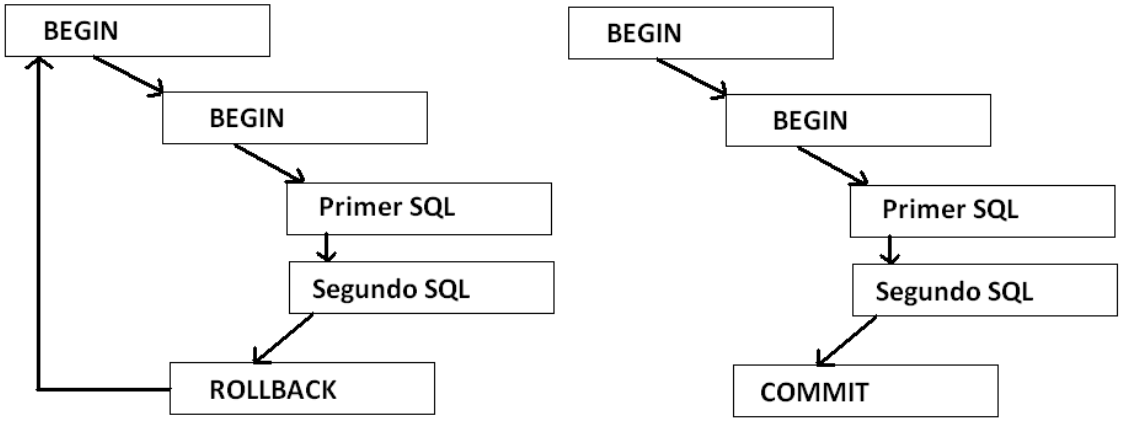
Los logs de transacciones, o archivos de registros de transacciones, se utilizan por la base de datos de forma interna para asegurarse que una transacción perdura. La forma en que funciona el archivo es simple, mientras se ejecuta una transacción, los cambios se realizan en base de datos y además en el log. Una vez que la transacción se completa, un marcador señala que la transacción ha acabado, y que los datos del log se guardan de forma permanente, y por tanto aunque fallase el sistema, los datos se han asegurado. Si el sistema fallase por cualquier motivo en medio de una transacción, cuando se reiniciase debe asegurarse automáticamente que las transacciones completadas se han visto reflejadas correctamente en la base de datos. Además se asegura que ningún cambio de las transacciones que aun estaban en proceso aparece en la base de datos.

En PostgreSQL el log mantiene, además de todas las filas que están siendo modificadas, sino también las filas en la situación anterior. Esto hace que este archivo ocupe mucho espacio en poco tiempo. Una vez que la sentencia COMMIT se realiza para una transacción, PostgreSQL reconoce que ya no hace falta la información de esa transacción, porque el cambio ya está hecho sobre la base de datos.

PostgreSQL usa una técnica en la que los datos se guardan en el log de la transacción antes de que se guarden en las tablas, porque sabe reconocer que una vez que los datos se guardan en el log, se puede recuperar el estado de la tabla desde el log, incluso si el sistema falla antes de que los datos reales se hayan actualizado. A esta técnica se le llama WAL (Write Ahead Logging).

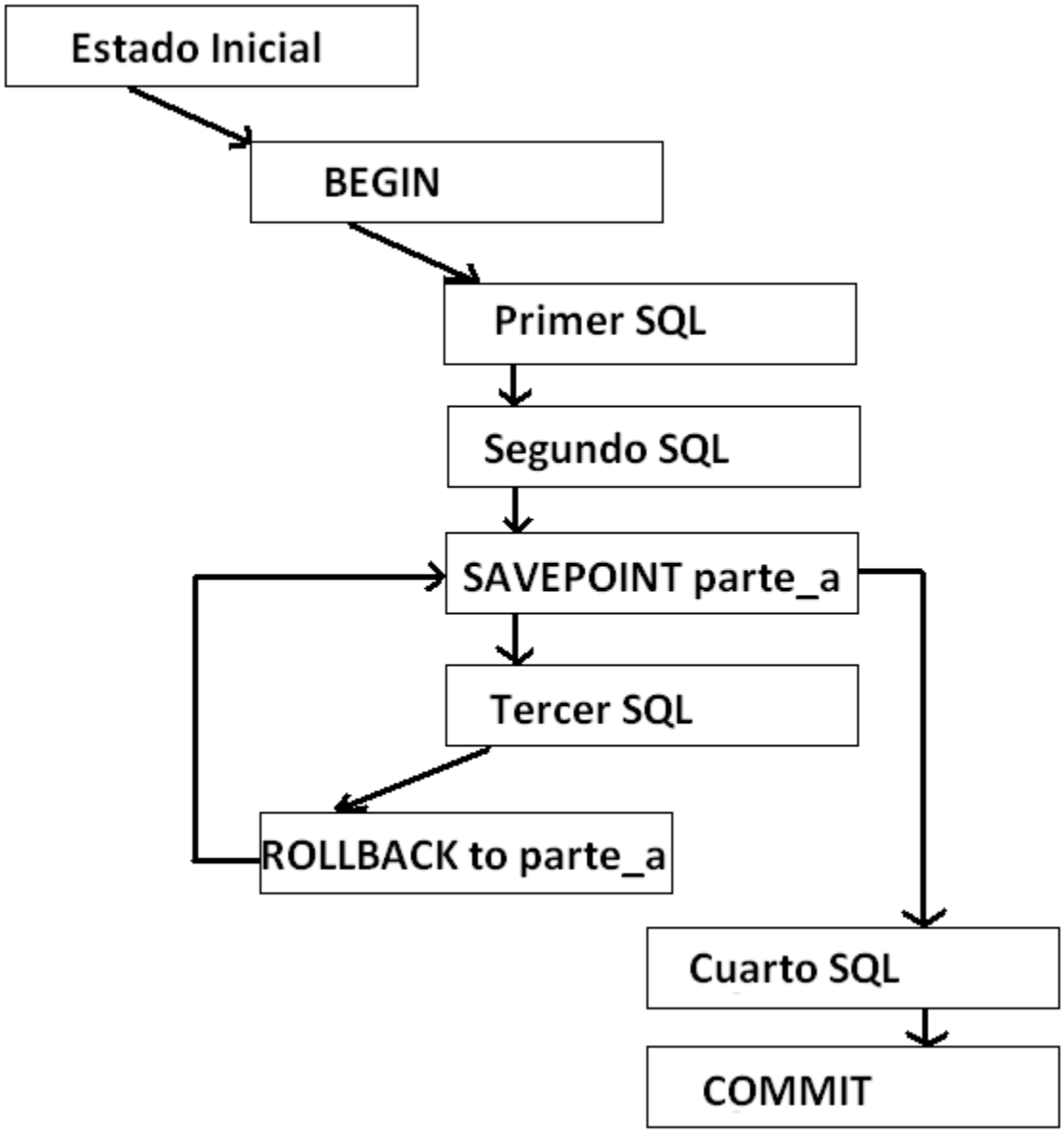
-Único usuario:

Para entender mejor el acceso de varios usuarios, hay que ver por qué fases pasa un único usuario. Aunque sea una forma simple de acceder, las transacciones pueden aportar ciertas ventajas. La gran ventaja de crear transacciones es que permiten ejecutar varias sentencias SQL, y posteriormente deshacer el proceso realizado. Además, si falla una de las sentencias SQL, se puede deshacer el proceso hasta un determinado punto. Usando una transacción, la aplicación no necesita preocuparse de almacenar lo que los cambios han realizado sobre la base de datos y cómo deshacerlos. Simplemente le pide al motor de base de datos que deshaga el conjunto de cambios a la vez.



Si se decide que todos los cambios sobre la base de datos son válidos, y que se apliquen de forma permanente, tras el 'Segundo SQL' se ejecutará un COMMIT. Después del COMMIT todos los cambios estarán asegurados en la base de datos y no se perderán ante cambios del sistema.

Las transacciones no se limitan a una única tabla o a actualizaciones simples, pero al ejecutarlas con un único usuario, los datos se procesan correctamente. La mayoría de aplicaciones sólo necesitan transacciones básicas como las vistas. Sin embargo, los puntos de retorno (savepoints) pueden ser útiles cuando se quieren deshacer sólo unas sentencias dentro de toda la transacción, haciendo uso del comando ROLLBACK TO. Haría falta crear un punto de retorno con un nombre al que hacer rollback.



-Limitaciones en las transacciones:

Aunque las transacciones funcionen correctamente, tienen ciertas limitaciones en lo que respecta al anidamiento, al tamaño y a la duración:

- Anidamiento de transacciones: PostgreSQL no permite anidamiento en las transacciones (ni tampoco lo permiten la mayoría de bases de datos relacionales), y si dentro de una transacción (un BEGIN) se encuentra el inicio de otra (otro BEGIN), PostgreSQL devolverá una advertencia (warning) señalando que una transacción ya está en marcha.

- Tamaño de transacción: Es aconsejable mantener las transacciones lo más pequeñas posible. El funcionamiento que realiza PostgreSQL para asegurarse que las transacciones de distintos usuarios se mantienen separadas requiere de mucho procesamiento. Una consecuencia de esto es que las partes de la base de datos que están relacionadas con una transacción a veces

necesitan estar bloqueadas, para asegurarse que están separadas de las otras transacciones. Excesivas cantidades de modificaciones en una transacción hará que se bloqueen excesivas cantidades de datos, perjudicando al funcionamiento y a otros usuarios que acceden a los datos. Se detallarán con más profundidad los bloqueos en el apartado de bloqueos.

- Duración de transacción: Las transacciones no deben extenderse largos periodos de tiempo. Aunque PostgreSQL bloquea la base de datos para el usuario, cuando la transacción se alarga acaba perjudicando a otros usuarios que acceden a los datos relacionados con la transacción hasta que la transacción acaba. El programador debe evitar que dentro de la transacción se esté solicitando algún dato o alguna confirmación al usuario, ya que eso alargaría la transacción hasta periodos impredecibles. También hay que tener en cuenta que aunque el proceso interno de un COMMIT es bastante rápido, cuando se trata de un ROLLBACK de muchos procesos, el sistema necesitará bastante tiempo para realizarlo.

CONCURRENCIA

Las transacciones que se ejecutan por múltiples usuarios de forma concurrente deben estar aislados unos de otros (la I de ACID). Aunque PostgreSQL controla el aislamiento correctamente en la mayoría de los casos, hay ciertas circunstancias que hay que tener en cuenta.

-Implementación del aislamiento:

Uno de los aspectos más complejos para las bases de datos relacionales es el aislamiento entre distintos usuarios a la hora de actualizar la base de datos. Si la aplicación no necesita de una buena ejecución hay formas sencillas de aislamiento, simplemente permitiendo una única transacción en cada momento dado. Este extremo bloqueará la aplicación para el resto de usuarios, por lo que no es posible aplicarlo en la mayoría de los casos. Por esto se utilizan métodos con ciertos riesgos. Para minimizar el impacto del aislamiento, el estándar SQL define distintos niveles de aislamiento para una base de datos. Esto permite al administrador de la base de datos elegir entre un mejor funcionamiento o un mejor aislamiento. Normalmente una base de datos relacional implementa al menos uno de estos niveles por defecto, y también permite a los usuarios especificar al menos otro nivel. Los niveles se definen según la situación no deseada que puede ocurrir cuando interactúan múltiples usuarios. Estas situaciones son ‘lecturas sucias’, ‘lecturas no repetibles’, ‘lecturas fantasma’ y ‘pérdida de actualizaciones’.

- Lecturas sucias (en inglés ‘dirty reads’): Una lectura sucia ocurre cuando una transacción lee datos que están siendo cambiados por otra transacción, pero la transacción que está modificando los datos aun no ha acabado (no ha hecho COMMIT). Como se ha visto antes, una transacción es una unidad lógica o bloque de trabajo que debe ser atómica. O ocurre todas las sentencias de la transacción o no ocurre ninguna. Hasta que la transacción hace COMMIT, existe la posibilidad de que falle o que haga un ROLLBACK. Por tanto ningún otro usuario debe poder ver los cambios de la transacción. PostgreSQL no permite en ningún caso lecturas sucias.

- Lecturas no repetibles (en inglés 'unrepeatable reads'): Una lectura no repetible es una situación similar a la anterior, pero más restrictiva. Esta situación ocurre cuando una transacción lee un conjunto de datos, luego la vuelve a leer y descubre que se ha modificado. Es una situación menos seria que la anterior, pero no es una situación óptima. Hay que señalar que en este caso la transacción puede ver los cambios que han realizado otras transacciones, aunque la transacción de lectura no ha acabado. Si el sistema está configurado para prevenir lecturas no repetibles, las transacciones no verán los cambios que realizan los demás hasta ellas mismas acaban (hacen COMMIT). PostgreSQL permite lecturas no repetibles por defecto, pero puede configurarse para que no las permita.

- Lecturas fantasmas (en inglés 'phantom reads'): Una lectura fantasma es una situación similar a la anterior, pero ocurre cuando se inserta una nueva fila en una tabla mientras otra transacción está actualizando la misma tabla, y por tanto la nueva fila debería haberse actualizado pero no lo hace. Las lecturas fantasma ocurren muy pocas veces y difíciles de localizar, por lo que normalmente los sistemas no las tratan de ninguna forma. PostgreSQL permite lecturas fantasmas.

- Pérdida de actualizaciones (en inglés 'lost updates'): Las pérdidas de actualizaciones son situaciones bastantes distintas a las anteriores, y ocurre cuando dos actualizaciones se escriben en la base de datos, y la segunda causa la pérdida de la primera. Esta situación se puede dar aunque la base de datos esté correctamente aislada, ya que a partir de los datos iniciales, dos actualizaciones se realizan sobre el mismo dato, y sólo la última tendrá efecto. Hay varias formas de tratar este problema, y elegir la más apropiada dependerá de cada aplicación. Primero hay que manejar transacciones lo más pequeñas posibles. Segundo, las aplicaciones deben guardar sólo información que ha sido modificada. Estos dos pasos previenen muchas situaciones de pérdida de actualizaciones.

- Niveles de aislamiento: El estándar ANSI define diferentes niveles de aislamiento que una base de datos puede utilizar como combinación de los 3 tipos de situaciones no deseables que pueden ocurrir:

Nivel de aislamiento	Lecturas sucia	Lecturas no repetibles	Lecturas fantasma
Lectura no confirmada	Posible	Posible	Posible
Lectura confirmada	No Posible	Posible	Posible
Lectura repetible	No Posible	No Posible	Posible
Serializable	No Posible	No Posible	No Posible

-Niveles de aislamiento:

Por defecto en PostgreSQL el aislamiento está configurado al nivel de 'lectura confirmada', y la otra opción que se puede seleccionar es el nivel de 'serializable'. Los otros niveles no se pueden seleccionar, el de 'lectura no confirmada' porque no solucionan los conflictos, y el de 'lectura repetible' únicamente previene el caso de 'lectura fantasma' que se un caso muy poco

frecuente. Para cambiar el nivel de aislamiento, se usa el comando SET TRANSACTION ISOLATION LEVEL:

```
SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }
```

-Transacciones explícitas y transacciones implícitas:

Para explicar el principio y final de una transacción de forma simple, se ha usado BEGIN y COMMIT (o ROLLBACK), pero PostgreSQL por defecto opera en modo 'auto-commit', a veces diferido a un 'modo encadenado' o 'modo de transacción implícita', donde cada sentencia SQL puede modificar datos como si se tratase de una transacción. Esto ayuda a realizar pruebas en pgAdmin III, o para comprobar la información antes de aceptarla, aunque para el uso en aplicaciones no es muy eficiente, donde es deseable definir explícitamente donde empieza una transacción (BEGIN) y donde acaba (COMMIT o ROLLBACK). En PostgreSQL para cambiar el modo (de implícito a explícito), sólo se ha de escribir la palabra BEGIN, y PostgreSQL cambia automáticamente de modo, hasta que encuentra la palabra COMMIT o ROLLBACK. En el estándar de SQL se consideran todas las sentencias SQL en una misma transacción, por lo que la transacción empieza automáticamente en la primera sentencia SQL, y continúa hasta que se realice un COMMIT o ROLLBACK. Por tanto en el estándar no existe la palabra BEGIN, pero otros sistemas además de PostgreSQL la utilizan.

BLOQUEOS

Las bases de datos también tienen otro mecanismo de controlar que una transacción de un usuario se realiza aislada del resto de transacciones, y es usando bloqueos para restringir el acceso a los datos desde otros usuarios. Hay dos tipos de bloqueos:

- Bloqueo compartido: el que permite a los demás usuarios leer pero no actualizar datos.
- Bloqueo exclusivo: el que evita que otras transacciones puedan leer o actualizar los datos.

Por ejemplo, el servidor bloquea filas que se están modificando por una transacción, hasta que la transacción acaba y entonces el bloqueo se libera. Esto ocurre en las bases de datos de forma automática, de forma invisible al usuario. Pero hay mecanismos mucho más complejos para tratar de formas muy distintas los bloqueos. En PostgreSQL se pueden realizar ocho tipos de bloqueos, o incluso se puede realizar una tipo multiversión que reduce los conflictos entre bloqueos y mejora el rendimiento si se compara con los demás sistemas. Desde el punto de vista del usuario, sólo hay dos circunstancias que pueden configurar: evitando interbloqueos (y recuperarse de ellos) y bloqueos explícitos desde una aplicación.

-Interbloqueos:

Es lo que sucede cuando dos aplicaciones diferentes intentan cambiar el mismo dato en el mismo momento. Ambas sesiones se bloquean, y cada una se queda esperando a que la otra libere el dato. Este comportamiento es la razón porque PostgreSQL usa por defecto el modo de lectura confirmada para el aislamiento de transacciones, ya que aporta un equilibrio entre concurrencia, rendimiento y minimización del número de bloqueos por una parte, y consistencia y comportamiento óptimo por la otra parte.

Conforme el usuario incrementa el nivel de aislamiento, el rendimiento de la aplicación multiusuario disminuye. Si el comportamiento de la base de datos es óptimo, el número de bloqueos necesarios aumenta, la concurrencia entre usuarios disminuye, pero sobretodo el rendimiento baja. Por tanto, encontrar un punto de equilibrio es necesario.

-Bloqueos explícitos:

En ciertos casos, para el usuario puede no ser suficiente el bloqueo automático de PostgreSQL. En ese caso puede bloquear de forma explícita algunas filas o la tabla entera. El usuario debe usar sólo este tipo de bloqueos para procesos críticos, y evitarlo siempre que sea posible. El bloqueo explícito de una tabla no es un estándar de SQL, pero se usa comúnmente en muchas bases de datos.

Es posible bloquear filas o tablas en medio de una transacción. Una vez la transacción acaba, con un COMMIT o con un ROLLBACK, todos los bloqueos que se generaron durante la transacción se liberan automáticamente. No hay forma de liberar explícitamente los bloqueos en la transacción.

- Bloqueo de filas: Lo más común es que el usuario necesite únicamente bloquear ciertas filas para modificarlas. Es una forma de asegurarse que no van a existir interbloqueos, al bloquear desde un principio las filas que se sabe que van a ser modificadas, y así asegurar que otras aplicaciones no entran en conflicto con los cambios del usuario. Para bloquear un conjunto de filas, se usa una sentencia SELECT seguida de de FOR UPDATE:

```
SELECT nombre_columna FROM nombre_tabla WHERE condición FOR UPDATE;
```

- Bloqueo de tablas: En PostgreSQL es posible bloquear tablas, aunque no se recomienda ya que no forma parte del estándar SQL. Para bloquear una tabla hay tres tipos de comandos:

LOCK [TABLE] nombre_tabla
LOCK [TABLE] nombre_tabla IN [ROW | ACCESS] { SHARE | EXCLUSIVE } MODE
LOCK [TABLE] nombre_tabla IN SHARE ROW EXCLUSIVE MODE

BIBLIOGRAFÍA

- [1] “PostgreSQL Introduction and Concepts”
- [2] “Beggining Databases with PostgreSQL: From Novice to Professional, Second Edition”
- [3] http://es.wikipedia.org/wiki/SQL

TEMA 3 – INTEGRIDAD SEMÁNTICA

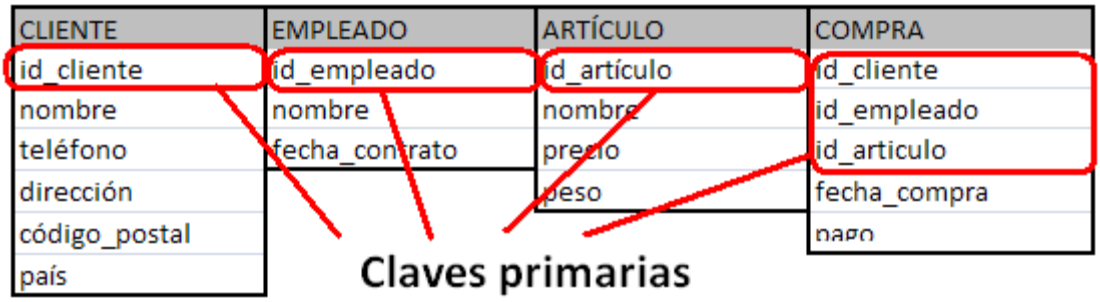
RESTRICCIÓN POR CLAVE PRIMARIA (PK)

La restricción de clave primaria (PK: Primary Key) es la que marca la columna como única en la tabla, y por eso sirve de identificador de cada fila.

Técnicamente, una clave primaria es simplemente una combinación de la restricción de unicidad y la restricción de valor no nulo, aunque en una clave primaria caben más de una columna. Una clave primaria indica que una columna o grupo de columnas se pueden usar como identificadores únicos de filas en la tabla, lo que le proporciona funcionalidades especiales para la documentación y para las aplicaciones, por ejemplo en una aplicación del cliente que necesite modificar algún valor de una fila, deberá conocer el valor de la clave primaria para poder limitar esa fila a la hora de modificarla.

Una tabla no puede tener más de una clave primaria (aunque puede tener varias restricciones de unicidad y de valor no nulo). Aunque lo más estándar y eficiente es que cada tabla tenga una clave primaria, PostgreSQL permite crear tablas que no la tengan.

En los ejemplos de los temas anteriores, en la tabla 'cliente' habría que elegir un atributo que fuese único y no nulo, como podría ser el dni (suponiendo que el cliente siempre proporciona este valor), o un identificador interno 'id_cliente' independiente de los cambios de valor de los demás atributos. De forma similar se asigna clave primaria a las tablas 'empleado' y 'artículo'. En el caso de la tabla 'compra' se podría actuar de igual forma creando un atributo 'id_compra', pero como en el ejemplo cada compra se define como una relación entre cliente, empleado y artículo (suponiendo que una compra sólo incluye un artículo), se ha elegido estos tres atributos como clave primaria de la tabla.

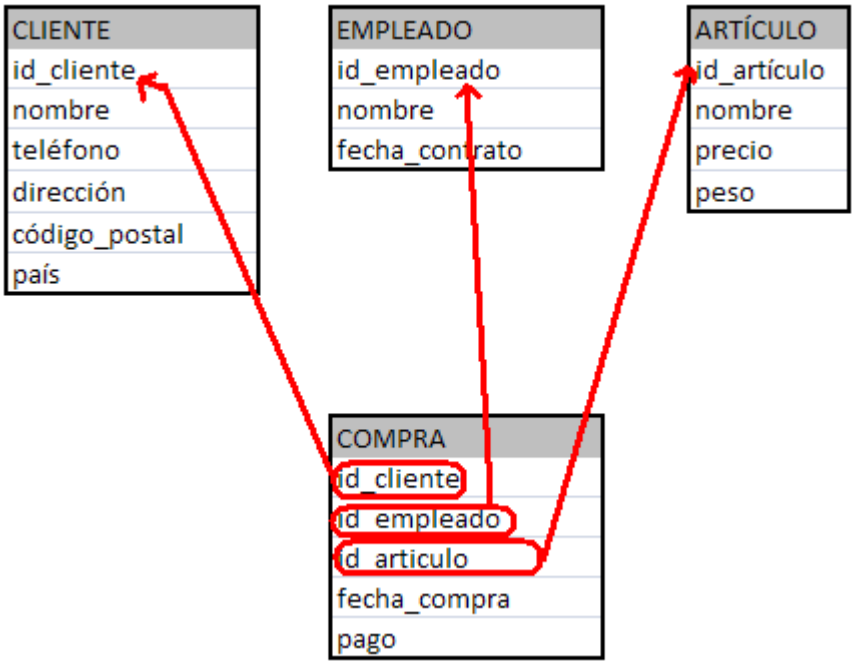


RESTRICCIÓN DE CLAVE AJENA (FK)

Una de las restricciones más importantes en las bases de datos es la restricción por clave ajena (FK: Foreign Key). En las tablas de ejemplo había datos que unían varias tablas, o mejor dicho las relacionaban. En la tabla 'compra', hay campos que relacionan cada fila con las tablas 'cliente', 'empleado' y 'artículo'.

Esta relación permite, a partir de una compra, obtener información personal del cliente, o del empleado, o detalles del artículo, aunque toda esta información no se encuentre en la tabla 'compra'. La relación en las tablas es clave primaria, por lo que cada fila de 'compra' tiene una relación con una única fila en cada una de las tablas 'cliente', 'empleado' y 'artículo'. Esto no ocurre en sentido opuesto: un cliente puede tener más de una compra.

Cada una de estas relaciones forma una clave ajena. Es decir, el campo 'id_cliente' de la tabla 'compra' tiene como clave ajena el campo 'id_cliente' de la tabla 'cliente'. Como la relación no es inversa, no existe clave ajena en el campo 'id_cliente' de la tabla 'cliente'.



Una parte importante del diseño de una base de datos es crear la estructura de claves ajenas, es decir crear correctamente la relación en la base de datos para que sea una base de datos relacional. Además esta restricción es una forma de asegurar que las relaciones entre tablas se mantienen ante intentos de borrado.

Cuando se crea una restricción de clave ajena, PostgreSQL comprueba que la columna referenciada está declarada como única en la tabla. Es muy común que la columna referenciada por la clave ajena es la clave primaria de la tabla (asegurando su unicidad).

En una tabla puede haber varias claves ajenas, igual que en el ejemplo la tabla 'compra' tiene 3 claves ajenas. O también puede darse el caso de que la clave ajena está formada por 2 ó más columnas, siempre que esas columnas sean únicas en la tabla referenciada.

La clave ajena evita situaciones como que la columna tenga un valor que no existe en la tabla relacionada, es decir, que la compra tenga un identificador de cliente, y ese identificador no se

corresponda con ningún cliente. Durante las inserciones, actualizaciones o borrados en las bases de datos, habrán situaciones en los que la relación o la unicidad se puedan perder, y PostgreSQL está configurado para actuar ante esta situación de diversas formas, según lo programe el diseñador de la base de datos, con distintas reglas.

-Clave ajena como restricción de columna:

La sentencia para declarar una clave ajena es sencilla. Darle un nombre a la restricción es opcional en PostgreSQL, pero es recomendable porque aparecerá ese nombre en los posibles errores de integridad que se den.

```
[CONSTRAINT nombre_restricción] nombre_columna tipo
REFERENCES tabla_referenciada (columna_referenciada);
```

-Clave ajena como restricción de tabla:

Aunque se puede declarar la clave ajena a nivel de columna, suele quedar más claro a nivel de tabla, junto con la declaración de clave primaria. Además a nivel de tabla también se permite crear la clave ajena para varias columnas. La sentencia para declarar la clave ajena es similar a la anterior.

```
CONSTRAINT [nombre_restricción] FOREIGN KEY (lista_columnas)
REFERENCES tabla_referenciada (lista_columnas_referenciadas);
```

Se puede añadir la restricción a una tabla ya existente, mediante la sentencia para modificación de la tabla ALTER TABLE.

```
ALTER TABLE tabla
ADD CONSTRAINT [nombre_restricción] FOREIGN KEY (lista_columnas)
REFERENCES tabla_referenciada (lista_columnas_referenciadas);
```

-Reglas:

A veces la modificación de datos puede romper la relación de integridad que se ha creado por una clave ajena. Esto es muy frecuente, un ejemplo sería querer modificar en la tabla cliente el campo 'id_cliente', que está referenciado en la tabla 'compra'. La clave ajena no permitiría modificar el valor en la tabla cliente, porque tiene valores referenciados en 'compra', ni tampoco cambiar esos valores en 'compra', porque debe existir una fila referenciada en la tabla 'cliente'. PostgreSQL previene que ante esta violación de la clave ajena, con dos posibles soluciones según el estándar SQL:

- Posponer la restricción (restricción diferida)
- Especificar reglas en la clave ajena para manejar las violaciones

-Restricción diferida:

Sirve para modificar la forma en que se cumple la restricción, ya que normalmente comprueba que se cumplen las restricciones antes de permitir cualquier cambio, pero en este caso esperará a que se acabe la transacción entera para comprobar las restricciones. De esta manera las restricciones se incumplirán durante un momento, pero únicamente para el

usuario actual, ya que el resto verán los datos antes y después de la transacción, pero nunca a mitad.

Basta con añadir INITIALLY DEFERRED al final de la creación de la restricción:

```
CONSTRAINT [nombre_restricción] FOREIGN KEY (lista_columnas)
REFERENCES tabla_referenciada (lista_columnas_referenciadas)
INITIALLY DEFERRED;
```

En el ejemplo habría que iniciar una transacción, modificar el 'id_cliente' de la tabla 'cliente', actualizar los 'id_clientes' relacionados en la tabla 'compra', y después validar la transacción. En este momento será cuando PostgreSQL comprobará las restricciones.

-Reglas para las restricciones:

Es una alternativa más automática que la anterior, en la que se especifica qué reglas ha de realizar la base de datos para manejar violaciones de la restricción. Se distinguen dos situaciones en las que la base de datos debe actuar:

- ON UPDATE: acción a realizar cuando se intenta actualizar un dato que viola la clave ajena.
- ON DELETE: acción a realizar cuando se intenta borrar un dato que viola la clave ajena.

Y existen dos acciones posibles:

- En cascada (CASCADE): modificar en cascada desde la tabla con la clave primaria.

```
CONSTRAINT [nombre_restricción] FOREIGN KEY (lista_columnas)
REFERENCES tabla_referenciada (lista_columnas_referenciadas)
ON DELETE CASCADE;
```

En este caso, PostgreSQL ante el borrado (o modificación) del 'id_cliente' de la tabla 'cliente', borrará (o actualizará al mismo valor) la fila referenciada en la tabla 'compra' de forma automática. Esta opción es oculta al usuario, ya que no ve visualmente los cambios realizados, y puede eliminar información de tablas que no desea eliminar.

- Dejar a nulo (SET NULL): modificar a NULL cuando deje de existir la referencia.

```
CONSTRAINT [nombre_restricción] FOREIGN KEY (lista_columnas)
REFERENCES tabla_referenciada (lista_columnas_referenciadas)
ON UPDATE SET NULL;
```

En este caso, PostgreSQL ante el borrado (o modificación) del 'id_cliente' de la tabla 'cliente', dejará a NULL ese campo en las filas referenciadas en la tabla 'compra'. Evidentemente si el campo está definido como NOT NULL, no funcionará y PostgreSQL mostrará un mensaje de error. Esta opción puede no ser una buena opción si se quiere mantener valor en el campo con clave ajena. En el ejemplo, se podrían dejar compras que tienen cliente desconocido.

Ambas acciones pueden combinarse, una ante borrados y otra ante actualizaciones:

```
CONSTRAINT [nombre_restricción] FOREIGN KEY (lista_columnas)
REFERENCES tabla_referenciada (lista_columnas_referenciadas)
ON UPDATE SET NULL
ON DELETE CASCADE;
```

Más adelante se explica otra posible solución, que es asociar triggers para controlar con más profundidad el efecto que tiene una actualización o un borrado.

BIBLIOGRAFÍA

- [2] “Beggining Databases with PostgreSQL: From Novice to Professional, Second Edition”
- [3] http://www.postgresql.org/docs/8.1/static/ddl-constraints.html

TEMA 4 – RECUPERACIÓN

COPIA DE SEGURIDAD (BACKUP) Y RECUPERACIÓN

La copia de seguridad y recuperación es una apartado muchas veces pasado por alto, pero con posibles consecuencias nefastas. Un sistema de bases de datos depende de sus datos, y los datos pueden perderse de varias formas, desde un fallo eléctrico en el disco duro, un error humano al borrar archivos equivocados, una mala programación sobre los datos,…. Todas las bases de datos de PostgreSQL deben realizar una copia de seguridad de forma periódica, ya que guardar una copia de los datos fuera del sistema ayudará a proteger los datos.

Una buena copia de seguridad y un buen plan de recuperación son los que han sido probados y demuestran que funcionan, y los que obtienen la copia de forma periódica. Aunque PostgreSQL usa ficheros normales en el sistema operativo para almacenar los datos, no es aconsejable basarse en la copia de seguridad del sistema operativo. Si esta copia se hiciese cuando la base de datos está activa y funcionando, puede que no almacene información estable o consistente. Una solución sería mantener el servidor de base de datos apagado durante la copia, pero PostgreSQL ofrece otra solución. PostgreSQL proporciona sus propios mecanismos para copia de seguridad y respaldo: ‘pg_dump’ y ‘pg_restore’. Además es posible realizar copias de seguridad desde pgAdmin III.

Una lista de posibles situaciones en las que PostgreSQL podría perder información son:

Evento	Acción PostgreSQL
Caída del cliente	PostgreSQL hará rollback en las transacciones en marcha para ese cliente
Fallo de red del cliente	PostgreSQL hará rollback en las transacciones en marcha para ese cliente
Caída del servidor	PostgreSQL hará rollback en las transacciones incompletas cuando se reinicie el servidor
Caída del sistema operativo sin pérdida de datos	PostgreSQL hará rollback en las transacciones incompletas cuando se reinicie el servidor
Borrado accidental de datos de base de datos o de una tabla	Se requiere recuperación manual desde el respaldo
Borrado accidental de archivos de PostgreSQL desde el sistema operativo	Se requiere recuperación manual desde el respaldo
Fallo de disco u otros fallos de archivos de PostgreSQL	Se requiere recuperación manual desde el respaldo

Un problema típico con las copias de seguridad es que desde que se realizan hasta el momento en que se pierden los datos han transcurrido eventos que han transformado los datos, por lo que hace falta volver a realizar esos eventos intermedios. PostgreSQL ofrece una nueva opción desde su versión 8, llamada PITR, que se basa en las ventajas que proporciona WAL (Write Ahead Logging), para ayudar a mitigar el problema de trabajo perdido mediante registros

adicionales de transacciones. A menos que el evento que causó el fallo de la base de datos esté en el registro, a partir de una restauración se podrá seguir estos registros adicionales para volver a ejecutar los eventos que se ejecutaron entre la creación de la copia de seguridad y el momento del fallo. También es posible elegir los comandos desde un punto dado en el tiempo.

-Creación de la copia de seguridad:

La forma más sencilla de crear la copia de seguridad es ejecutando 'pg_dump' y redireccionando la salida a un fichero:

```
pg_dump [nombre_base_de_datos] [opciones…]
```

Las opciones son:

Opción	Descripción
-f **--file=nombre_archivo**	Especifica un nombre de archivo
-F **--format=c|t|p**	Especifica un formato de salida para el archivo (cliente, tar, o texto plano)
-v **--verbose**	Usa el modo detallado
-Z **--compress=0-9**	Especifica el nivel de compresión para formatos comprimidos
--help	Muestra texto de ayuda
-a **--data-only**	Vuelca sólo los datos, no el esquema
-C **--create**	Incluye comandos para crear la base de datos en el volcado
-c **--clean**	Elimina los objetos antes de la creación
--disable-triggers	Deshabilita los triggers durante una restauración de sólo datos
-d **--inserts**	Vuelca datos como comandos INSERT, y no como comandos COPY
-s **--schema-only**	Vuelva sólo el esquema, no los datos
-S **--superuser=nombre_superusuario**	Especifica el nombre de usuario del superusuario
-t **--table=tabla**	Vuelva solo la tabla especificada
-h **--host=nombre_host**	Especifica el host de servidor de base de datos o directorio del socket
-p **--port=puerto**	Especifica el número de puerto del servidor de base de datos
-U nombre_usuario	Especifica nombre de la copia de seguridad

En esencia la copia de seguridad produce un gran script con comandos SQL y con comandos internos de PostgreSQL, de forma que al ejecutarlos se consigue crear la base de datos entera.

El archivo de salida es ese script escrito en principio en texto plano con las sentencias para la creación de usuarios, privilegios, tablas y datos. No contiene la creación de la base de datos, así que si hiciese falta habría que crearla previamente antes de ejecutar el script. Este archivo también permite copiar la base de datos para otra finalidad (tener una copia separada de los datos reales).

El siguiente comando crearía la estructura de la base de datos en una nueva desde la copia de seguridad, mediante la opción '-f':

```
psql -f copia_seguridad.backup nueva_base_de_datos
```

-Restauración desde la copia de seguridad:

Para restaurar usando un archivo en texto plano, se ejecuta el siguiente comando:

```
pg_dump -U nombre_superusuario nombre_base_de_datos > copia_seguridad.bak
createdb -U nombre_usuario1 nueva_base_de_datos
psql -U nombre_usuario1 -d nueva_base_de_datos < copia_seguridad.bak
```

Como la utilidad 'createdb' contiene los comandos para crear la base de datos, también se puede conectar a la base de datos y usar simplemente el comando CREATE DATABASE. Para restaurar una copia de seguridad que contiene una instalación entera hay que conectarse a una instalación de PostgreSQL con la base de datos por defecto, 'template1'. La copia creada por 'pg_dumpall' contiene sentencias SQL para crear cada base de datos. Se necesita ejecutar la copia de seguridad y restaurar iniciando la sesión como superusuario para tener suficientes permisos para leer y escribir todos los datos:

```
psql -f all.backup template1
```

Si la copia se ha realizado con formato cliente o con formato comprimido, entonces hay que utilizar la utilidad 'pg_restore':

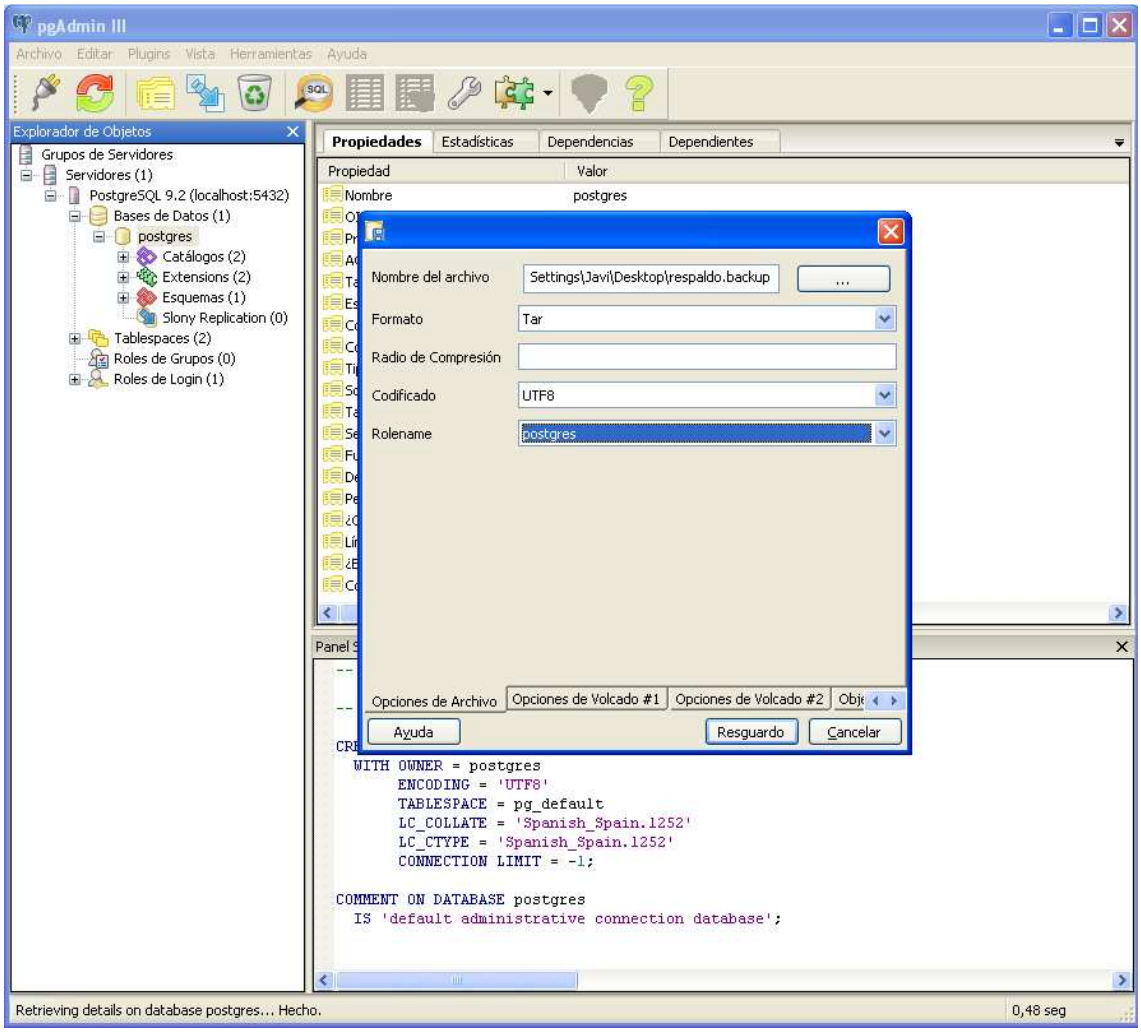
```
pg_restore [archivo] [opciones...]
```

Las opciones más comunes para 'pg_restore' son:

Opción	Descripción
-d **--dbname=nombre_base_de_datos**	Conecta la base de datos especificada
-f **--file=nombre_archivo**	Especifica un nombre de archivo
-F **--format=c|t**	Especifica un formato de copia de seguridad (cliente o tar)
-l **--list**	Imprime un listado resumido de los contenidos del archivo
-v **--verbose**	Usa el modo detallado
--help	Muestra texto de ayuda
-a	Restaura sólo los datos, no el esquema

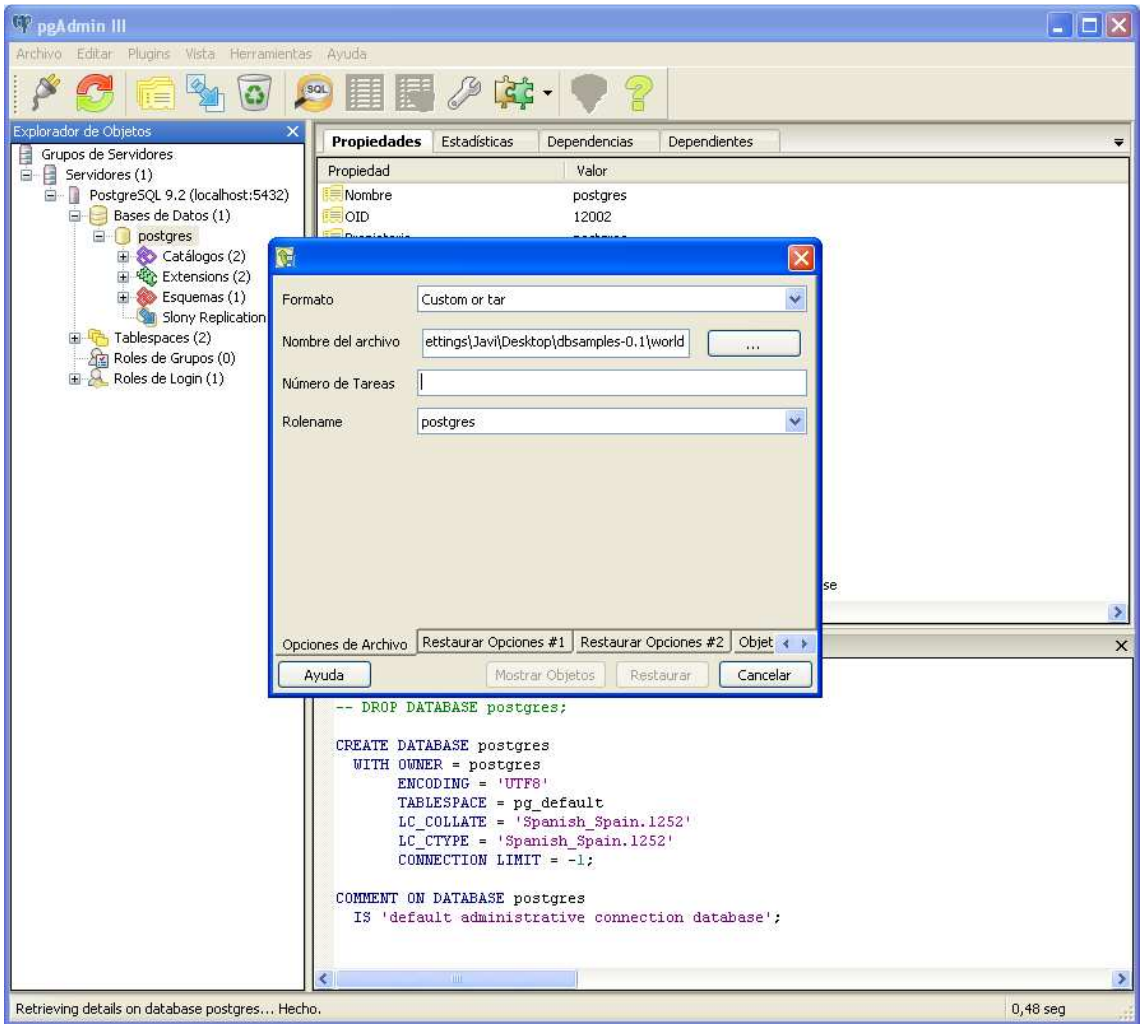
--data-only	
-c **--clean**	Limpia (elimina) el esquema antes de la creación
-C **--create**	Crea la base de datos
-s **--schema-only**	Restaura sólo el esquema, no los datos
-t **--table=tabla**	Restaura sólo la tabla especificada
-h **--host=nombre_host**	Especifica el host de servidor de base de datos o directorio del socket
-p **--port=puerto**	Especifica el número de puerto del servidor de base de datos
-U **--username= nombre_usuario**	Se conecta como el usuario de base de datos especificado
-e **--exit-on-error**	Sale ante un error (por defecto continúa)

- Copia de seguridad y recuperación desde pgAdmin III:



Igual que en todas las operaciones sobre PostgreSQL, pgAdmin III ofrece una forma gráfica de crear copias de seguridad, respaldar la base de datos y crear copias automáticas. Para crear la copia de seguridad En pgAdmin III, hay que hacer clic con el botón derecho sobre el nombre de la base de datos deseado, seleccionar 'Resguardo', y en la ventana seleccionar un archivo destino.

Para restaurar una copia de seguridad hay que seleccionar el objeto de base de datos, hacer clic con el botón derecho y seleccionar la creación de nueva base de datos donde restaurar los datos. Luego clic con el botón derecho sobre la base de datos, seleccionar 'Restaurar', y seleccionar el archivo con la copia de seguridad. Se pueden seleccionar distintas opciones para la restauración.



CONCEDER O REVOCAR PRIVILEGIOS

Los datos de una base de datos pueden tener muchos tipos de restricciones, algunas filas o tablas sólo pueden ser vistas por ciertos usuarios, e incluso para las tablas que son visibles por

todos pueden haber restricciones sobre quién utiliza los datos, insertan nuevos datos, o cambian los existentes. Todo esto se maneja mediante un sistema de privilegios, donde los usuarios tienen distintos privilegios para las distintas tablas u otros objetos de base de datos, como esquemas o funciones. Es una buena práctica no dar permisos directamente a los usuarios, sino crear roles con distintos conjuntos de privilegios, y son los roles los que se asignan a los usuarios. Otro aspecto de la seguridad sobre base de datos es asegurarse que únicamente accede a los datos la persona correcta, y esa persona no puede ver lo que hacen el resto de usuarios, lo que pueden hacer, o el rol que tienen. También es parte de la seguridad asegurarse que los servidores de base de datos están en localizaciones físicamente seguras, y los procesos necesarios para acceder a los servidores también son seguros.

En un sistema básico habrían dos roles, uno de administrador que tienen permiso para todo (superusuarios), y otros de usuarios finales que tienen restringido todo menos ver y modificar algunas pocas tablas. A partir de ahí se puede ir aumentando el esquema de roles para hacerlo conforme a cada caso.

BIBLIOGRAFÍA

- [2] “Beggining Databases with PostgreSQL: From Novice to Professional, Second Edition”
- [7] “PostgreSQL 9 Administration Cookbook”

TEMA 5 – IMPLEMENTACIÓN

ASPECTOS DE DISEÑO LÓGICO

	A	B	E	F
1	Valencia/València	Ademuz	1292 varones	686 mujeres
2	Valencia/València	Ador	1542 varones	792 mujeres
3	Valencia/València	Atzeneta d'Albaida	1236 varones	645 mujeres
4	Valencia/València	Agullent	2479 varones	1222 mujeres
5	Valencia/València	Alaquàs	30235 varones	14952 mujeres
6	Valencia/València	Albaida	6178 varones	3072 mujeres
7	Valencia/València	Albal	15594 varones	7864 mujeres
8	Valencia/València	Albalat de la Ribera	3609 varones	1826 mujeres
9	Valencia/València	Albalat dels Sorells	3859 varones	1886 mujeres
10	Valencia/València	Albalat dels Tarongers	1147 varones	601 mujeres
11	Valencia/València	Alberic	11331 varones	5723 mujeres
12	Valencia/València	Alborache	1263 varones	655 mujeres
13	Valencia/València	Alboraya	22563 varones	11262 mujeres
14	Valencia/València	Albuixech	3998 varones	1983 mujeres
15	Valencia/València	Alcàsser	9439 varones	4806 mujeres
16	Valencia/València	Alcàntera de Xúquer	1400 varones	686 mujeres
17	Valencia/València	Alzira	44752 varones	21836 mujeres
18	Valencia/València	Alcublas	807 varones	411 mujeres
19	Valencia/València	Alcúdia, l'	11432 varones	5596 mujeres

Una forma sencilla para un usuario sin conocimientos de bases de datos es almacenar información creando una hoja de cálculo de Microsoft Excel, que le permite almacenar e inspeccionar la información. El problema le llegaría a ese usuario cuando además de inspeccionar y manipular información sencilla pretendiese:

- Tener una gran cantidad de información
- Almacenar información compleja
- Tener que almacenar información repetida
- Dar acceso a distintos usuarios simultáneamente
- Asegurar que cierta información sea segura e inaccesible a ciertos usuarios
- Poder recuperar información anterior a la actual

Para poder crear una base de datos se necesita definir primero las tablas, que están compuestas por filas, que a su vez están formadas por atributos. Para empezar con una tabla básica se necesitan tres cosas:

- Número de columnas para guardar los atributos
- Tipo de dato para asignar a cada columna
- Manera de diferenciar las distintas filas

El orden de las filas no es importante en una base de datos, a diferencia de una hoja de cálculo, y por tanto una consulta devolverá la información en un orden independiente de su posición física.

-Elección de columnas:

De igual forma a como se haría en la hoja de cálculo se elije la información que se desea guardar. En un ejemplo de una tabla 'cliente', las posibles columnas serían 'nombre', 'apellidos', 'dirección', 'ciudad', 'teléfono',…. Una diferencia con la hoja de cálculo es que el número de columnas en una base de datos debe ser la misma en todas las filas.

-Elección de tipo de datos para cada columna:

Hay que determinar qué tipo de información va en cada columna. Mientras la hoja de cálculo permite tener en cada celda un tipo de información, en una tabla de base de datos cada columna debe tener el mismo tipo. Igual que en los lenguajes de programación las bases de datos usan tipos para clasificar diferentes valores. La mayoría de las veces se necesitan tipos básicos como números enteros, números en coma flotante, texto de longitud fija, texto de longitud variable, y fechas. Normalmente es fácil determinar el tipo de dato viendo ejemplos de la información a guardar, pero a veces no es tan claro, como puede ser guardar un número de teléfono, al que se le podría asignar un tipo numérico fijo pero entonces no se permitiría guardar teléfonos internacionales que incluyen el símbolo '+'.

-Identificación de filas únicas:

De alguna forma se tiene que poder identificar cada fila porque es la forma con la que una base de datos crea las relaciones con las demás tablas, por tanto hay que decidir qué es lo que hace una fila distinta de las otras para que la base de datos esté correctamente creada. Una solución sería usar un nombre (por ejemplo, el nombre del cliente), pero habría que asegurarse que no hay dos filas con el mismo nombre. Además la identificación de la fila debe ser independiente de la información, para que sea más estable. Por ejemplo se podría escribir incorrectamente el nombre, y tener que modificarlo posteriormente. La solución más estándar es asignar un número único a cada fila, que será el que identifica la fila, o PK ('primary key') como se le llama en base de datos. Esta solución es tan común que en PostgreSQL tiene un tipo de datos separada de los numéricos, llamada 'serial'.

TIPOS DE DATOS

PostgreSQL tiene un variado número de tipos de datos además de los básicos. Tiene algunos muy especializados y otros de uso interno. En casi todos los casos de tipos estándares, PostgreSQL los nombra tomando el nombre estándar de SQL. La razón de por qué las bases de datos usan tipos de datos son varias:

- Crea consistencia: Las columnas con un tipo uniforme produce resultados consistentes, y evita conflictos sobre cómo visualizar datos de distinto tipo.
- Permite validar datos: Las columnas con un tipo uniforme sólo aceptan datos de ese formato, y rechaza cualquier otro.
- Almacena de forma compacta: Al conocer el tipo de información, la ejecución de los procesos es más rápida.

-Tipos lógicos o booleanos:

Son el tipo de datos más sencillo con únicamente dos posibles valores conocidos: 'true' (verdadero) y 'false' (falso), o NULL en caso de valor desconocido. La declaración del tipo se realiza con la palabra 'boolean' o 'bool'. Cuando un dato se inserta en una columna booleana PostgreSQL es bastante flexible sobre lo que interpreta como 'true' o 'false', indistintamente en minúsculas o mayúsculas:

- Valores posibles para 'true': '1', 'yes', 'y', 'true' y 't'
- Valores posibles para 'false': '0', 'no', 'n', 'false' y 'f'

Nombre	Descripción
boolean **bool**	Almacena un valor de cierto o falso Usa 1 byte de almacenamiento

-Cadena de caracteres:

Son posiblemente el tipo de datos más usado en cualquier base de datos. Hay tres variantes, según se quiera representar:

- Un único carácter
- Cadena de caracteres con un tamaño fijo
- Cadena de caracteres con un tamaño variable

Además PostgreSQL incluye un tipo de dato 'text', poco estándar pero que no necesita declarar un límite de tamaño.

Nombre	Descripción
char **carácter** **bpchar**	Un único carácter
char(n) **bpchar(n)**	Un conjunto de exactamente n caracteres de largo, rellenado con espacios si se le asigna una palabra más pequeña
varchar(n) **carácter varying (n)**	Un conjunto de de hasta n caracteres
text	Un conjunto de ilimitados caracteres. Es una variante de PostgreSQL para varchar donde no se requiere limitar el número de caracteres

Para la elección de uno u otro tipo hay que conocer los datos que se van a almacenar en la columna:

- Aunque el tipo 'text' es el menos restrictivo, es también el menos estándar a la hora de exportar la información a otra base de datos
- El tipo 'char(n)' se usa cuando la longitud de los datos es fija o varía muy poco. Este tipo existe porque en algunas bases de datos el almacenamiento interno de cadenas de longitud fija es más eficiente
- El tipo 'varchar(n)' se usa cuando la longitud puede variar mucho y puede aceptar muchos tipos de palabras

Aunque la mayoría de las veces la elección de un tipo u otro es algo subjetivo y se puede implementar de distintas formas correctamente, normalmente cuando no se está seguro de los datos que se almacenarán se suele usar 'varchar(n)'. Igual que los booleanos, los caracteres pueden contener información sin rellenar, o lo que es lo mismo valor NULL, a no ser que la columna esté expresamente definida como NOT NULL. Si en el momento de crear una tabla se le señala un valor por defecto a una columna, al querer insertar una fila sin valor para esa columna se insertará ese valor por defecto en vez de NULL.

Los caracteres y cadenas se separan por comillas simples ('ejemplo'), y para incluir una comilla dentro de la cadena, hará falta poner doble comilla ('O''Donnell').

-Números:

Son un tipo de datos más complejo que los anteriores pero no son difíciles de comprender. Se distinguen dos tipos de números que se pueden almacenar en una base de datos:

- Enteros
- Números en coma flotante

Los enteros se subdividen en un subtipo especial de enteros, el tipo 'serial', y en diferentes tamaños de los enteros. Los números en coma flotante también se subdividen en los que ofrecen valores en general y los que ofrecen números con precisiones fijas.

Nombre	Descripción
small integer **smallint** **int2**	Entero de 2 bytes capaz de almacenar números desde –32768 hasta +32767
integer **int** **int4**	Entero de 4 bytes capaz de almacenar números desde –2147483648 hasta 2147483647
big integer **bigint** **int8**	Entero de 8 bytes capaz e almacenar aproximadamente enteros de 18 dígitos de precisión
bit	Bit: 0 o 1
bit varing **varbit(n)**	Cadena de n bits
serial	Entero automáticamente generado por PostgreSQL
real **float4**	Número en coma flotante de precisión simple de 4 bytes
double precisión **float8**	Número en coma flotante de doble precisión (8 bytes)
numeric(p,s)	Número real con p dígitos de los cuales s números son decimales. A diferencia de float siempre es un número exacto, pero menos eficiente
money **numeric(9,2)**	Tipo específico de PostgreSQL actualmente obsoleto

Distinguir una columna entre entero y número en coma flotante es sencillo, pero distinguir entre los distintos enteros, o entre los números en coma flotante es más complejo. Los 'float' se almacenan con notación científica con una mantisa y un exponente, mientras que el tipo 'numeric' permite especificar tanto la precisión como el número exacto de dígitos que se almacenan.

-Temporales:

Son los que almacenan la información sobre el tiempo, con un rango de fecha y momento del día. El problema de este tipo de dato en cualquier base de datos es codificar la información temporal, tanto a la hora de almacenarlo como a la hora de obtenerlo en una consulta. Hay dos cosas que controla el usuario:

- El orden en el que se manejan los días y meses (estilo estadounidense, estilo europeo,…)
- El formato de visualización (sólo numérico, con meses en texto en inglés,…)

PostgreSQL se basa por defecto en la ISO-8601 para mostrar los temporales (YYYY-MM-DD hh:mm:ss.ssTZD) almacenando el año, el mes, el día, las horas, los minutos, los segundos, la parte decimal de los segundos con dos dígitos, y una zona horaria (horas de diferencia entre la hora local y la hora UTC). Un ejemplo sería '2005-02-01 05:23:43.23+5' para representar el 1 de febrero de 2005, a las 5 de la mañana, 23 minutos, 43,23 segundos, en la zona horaria UTC+5. Si se escribe la fecha como NN/NN/NNNN, PostgreSQL interpreta el mes antes que el día (estilo estadounidense), y por tanto 2/1/05 es el 1 de Febrero. El estilo por defecto está controlado en el fichero 'postgresql.conf', que incluye una línea como la siguiente, que se ha de modificar para cambiar el estilo:

```
datestyle = 'iso, mdy'
```

Nombre	Descripción
date	Fecha desde el año 4713 AC hasta 1465001 DC, con precisión de 1 día
time	Momento del día (hora, minuto, segundo), desde 0 hasta 23:59:59.99 con precisión de 1 milisegundo
timestamp **datetime**	Fecha y momento del día, desde el año 4713 AC hasta 1465001 DC, con precisión de 1 milisegundo
interval	Tiempo de duración de aproximadamente +/- 178.000.000 años con precisión de 1 milisegundo
timestampz	Extensión de PostgreSQL que añade la zona horaria al tipo timestamp

-Tipos de datos especiales:

Desde los orígenes de PostgreSQL cuando tenía un uso científico universitario, su base de datos ha adquirido tipos de datos poco usuales para almacenar tipos geométricos y de red. El uso de alguno de estos tipos poco estándares hace de la base de datos poco portable a otros entornos.

Nombre	Descripción
point	2 números geométricos
line	Conjunto de 2 puntos

lseg	Segmento de una línea a partir de 2 puntos
box	Caja rectangular a partir de 2 puntos
path	Línea geométrica abierta a partir de una secuencia de puntos
polygon	Línea geométrica cerrada a partir de una secuencia de puntos
circle	Un círculo a partir de un punto y una longitud
serial	Columna numérica en una tabla que se incrementa cada vez que se añade una fila
oid	Objeto identificador interno para PostgreSQL, que añade un valor a cada fila y almacena un entero de 4 bytes (limitándolo a 4 mil millones).
cidr	Dirección de red con formato x.x.x.x/y, donde y es la máscara de red
inet	Dirección IP v4
macaddr	Dirección MAC con formato XX:XX:XX:XX:XX:XX

-Vectores:

Otra tipo de datos poco usual en otros entornos es el de los vectores. PostgreSQL permite almacenar vectores en las tablas, lo que es útil para almacenar un número fijo de elementos repetidos.

Hay dos sintaxis para crear vectores:

- El original de PostgreSQL: Para declarar una columna como un vector simplemente se añade '[]' después del tipo, sin tener que declarar el número de elementos (se permite declarar un tamaño pero PostgreSQL no dará error si se supera).

```
días_semana int[]
```

- El estándar SQL: En el estándar SQL99 se introdujo una sintaxis para declarar vectores. Es más explícito que el anterior y el número de elementos se tiene que declarar.

```
días_semana int array[7]
```

-Conversión de tipos:

En una base de datos en ocasiones es necesario y útil convertir un tipo de datos a otro, como por ejemplo cuando se trabaja con fechas como caracteres y se quieren almacenar como tipo fecha en PostgreSQL. PostgreSQL permite la conversión con un casting:

```
cast(columna AS nuevo_tipo)
```

O también con dobles dos puntos:

```
columna::nuevo_tipo
```

BUEN DISEÑO DE BASES DE DATOS

-Comprendiendo el contexto:

El primer paso para diseñar una base de datos es entender el contexto, el área en el que se encuentra, antes de meterse en los aspectos técnicos del diseño. Si el sistema va a sustituir otro ya existente será importante captar la información que se tenía y remplazarla si se cree

conveniente para el nuevo sistema. Es importante también conocer la situación de los usuarios que utilizarán la base de datos, y sus conocimientos técnicos. Para el diseño nuevo de una base de datos, posiblemente serán los usuarios los que expliquen al diseñador de la base de datos la información y la estructura de la información. Igualmente el diseñador tendrá que enfocarse en el 'qué' y no en el 'cómo' para entender el propósito general del sistema.

-Aspectos iniciales del diseño:

Es importante dejar claro cuál es el objetivo del diseño de la base de datos, porque diferentes objetivos llevarán a cabo distintos sistemas. En este paso se deben conseguir aclarar varios aspectos:

- Capacidad de almacenar datos:
  La capacidad de almacenar los datos es la base de todas las bases de datos, aunque con distintos niveles para cada caso. Para un diseño complejo de una base de datos que irá evolucionando con el tiempo se tendrá que dar prioridad a los requerimientos e implementaciones necesarias, y se dejará en un segundo plano la presentación visual.

  El diseño de una base de datos normalmente va evolucionando desde un diseño inicial a través de correcciones y mejoras del diseño. Conforme se vayan usando más datos reales y más cercanos a un diseño completo, la estructura del diseño va siendo más compleja y con tiempos mayores.

  En la mayoría de los diseños de bases de datos, incluso en las más complejas, únicamente el 25% de las tablas son fundamentales en el diseño final. Identificar y diseñar esas tablas es el primer objetivo del diseñador. Las tablas restantes son importantes por la información que contienen, pero no lo son para la base del diseño.

- Capacidad de dar soporte a las relaciones:
  El diseño de la base de datos debe dar soporte a las relaciones entre las entidades, y por eso es importante tenerlas en cuenta además de los datos a almacenar. Una aplicación que utiliza un diseño con todos los datos, pero deja de lado estas relaciones, con el tiempo tendrá problemas de integridad de la información, además de una complejidad excesiva del sistema.

- Capacidad para cumplir los objetivos iniciales:
  Un sistema bien diseñado no tiene valor si no consigue dar solución a los objetivos que se marcaron cuando se decidió crear una base de datos. Durante el proceso del diseño hay que tener muy en cuenta estos objetivos.

- Capacidad para imponer la integridad de los datos:
  Los aspectos de la integridad de los datos están relacionados con los puntos anteriores. El objetivo global de la base de datos es almacenar datos, y la calidad de éstos es importante para los diseñadores. Los datos reales tienen deficiencias: datos desconocidos, palabras

escritas a mano difíciles de leer, información incompleta,… y aunque estas deficiencias afectarán al diseño, no deben interferir en la calidad de los datos.

El diseñador debe elegir los tipos de datos cuidadosamente, crear restricciones de columnas, y si es necesario crear triggers para mantener los datos en la base de dato con todo el rigor que razonablemente se pueda. En ocasiones el diseñador tendrá que usar el sentido común y su experiencia, pero nunca debe inventar datos ni dar por supuesto nada.

- Capacidad para imponer la eficiencia de los datos:
  La eficiencia de los datos es un aspecto difícil en el diseño de la base de datos, porque cuánto más grande y compleja sea la base de datos, más se tiene que tener en cuenta los aspectos de rendimiento, que pueden perjudicar la eficiencia. Por eso el diseñador debe tener en cuenta primero el diseño óptimo, y después si es necesario modificarlo para optimizarlo en términos de rendimiento.

- Capacidad para adaptarse a futuros cambios:
  En bases de datos las migraciones de un diseño viejo a otro nuevo no es una práctica muy seguida, y se suele preferir adaptar el diseño a los cambios antes que comenzar de nuevo

-Etapas en diseño de base de datos:

El diseño de una base de datos no es un aspecto técnico, sino de comprender las necesidades y las expectativas de los usuarios. Después de obtener la información se puede pasar al diseño lógico.

- Recopilando información:
  La primera etapa en el diseño de una base de datos es recopilar información sobre qué se espera de ella, y poder marcar un requerimiento inicial. A partir de este requerimiento inicial se tiene que ampliar a otros requerimientos básicos necesarios y a lo que el sistema necesita.

  Además de los requerimientos, es útil observar datos reales que se usarán, y hacer un estudio sobre las relaciones y necesidades

- Desarrollo del diseño lógico:
  Esta etapa se centra en la estructura lógica de la base de datos, y obvia detalles de implementación. Una vez recopilada la información sobre los requerimientos iniciales, el diseñador debe reconocer las principales entidades (los objetos clave que aparecerán en la base de datos). Y una vez determinadas estas entidades, se pasa a identificar los atributos que cada entidad necesita. En este punto no hay que fijarse en cómo se debe almacenar los datos u otros aspectos complejos, simplemente se identifican entidades y atributos.

  Una vez identificadas las entidades, el diseñador ya puede empezar con aspectos más técnicos. Lo primero es nombrar a las entidades de forma que queden claras e

inconfundibles. Después hay que hacer lo mismo con los atributos. Una elección correcta de los nombres ayudará a la consistencia de los datos.

-Determinar relaciones y cardinalidades:

En este punto el diseñador tiene una lista incompleta con las principales entidades. A partir de las entidades se tiene que determinar qué atributos van a ocurrir varias veces para cada entidad, y también se tiene que decidir cómo se relacionan las distintas entidades.

- Relaciones:

  Una forma útil de determinar las relaciones entre entidades es representarlas gráficamente. Esta representación de la estructura es un modelo conceptual de la base de datos donde se muestran las tablas y las relaciones entre ellas. En las relaciones entre dos tablas son en dos sentidos, y en el modelo se marcan indicando la cardinalidad en los dos sentidos.

Centrándose en 'relación_1', los posibles valores son:

Relación	Nombre	Descripción
Cero o uno	0, 1	Cada fila de la tabla B tiene relación con cero o una fila de la tabla A
Exactamente uno	1	Cada fila de la tabla B tiene relación con exactamente una fila de la tabla A
Cero o más	0 .. n	Cada fila de la tabla B tiene relación con 0, 1 ó más filas de la tabla A
Uno o más	1 .. n	Cada fila de la tabla B tiene relación con 1 ó más filas de la tabla A

Aunque en algunos casos también se pueden alterar los números (por ejemplo '3 .. 5' si en la tabla A existiesen siempre entre 3, 4 o 5 filas relacionadas).

- Diseño conceptual:

  Tras establecer todas las relaciones entre las entidades se consigue un diseño conceptual inicial, que habrá que comprobar que sea correcto porque un error en este paso será difícil de corregir posteriormente.

-Convertir en un diseño físico:

A partir del modelo lógico anterior el diseñador debe iniciar el diseño de la representación física del modelo, que también se puede dividir en distintos pasos:

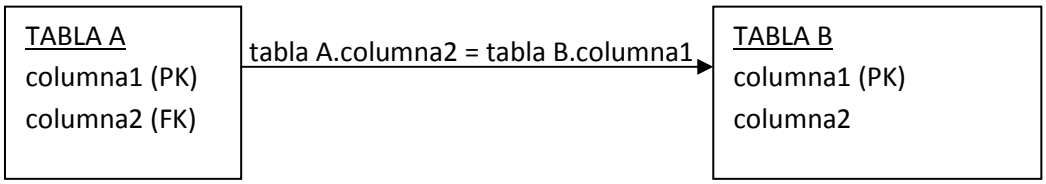
- Establecer claves primarias (PK):

  El primer paso es establecer las claves primarias, viendo las tablas una a una y considerando qué dato o datos son únicos en cada fila, si lo hay, y además siempre es un valor conocido.

Una vez se tiene se elige como clave primaria, y si no se consigue encontrar uno entonces se genera una clave primaria lógica, por ejemplo enumerando cada fila.

- Establecer claves ajenas (FK):
  Después de identificar las claves primarias hay que establecer las relaciones entre las tablas, y con el modelo conceptual del apartado anterior es una tarea bastante sencilla. Para establecer claves ajenas lo más sencillo es asegurarse que la columna (o columnas) que están en una tabla identificadas como clave primaria también aparecen en todas las tablas relacionadas directamente con esa tabla.

  Las claves ajenas se pueden marcar en un diagrama de la siguiente forma:



  Desde el diagrama del conceptual se está desarrollando el modelo físico, mostrando la información de cómo se relacionan las tablas físicamente, no sólo la cardinalidad. Además se señala qué columna es clave primaria.

- Establecer tipos de datos:
  Una vez establecidas las tablas, las columnas y las relaciones, se puede seguir con los tipos de datos para cada columna. Juntamente con los tipos de datos se tiene que establecer qué datos podrán tomar valor NULL, y por tanto marcar el resto como NOT NULL.

  Las columnas que forman parte de alguna clave primaria o alguna clave ajena deberían ser de algún tipo nativo de datos, como integer, para que sea fácil procesarlas posteriormente.

- Implementar el esquema:
  Una vez establecidas las definiciones de las tablas, las columnas, las relaciones y los tipos de datos, y revisado todo nuevamente para ver que todo es correcto, ya se puede pasar a generar el código SQL para generar el esquema de la base de datos. Es el momento de plantearse otras restricciones, o incluso triggers que hagan falta para mantener la base de datos.

- Comprobar el diseño:
  Una vez implementada la base de datos, llega el momento de testearla con datos de ejemplo que se ajusten a los reales, y comprobar que se cumplen las relaciones, las restricciones,…. Es aconsejable crear consultas en SQL con varias tablas para ver las relaciones.

-Formas normales:

Existen unos estándares de normalización del diseño de las bases de datos llamados formas normales, divididos en varios niveles. Cada forma normal se construye sobre la anterior y aplica requerimientos más estrictos al diseño.

- Primera forma normal:
  La primera forma normal requiere que cada atributo en una tabla no pueda ser subdividido y que no haya grupos repetidos. Un ejemplo de lo primero es separar en dos columnas el nombre y el apellido de una persona. Si existen grupos repetidos en una tabla es porque la tabla no está bien definida y posiblemente las columnas que crean esas repeticiones deban formar su propia tabla.

- Segunda forma normal:
  La segunda forma normal requiere que no haya información en una fila que dependa únicamente de una parte concreta de la clave primaria. Esa información no identifica a la fila, y por tanto debe almacenarse independientemente en otra tabla.

- Tercera forma normal:
  La tercera forma normal sólo generaliza la anterior, y requiere que todos los datos de las columnas que no son la clave primaria únicamente dependen de la clave primaria.

No siempre es posible ajustar un diseño a la tercera forma normal, muchas veces por mejorar el rendimiento, pero es aconsejable normalizarlo lo máximo posible. Existen además dos niveles más de formas normales, pero son estas tres las más extendidas.

LIMITACIONES DE POSTGRESQL

Todas las bases de datos están limitadas de una u otra forma, y PostgreSQL no es una excepción. La cantidad de datos que se puede almacenar en una columna, el máximo número de columnas permitido en una tabla, el tamaño total de cualquier tabla,… son límites que siempre existen.

Cuanto más cerca se encuentre una base de datos de sus limitaciones, peor será el rendimiento de la misma. Si por ejemplo se manejan campos muy grandes que consumen mucho tiempo de memoria al procesarlos, el rendimiento de PostgreSQL irá bajando hasta llegar a un punto en que no podrá modificar los datos.

En las últimas versiones de PostgreSQL se aceptan limitaciones más extensas, e incluso se han eliminado algunas. En este apartado se listan las principales limitaciones de PostgreSQL en su versión 8.0. Cuando un tamaño es 'ilimitado' quiere decir que PostgreSQL no impone un límite, pero siempre hay que tener en cuenta el sistema operativo, el servidor, la cantidad de disco duro disponible, la red y otros factores ajenos al código de PostgreSQL.

-Tamaño de base de datos:

Ilimitado.

PostgreSQL no impone un límite en el tamaño total de una base de datos, e incluso existen pruebas con bases de datos de 4 TB. Debido a la forma en que PostgreSQL maneja sus datos, el rendimiento se irá degradando conforme más tablas existan en la base de datos. PostgreSQL puede usar muchos ficheros para almacenar los datos, y el rendimiento puede empeorarse si el sistema operativo no es capaz de trabajar de forma óptima con muchos archivos en un único directorio. Desde que PostgreSQL incorporó los tablespaces a sus tablas, el rendimiento con grandes bases de datos es menos costoso.

-Tamaño de tabla:

16 TB – 64 TB

PostgreSQL almacena los datos normales de las tablas en bloques de 8 KB, y el número de estos bloques está limitado a un número entero con signo de 32 bits (poco más de dos mil millones), dando un tamaño máximo de tabla de 16TB. El tamaño del bloque básico se puede incrementar una vez instalado PostgreSQL hasta un máximo de 32 KB, dando un límite de la tabla de 64 TB.

Algunos sistemas operativos imponen un limite te tamaño de archivo que impiden crear archivos tan grandes, por eso PostgreSQL almacena los datos en múltiples archivos, cada uno de 1 GB de tamaño. Para grandes tablas se crearán muchos archivos y eso hará que el sistema operativo degrade el rendimiento de la base de datos.

-Filas en una tabla:

Ilimitado

PostgreSQL no impone un límite en el número de filas máximo en una tabla.

-Índices en una tabla:

Ilimitado

PostgreSQL no impone un límite en el número de índices que se pueden crear sobre una tabla. Como se vio en el apartado de índices, crear muchos empeora el rendimiento.

-Tamaño de columna:

1 GB

PostgreSQL tiene un límite de 1 GB para el tamaño de cualquier campo en una tabla. En la práctica el tamaño viene por la cantidad de memoria que se dispone en el servidor para manipular datos y transferirlos al cliente.

-Columnas en una tabla:

Al menos 250

El máximo número de columnas en una tabla para PostgreSQL depende en la configuración del tamaño del bloque y el tipo de columna. Para el tamaño por defecto del bloque (8 KB) al menos se pueden almacenar 250 columnas. Pero este número puede aumentarse hasta 1.600 si todas las columnas son muy sencillas (por ejemplo enteros).

-Tamaño de fila:

Ilimitado

No hay un tamaño máximo explícito de una fila, pero el número máximo de columnas sí que limitará el total de tamaño de la fila.

-Conexiones:

Limitado por el servidor.

Por defecto se encuentra a 100, pero se este valor está guardado en la variable 'max_connections' de 'initdb'. Cada conexión usa una pequeña cantidad de memoria compartida, así que en sistemas limitados en memoria no se permitirá un valor alto.

BIBLIOGRAFÍA

- [1] "PostgreSQL Introduction and Concepts"
- [2] "Beggining Databases with PostgreSQL: From Novice to Professional, Second Edition"
- [5] http://wiki.postgresql.org/wiki/Help:Variables
- [6] "PostgreSQL 9.0 High Performance"

TEMA 6 – PROGRAMACIÓN

PROPIEDADES DEL LENGUAJE POSTGRESQL

Aunque SQL no es un lenguaje procedural, PostgreSQL añade algunas funcionalidades típicas de otros lenguajes de programación como son sentencias condicionales (IF-THEN-ELSE, CASE-WHEN-ELSE,…), funciones (abs, _varchar2, int2, upper, sqrt,…), operadores (+, -, *, /, %, ^, !,…). PostgreSQL también añade funcionalidades de resumen de la información como son COUNT para contar las filas que cumplen una condición, AVG para calcular la media, SUM para sumar números, MAX para calcular el valor máximo entre varios (numérico o alfanumérico), MIN para calcular el valor mínimo.

IDENTIFICADOR DE FILA OID

-Identificador OID:

Cada vez que se insertan datos en una tabla, PostgreSQL muestra un número que es la referencia interna de la nueva fila insertada, un identificador que PostgreSQL almacena en cada fila como una columna oculta llamada 'oid'. La mayoría de bases de datos no tienen este identificador, o si lo tienen no es accesible a los usuarios. Pero en PostgreSQL se puede consultar si en la SELECT se especifica la columna 'oid'.

```
SELECT oid, nombre FROM cliente;
```

En cada fila de una tabla habrá valores únicos. Además la columna 'oid' también es opcional en la configuración del controlador ODBC.

-Secuencias:

PostgreSQL ofrece otra forma de numerar filas de forma única, usando secuencias, que son contadores creados por los usuarios e identificados por un nombre. Después de crear una secuencia se le puede asignar a una tabla como una columna DEFAULT. Al usar secuencias se pueden asignar valores automáticamente durante una inserción.

```
CREATE SEQUENCE nombre_secuencia;
```

Las secuencias nuevas tienen su contador con valor 0, y tienen que asignarse a la tabla para que funcione como identificador.

```
CREATE TABLE cliente (
        cliente_id INTEGER DEFAULT nextval('nombre_secuencia'),
        nombre CHAR(30)
);
```

Función	Acción
nextval('nombre')	Devuelve el siguiente valor de la secuencia, y actualiza el contador
currval('nombre')	Devuelve el número de secuencia devuelto en la última llamada a nextval()
setval('nombre', nuevo_valor)	Ajusta el valor de la secuencia a un valor específico

Las ventajas de usar secuencias son que éstas no asignan números a filas no válidas, y además son visibles y manejables por los usuarios.

OPERADORES

En las consultas hechas con SELECT, a veces es necesario introducir operadores para tratar la información. Un ejemplo sencillo es obtener las compras de más de 10 euros, donde es necesario introducir el operador comparador '>' entre el atributo y un número fijo:

```
SELECT * FROM articulo WHERE precio > 10;
```

Hay un gran número de operadores soportados por PostgreSQL, de hecho si se consideran distintas las distintas versiones del mismo operador (un comparador de enteros es diferente de un comparador de reales), hay sobre 600 operadores disponibles.

-Prioridad de operador y asociatividad:

Muchos de los operadores en PostgreSQL actúan como los operadores aritméticos normales igual que en muchos lenguajes de programación. Los operadores tienen una prioridad no modificable en el analizador que determina el orden en que se ejecutan los operadores en expresiones compuestas. Las prioridades se pueden establecer con paréntesis. Además PostgreSQL permite usar operadores fuera del WHERE en una sentencia de consulta:

```
SELECT 1+2*3;
```

Aunque la mayoría de las veces el comportamiento de los operadores es el mismo que en un lenguaje de programación, en ciertos casos la prioridad entre operadores no es tan intuitiva. Igual que en el lenguaje de programación C, los operadores sobre booleanos tienen menos prioridad que los operadores aritméticos, por lo que los paréntesis ayudan a ordenar la ejecución de los distintos operadores en caso de duda.

En PostgreSQL los operadores también muestran asociatividad, de derecha o de izquierda, que determina el orden en que los operadores de igual prioridad se evalúan. Los operadores aritméticos como suma y resta tienen asociatividad izquierda (en "1 + 2 - 3" se evalúa como si se hubiese escrito "(1 + 2) - 3"). Otros operadores como los booleanos tienen asociatividad derecha (en "x = y = z" se evalúa si se hubiese escrito "x = (y = z)".

La siguiente tabla muestra la prioridad léxica (en orden descendiente) de los operadores más comunes en PostgreSQL:

Operador	Asociatividad	Descripción
:: **[]**	Izquierda	Conversión de tipo (sinónimo de CAST) Selección de vector
.	Izquierda	Selección de objeto (esquema, tabla, columna)
-	Derecha	Menos unario (negación de entero)
^	Izquierda	Exponenciación
*** / %**	Izquierda	Operadores multiplicativos
+ -	Izquierda	Operadores de adición
IS **ISNULL** **NOTNULL** **OR**	Izquierda	Test (para TRUE, FALSE, UNKNOWN y NULL) Test (para NULL) Test (para non-NULL) Disyunción lógica
IN **BETWEEN** **LIKE ILIKE SIMILAR** **<>** **=**	Derecha	Test para miembro de un conjunto Test para inclusión en un rango Test para coincidencia de cadenas Test para desigualdad Test para igualdad
NOT	Derecha	Negación lógica
AND	Izquierda	Conjunción lógica
Otros…		Operadores integrados y definidos por usuario que no están listados, tienen la misma prioridad

-Operadores aritméticos:

PostgreSQL proporciona una gran variedad de operadores aritméticos. Los más comunes están en la lista siguiente y todos tienen la misma prioridad y son asociativos por la izquierda:

Operador	Ejemplo	Descripción
+	2 + 5 => 5	Adición
-	3 – 2 => 1	Sustracción
*****	2 * 3 => 6	Multiplicación
/	3 / 2 => 1 3 / 2.0 => 1.5 3 / 2 ::float => 1.5	División
%	22 % 7 => 1	Resto (módulo)
^	4 ^3 => 64	Elevado a la potencia (exponenciación)
&	14 & 23 => 6	AND binario
|	14 | 23 => 31	OR binario
#	14 # 23 => 25	XOR binario
>>	128 >> 4 => 8	Desplazamiento a la derecha
<<	1 << 4 => 16	Desplazamiento a la izquierda

También hay varios operadores aritméticos unarios:

Operador	Ejemplo	Descripción
%	%2.3 => 2	Truncamiento
!	4! => 24	Factorial

!!	!!4 => 24	Factorial como operador izquierdo
@	@(-2) => 2	Valor absoluto
|/	|/ 64 => 8	Raíz cuadrada
||/	||/64 => 4	Raíz cúbica
~	~15 => -16	NOT binario

En general los operadores aritméticos en PostgreSQL funcionan como se espera de ellos. Se usará en cada caso la versión que encaje con los argumentos, por lo que al dividir un número entero el resultado es un número entero, y al dividir un número real el resultado es un número real.

-Operadores de comparación y de cadena:

PostgreSQL proporciona el conjunto básico de operadores de comparación tales como el "menor que" o el "mayor que". Estos operadores se pueden usar en muchos de los tipos de datos, incluidos los tipos alfanuméricos (compara dos cadenas alfabéticamente). El resultado de una comparación es 'true' o 'false'.

Operador	Ejemplo (con valor 'true')	Descripción
<	2 < 3 'axy' < 'azz'	Menor que
<=	2 <= 3	Menor o igual que
<> **!=**	2 <> 3 2 != 3	Distinto que
=	3 = 1 + 2	Igual que
>	3 > 2	Mayor que
>=	3 >= 2	Mayor o igual que

Las cadenas alfanuméricas tienen su propio conjunto de operadores en PostgreSQL. Hay operadores para concatenar cadenas y para validar patrones:

Operador	Ejemplo	Descripción
||	'abc' || 'def' => 'abcdef'	Concatenación de cadenas
~~	'xyzzy' ~~ '%zz%'	Sinónimo de LIKE
!~	'xyzzy' !~~ '%aa%'	Sinónimo de NOT LIKE
~	'xyzzy' ~ 'y.*y'	Encaja la subcadena de la expresión regular
~*	'xyzzy' ~* '^X.*Y$'	Encaja la expresión regular, sin diferenciar mayúsculas de minúsculas
!~	'xyzzy' !~ 'aa'	No encaja (el inverso de ~)
!~*	'xyzzy' !~* 'AA'	No encaja, sin diferenciar mayúsculas de minúsculas (el inverso de ~*)

Además existe otra forma de comparar cadenas, con el comparando LIKE. Sirve para comparar cadenas y ver si cumplen un patrón, donde el valor '%' se sustituye por cualquier cadena (o cadena vacía) y el valor '_' se sustituye por un único carácter de cualquier valor.

Operador	Descripción
LIKE 'D%'	La cadena empieza con 'D'
LIKE '%D%'	La cadena contiene 'D'
LIKE '_D%'	La cadena tiene una 'D' en la segunda posición
LIKE 'D%e%'	La cadena empieza con 'D' y contiene 'e'
LIKE 'D%e%f%'	La cadena empieza con 'D', contiene 'e', luego contiene 'f'
NOT LIKE 'D%'	La cadena no empieza con 'D'

-Operadores de tiempo:

Operador	Descripción
x + y	Suma de los valores temporales x e y
x – y	Resta de los valores temporales x e y
(x, y) OVERLAPS (z, w)	Booleano indicando si el intervalo de tiempo entre x e y se solapa con el intervalo de tiempo entre z y w

-Operadores de red:

Operador	Descripción
x << y	Booleano indicando si x es una subred de y
x <<= y	Booleano indicando si x es igual o es una subred de y
x >> y	Booleano indicando si x es una supernet de y
x >>= y	Booleano indicando si x es igual o es una supernet de y

-Expresiones regulares:

Las expresiones regulares permiten comparaciones más potentes que LIKE, y es una característica de PostgreSQL que no ofrecen otros sistemas. Un ejemplo es:

```
SELECT * FROM cliente WHERE nombre ~* '^[PR].*E$';
```

Expresión	Descripción
^	Comienzo de la cadena
$	Fin de la cadena
.	Cualquier carácter
[ccc]	Conjunto de caracteres
[^ccc]	Conjunto de caracteres no iguales
[c-c]	Rango de caracteres
[^c-c]	Rango de caracteres no iguales
?	Cero o un carácter previo
*****	Cero o más caracteres previos
+	Uno o más caracteres previos
|	Operador OR

Operador	Descripción
~'^D'	La cadena empieza con 'D'
~'D'	La cadena contiene 'D'
~'^.D'	La cadena contiene 'D' en la segunda posición
~'^D.*e'	La cadena empieza con 'D' y contiene 'e'
~'^D.*e.*f'	La cadena empieza con 'D', contiene 'e', y luego contiene 'f'

~'[A-D]' **~'[ABCD]'**	La cadena contiene 'A', 'B', 'C' o 'D'
~*'a' **~'[Aa]'**	La cadena contiene 'A' o 'a'
!~'D'	La cadena no contiene 'D'
!~'^D' **~'^[^D]'**	La cadena no empieza con 'D'
~'^?D'	La cadena empieza con 'D', con un espacio opcional al inicio
~'^*D'	La cadena empieza con 'D', con espacios opcionales al inicio
~'^+D'	La cadena empieza con 'D', con al menos un espacio
~'G*$'	La cadena termina con 'D', con espacios opcionales al final

-Otros operadores:

PostgreSQL soporta muchos más operadores para comparar y manejar los tipos de datos específicos de PostgreSQL como puntos, círculos, intervalos de tiempo y direcciones IP.

FUNCIONES INTEGRADAS

PostgreSQL cuenta con una larga lista de funciones integradas que se pueden usar dentro de las consultas SELECT. Los tipos de funciones son:

- Funciones equivalentes a los operadores de la sección anterior
- Otras funciones matemáticas
- Otras funciones para manejar cadenas de caracteres
- Funciones para manejar fechas
- Funciones para dar formato a texto
- Funciones para los tipos de datos de PostgreSQL (círculos, puntos,…)
- Funciones para direcciones IP

Las funciones integradas, y también las definidas por el usuario, se graban en una tabla del sistema de la base de datos de PostgreSQL llamada 'pg_proc', actualmente con más de 1.700 entradas.

-Cadena de caracteres:

Función	Descripción
length (x) **character_length (x)**	Longitud de x
octet_length (d)	Longitud de x, incluyendo los multibytes completos
trim (x), **trim (BOTH, x)**	Cadena x sin espacios iniciales o finales
trim (LEADING, x)	Cadena x sin espacios iniciales
trim (TRAILING, x)	Cadena x sin espacios finales
trim (x FROM y)	Cadena y sin los caracteres x iniciales o finales
rpad (x, y)	x rellenado con espacios por la derecha hasta los y caracteres
rpad (x, y, z)	x rellenado con z por la derecha hasta los y caracteres

lpad (x, y)	x rellenado con espacios por la izquierda hasta los y caracteres
lpad (x, y, z)	x rellenado con z por la izquierda hasta los y caracteres
upper (x)	x en mayúsculas
lower (x)	x en minúsculas
initcap (x)	x con formato título (primera letra de cada palabra en mayúscula)
strpos (x, y) **position (x IN y)**	posición y en la cadena x
substr (x, y) **substring (x FROM y)**	x desde la posición y
substr (x, y, z) **substring (x FROM y FOR z)**	x desde la posición y hasta los siguientes z caracteres
translate (x, y, z)	x con las cadenas y cambiadas por la cadena z
to_number(x, máscara)	x convertido en NUMERIC() basado en la máscara
to_date(x, máscara)	x convertido en DATE basado en la máscara
to_timestamp(x, máscara)	x convertido en TIMESTAPM basado en la máscara

-Números:

Función	Descripción
round (x)	Redondea al número entero más cercano
round (x, d)	Redondea al número con ‘d’ decimales más cercano
trunc (x)	Trunca a un entero
trunc (x, d)	Trunca al número con ‘d’ decimales
abs (x)	Valor absoluto
factorial (x)	Factorial de x
sqrt (x)	Raíz cuadrada
cbrt (x)	Raíz cúbica
exp (x)	Antilogaritmo natural, eleva ‘e’ a la potencia
ln (x)	Logaritmo natural
log (x)	Logaritmo natural en base 10
log (b, x)	Logaritmo de una base b
to_char (x, máscara)	Convierte x en una cadena de caracteres basado en la máscara
mod (x, y)	Resto tras dividir x entre y (también tiene una versión para enteros)
pi ()	Devuelve π
pow (x, y)	Eleva x a la potencia de y
random ()	Devuelve un número aleatorio entre 0.0 y 1.0
trunc (x)	Trunca al número entero (hacia cero)
ceil (x)	Devuelve el entero más pequeño no menor que ‘x’
floor (x)	Devuelve el entero más grande no mayor que ‘x’
float8 (i)	Toma un entero y devuelve un equivalente en ‘float8’
float4 (i)	Toma un entero y devuelve un equivalente en ‘float4’
int4 (x)	Devuelve un entero, redondeando si es necesario

-Temporales:

Función	Descripción
date_part (unidades, x) **extract (unidades FROM x)**	Partes de unidades en x
date_trunc (unidades, x)	x redondeado a unidades
isfinite (x)	Booleano indicando si x es una fecha válida
now ()	TIMESTAMP representando la fecha y momento del día actual
timeofday()	Cadena de caracteres con la fecha y momento del día actual en formato Unix
overlaps(x, y, z, w)	Booleano indicando si x, y, z y w se superponen en el tiempo
to_char (x, máscara)	fecha x en cadena de caracteres basado en la máscara

-Trigonométricas:

Función	Descripción
sin	Seno
cos	Coseno
tan	Tangente
cot	Cotangente
asin	Seno inverso
acos	Coseno inverso
atan	Tangente inversa
atan2	Arcotangente de dos argumentos, dado ‘a’ y ‘b’, computa atan(a / b)
degrees (r)	Convierte medidas angulares de radianes a grados
radians (d)	Convierte medidas angulares de grados a radianes

-Red:

Función	Descripción
broadcast (x)	Dirección lógica (o broadcast) de x
host (x)	Dirección de host de x
netmask (x)	Máscara de red de x
masklen (x)	Longitud de máscara de x
network (x)	Dirección red de x

-NULL:

Función	Descripción
nullif (x, y)	Si x es igual que y devuelve NULL, en otro caso devuelve x
coalesce (x, y, …)	Devuelve el primer argumento que sea distinto de NULL

Una importante función de formateado es ‘to_char’ que hace el mismo papel que ‘printf’ hace en C, manejando cualquier formateado de valores para imprimir o visualizar. Mostrará una fecha de acuerdo con la plantilla de fecha y puede formatear valores numéricos de muchas formas.

LENGUAJES PROCEDURALES

Además de las funciones integradas en PostgreSQL es posible que el usuario defina funciones para su uso dentro de una base de datos. Es útil para cálculos particulares o para cálculos que se necesitan realizar en distintos sitios. La sintaxis para crear una función es:

```
CREATE FUNCTION nombre ( [ ftipo [, … ] ] )
RETURNS tipo_que_devuelve
AS definición
LANGUAGE 'nombre_lenguaje'
```

La 'definición' de la función es un string (entre comillas simples) que puede ocupar distintas líneas y está escrito en cualquiera de los lenguajes que PostgreSQL soporta, especificado en 'lenguaje'. Un ejemplo es el siguiente, en el que se usa el lenguaje PL/pgSQL (un lenguaje de programación desarrollado específicamente para programar procedimientos para PostgreSQL).

```
CREATE FUNCTION suma_uno (int4 ) RETURNS int5 AS
'
BEGIN
                RETURN $1 + 1;
        END;
'
LANGUAGE 'plpgsql';
```

Para poder manejar un lenguaje, PostgreSQL debe primero tener una extensión para ese lenguaje, incluyendo una función controladora escrita en C. Para el lenguaje PL/pgSQL, el controlador está incluido como una librería. Cuando se crea una función, su definición se almacena en la base de datos y cuando se llama a la función por primera vez, se compila por el controlador creando un ejecutable. Esto quiere decir que cualquier error no se identifica hasta que se usa la función.

-PL/pgSQL:

En una instalación estándar de PostgreSQL, la función controladora de PL/pgSQL se incluye en la librería 'plpgsql.so' en la carpeta 'lib'. Cada base de datos de PostgreSQL en un servidor tiene su propia lista de lenguajes procedurales, y cuando se instala uno nuevo se tiene que elegir sobre qué base de datos se usará. Esto se hace por seguridad para evitar funciones que externamente se ejecuten accidentalmente o maliciosamente consumiendo recursos del servidor. Por eso PostgreSQL no viene con lenguajes instalados, y para usar PL/pgSQL se ha de instalar el controlador.

```
Createlang [options ] nombre_lenguaje nombre_base_de_datos
```

-Sobrecarga de funciones:

PostgreSQL considera funciones distintas cuando tienen nombres distintos, cuando tienen distinto número de parámetros, o si los tipos de los parámetros son distintos. En el ejemplo anterior, el 'suma_uno' tiene como parámetro de entrada 'int4' y si se quisiera usar con un

número real fallaría, a menos que se cree otra función similar con el mismo nombre pero con el parámetro de entrada con el tipo 'float8'.

-Borrado de funciones:

Las funciones se pueden eliminar de la base de datos con el comando DROP FUNCTION:

```
DROP FUNCTION suma_uno(int4);
DROP FUNCTION suma_uno(float8);
```

-Cadenas en funciones:

Un problema en los procedimientos es cuando se quiere introducir algo con comillas, ya que estas delimitan la definición de la función y no quedaría claro. La solución es escapar las comillas usando la comilla simple dos veces (''). Si dentro de la cadena hay comillas escapadas, habrá que escaparlas otra vez (''').

ANATOMÍA DE PROCEDIMIENTOS ALMACENADOS

PL/pgSQL es un lenguaje estructurado en bloques, que necesita declaración de variables. La sintaxis para crear un bloque es:

```
[<<etiqueta>>]
[DECLARE declaracionesdeclarations]
BEGIN
sentencias
END;
```

Y para crear una función con un bloque como su definición:

```
CREATE FUNCTION nombre ( [ ftipo [, ...] ] )
RETURNS tipo_que_devuelve
AS 'bloque definición'
LANGUAGE 'plpgsql';
```

-Argumentos de la función:

Una función PL/pgSQL puede tomar cero o más parámetros, y los tipos de los parámetros se señalan entre paréntesis después del nombre de la función. Los tipos son los que están integrados en PostgreSQL tales como 'int4' o 'float8'. Todos estos procedimientos almacenados deben devolver un valor, y el tipo de ese valor que devuelve es el especificado en la cláusula RETURNS.

Dentro del cuerpo de la función los parámetros de la función se referencian como $1, $2,… en el orden en el que están definidos. También es posible darles nombres usando la palabra clave ALIAS.

-Comentarios:

PL/pgSQL permite escribir comentarios dentro de la definición de la función. Si el comentario está en una única línea, se marca el inicio del comentario con dos guiones '—'. Y para comentar un bloque de varias líneas se marca el inicio con '/*' y el fin del comentario con '*/'.

```
--Comentario de una línea
/* Comentario de
varias líneas */
```

-Declaraciones:

Las funciones en PL/pgSQL permiten declarar variables locales para usarlas dentro de la función. Cada variable tiene un tipo que puede ser de los integrados por PostgreSQL, de los definidos por el usuario, o incluso una fila de una tabla. La declaración de variables se escribe dentro de la sección DECLARE de la definición de la función, o de un bloque dentro de la función. Las variables creadas en un bloque sólo son accesibles internamente en ese bloque.

```
DECLARE
        n1 integer;
        n2 integer;
BEGIN
        -- aquí se pueden usar n1 y n2
        n2 := 1;
        DECLARE
                n2 integer;
                -- se define un nuevo n2, por tanto el anterior no se accede en este bloque
                n3 integer;
        BEGIN
                -- aquí se pueden usar n1, n2 (el segundo) y n3
                n2 := 2;
        END;
        -- n3 ya no es accesible
        -- aquí n2 sigue valiendo 1
END;
```

Todas las variables deben estar declaradas antes de ser usadas (excepto las variables de control en un bucle, que son un caso especial). El nombre de una variable no puede ser una palabra reservada de PostgreSQL, que son las siguientes:

alias	assign	begin	close	constant	cursor
debug	declare	default	diagnostics	dotdot	else
elsif	end	exception	execute	exit	fetch
for	from	get	if	in	info
into	is	log	loop	next	not
notice	null	open	or	perform	raise
rename	result_oid	return	return_next	reverse	row_count
select	then	to	type	when	warning

```
while
```

-Alias:

La forma más simple de hacer una declaración en una función es mediante ALIAS, que le da un nombre a un parámetro en una posición dada. Esta declaración ayuda a entender el código mejor y por tanto más fácil de modificar. Sigue la siguiente sintaxis:

```
nombre ALIAS FOR $n;
```

Donde una nueva variable llamada 'nombre' está disponible, y su valor es el parámetro que se encuentra en la posición 'n'.

Otra forma de declarar alias automáticamente cuando se declara la función es:

```
create function nombre_función
(parámetro_uno integer,
 parámetro_dos integer)
returns float8 as $$
...
```

-Renombrado:

También es posible renombrar variables mediante la declaración RENAME, aunque debido a la posible confusión no es recomendado usarlo.

```
RENAME nombre_original TO nombre_nuevo;
```

-Declaraciones simples:

Una variable simple se declara especificando un nombre, un tipo, y opcionalmente un valor inicial, siguiendo la siguiente sintaxis:

```
nombre [ CONSTANT ] tipo [ NOT NULL ] [ { DEFAULT | := } valor_inicial ];
```

La opción 'CONSTANT' significa que la variable no se podrá modificar, y será necesario asignarle un valor inicial. La opción 'NOT NULL' significa que la variable siempre ha de tener un valor.

El tipo debe tener un tipo integrado en PostgreSQL para evitar que los cambios en la base de datos afecten a la función.

-Declaraciones compuestas:

Una declaración compuesta es la que se corresponde con una fila completa de una tabla. Por tanto tiene filas que se corresponden con cada columna de la tabla. Se puede declarar una variable compuesta usando 'rowtype' o 'record':

```
variable1 tabla%rowtype;
```

Para usarlos dentro de la función habrá que especificar a qué campo de la variable se hace referencia:

```
variable1.nombre := 'Ejemplo';
```

El otro tipo de variable compuesta, 'record', no está basado en una tabla en concreto y tendrá los campos que se le asignen en tiempo de ejecución. Son útiles para funciones que pueden ser llamadas desde distintas tablas, o para procedimientos de un propósito general al que no le importe de qué tabla se trata, o también para almacenar resultados de una SELECT de forma general. Su sintaxis es sencilla:

```
variable2 record;
```

-Asignaciones:

Las variables en PL/pgSQL son asignadas con valores nuevos en las sentencias de asignación. La sintaxis para una asignación es:

```
referencia := expresión;
```

La 'referencia' es el nombre de una variable (o un campo en una variable compuesta). La 'expresión' debe contener una constante, otra variable, o una referencia a un campo de una expresión compleja.

- Sentencia SELECT INTO: Un mecanismo alternativo para la asignación es una extensión de una SELECT. Esta extensión hace posible asignar una variable, una lista de variables, una variable de fila o un record.

```
SELECT expresiones INTO variable [ FROM … ];
```

Esto permite asignar una fila o 'rowtype', siempre que se especifiquen los valores de cada columna en el orden correcto:

```
DECLARE
        producto articulo%rowtype;
BEGIN
        SELECT NULL, 'cuadro', '10.50', '2' INTO producto;
END;
```

Cuando se asignen valores o variables de tipos distintos, PostgreSQL aplica los castings apropiados que pueda hacer, por eso al intentar asignar '10.50' en un campo numérico PostgreSQL le cambia el tipo de caracteres a numérico.

Cuando una SELECT devuelva más de una fila sólo se almacenará en la variable la primera. Si en cambio la SELECT no devuelve filas, la asignación no se lleva a cabo, pero PostgreSQL almacenará en la variable booleana especial 'FOUND' el resultado positivo o negativo de la consulta:

```
SELECT * INTO  producto FROM articulo WHERE descripción ~~ '%Rojo%';
```

```
IF NOT FOUND THEN
        -- acciones a tomar en caso de que la consulta no devuelva filas
END IF;
```

- PERFORM: En algunos casos no se desea capturar el resultado de la SELECT, lo que se puede conseguir con la sentencia PERFORM:

```
PERFORM consulta;
```

-Estructuras de control:

PL/pgSQL tiene estructuras para controlar el flujo o ejecución dentro de una función:

- Devolución de valores de la función:

El procesamiento de una función acaba cuando encuentra la expresión RETURN y el valor de esta expresión es el que se le proporciona al que llamó la función como su resultado. Este valor debe tener el mismo tipo que está declarado en la función después de la palabra RETURNS, o que le permita a PostgreSQL un casting correcto entre los dos tipos.

```
RETURN expresión;
```

En PL/pgSQL una función debe devolver un valor, o fallará la ejecución.

- Excepciones y mensajes:

  La ejecución de una función puede parar si se da alguna condición que la hace imposible continuar. En vez de devolver un valor, la función se marcará como excepción, y PostgreSQL terminará el procedimiento. La sentencia RAISE registra la excepción, que tendrá un nivel de seriedad:

```
RAISE nivel 'formato' [, variable … ];
```

PostgreSQL define varios niveles de excepciones:

Nivel	Comportamiento
DEBUG, LOG, INFO	Escribe un mensaje en el log
NOTICE, WARNING	Escribe un mensaje en el log, y se lo envía a la aplicación
EXCEPTION	Escribe un mensaje en el log, y termina el procedimiento

El nivel de DEBUG es útil para capturar información adicional durante la ejecución. El nivel de NOTICE señala de avisos que no son errores graves. El nivel de EXCEPTION se usa para errores graves, tras el cual el procedimiento no puede continuar su ejecución.

- Condicionales:

  PL/pgSQL permite varios tipos de condicionales, que son construcciones que ejecutan una de dos o más sentencias, o devuelven distinto resultado a partir de una condición.

- IF-THEN-ELSE: El condicional más común, que determina que si se cumple una condición ('expresión') entonces se ejecutan unas sentencias ('sentencias_1'), y en caso contrario se ejecutan otras sentencias ('sentencias_2'):

```
IF expresión
THEN
sentencias_1
[ ELSE
  sentencias_2 ]
END IF;
```

- NULLIF: Es una función que devuelve NULL si la entrada cumple un específico valor, y si no devuelve el valor de la entrada sin cambiar:

```
NULLIF(entrada, valor);
```

- CASE: Es una función que elije uno entre varios valores, dependiendo del valor de entrada:

```
CASE
  WHEN expresión_1 THEN sentencias_1
  …
ELSE sentencias_n
END;
```

- Bucles:

PL/pgSQL tiene buenos mecanismos para hacer bucles, o estructuras de control iterativas, que proporcionan varias formas de ejecutar sentencias un número de veces. El más sencillo bucle es el siguiente:

```
[ <<etiqueta>> ]
LOOP
        sentencias
END LOOP;
```

En este bucle para salir de la iteración tendrá que encontrarse en algún momento la sentencia EXIT. Además se le puede dar una etiqueta al bucle LOOP para poder localizarlo durante las sentencias (por ejemplo, en el caso de se aniden dos bucles y querer hacer el EXIT de uno de ellos.

```
EXIT [ etiqueta ] [ WHEN expresión ];
```

- WHILE: Es una estructura alternativa al LOOP que ejecuta un conjunto de sentencias mientras la condición sea cierta ('true'):

```
[ <<etiqueta>> ]
WHILE expresión
        LOOP
```

```
        sentencias
    END LOOP;
```

- FOR: Sirve para ejecutar un bucle un número fijo de veces:

```
FOR nombre_variable IN [ REVERSE ] valor_desde .. valor_hasta
    LOOP
        Sentencias
    END LOOP;
```

Este tipo de bucle ejecuta las sentencias una vez por cada valor se encuentre en el rango dado por expresiones enteras en 'valor_desde' y 'valor_hasta'. Se crea una nueva variable para el bucle, 'nombre_variable', que toma cada uno de los valores del rango por turno, incrementando (o disminuyendo si está la opción REVERSE) una unidad cada vez que se ejecuta el cuerpo del bucle.

Una alternativa para el bucle FOR permite usar el bucle una vez por cada fila sea devuelta por una consulta SELECT:

```
FOR fila IN SELECT…
    LOOP
        Sentencias
    END LOOP;
```

En este caso, por cada una de las filas devueltas por la SELECT se le asigna a la variable 'fila'. En este caso la variable 'fila' tiene que haberse declarado antes, como un 'record' o como un 'rowtype'. La última fila que se procese será accesible cuando termine el bucle desde la variable.

-Consultas dinámicas:

Normalmente las consultas sobre bases de datos en un procedimiento son fijas o con parametrización simple, pero en ciertas ocasiones se necesita usar el valor de una variable para especificar una tabla o columna. PostgreSQL no permite esto, ya que necesita optimizar la consulta una única vez, y no cada vez que se ejecuta la consulta. Sin embargo, permite la sentencia EXCUTE que permite ejecutar una sentencia SQL arbitraria, especificada como una cadena de caracteres:

```
EXECUTE cadena_caracteres;
```

La cadena se puede crear dinámicamente, teniendo precaución con la citación de nombres y valores literales (escapar bien la cadena). Hay dos funciones que ayudan a crear la cadena para una consulta dinámica: 'quote_ident' permite procesar los nombres de las tablas y los nombres de las columnas, generando una cadena de caracteres que encaja para una SELECT, y por otra parte 'quote_value', que procesa los valores:

```
EXECUTE 'UPDATE '
```

```
|| quote_ident(nombre_tabla)
|| ' SET '
|| quote_ident(nombre_columna)
|| ' = '
|| quote_literal(valor_columna)
|| ' WHERE '
...;
```

FUNCIONES SQL

Además de poder crear procedimientos, PL/pgSQL permite también crear funciones mediante SQL. Hace falta especificar el lenguaje del procedimiento como 'sql' y usar sentencias SQL en vez de PL/pgSQL.

No tiene estructuras de control, está restringido a sentencias SQL, no permite variables, ni evaluaciones condicionales, ni bucles.

```
CREATE FUNCTION función_sql (texto) RETURNS tipo_devuelto
```

TRIGGERS

En algunas aplicaciones las restricciones pueden no ser suficientes para asegurar que se cumplen algunas condiciones complejas en la base de datos. Otras veces simplemente se pretende realizar ciertas acciones cuando sucede algo (inserción, modificación o borrado) en alguna tabla.

Una solución a esto es usar disparadores (triggers), que permiten ordenar a PostgreSQL a realizar un procedimiento cuando sucede una acción. Para usar un trigger, primero hace falta definirlo, y luego crearlo (definir cuando se ejecuta el trigger).

-Definición de un trigger:

Un trigger se dispara cuando se cumple una condición, y ejecuta un tipo propio de procedimiento para triggers. Este procedimiento es similar al resto de procedimientos, pero es ligeramente más restrictivo por la forma en que se invoca. El procedimiento de un trigger se crea como una función sin parámetros y devuelve un tipo especial.

PostgreSQL invocará al trigger cuando se realizan cambios en una tabla en particular. El procedimiento puede devolver el valor NULL o una fila que encaja con la estructura de la tabla que ha provocado la invocación del trigger. Dependiendo de cada caso, se tratará el valor devuelto por el procedimiento del trigger para determinar si se lleva a cabo la acción o por el contrario da un error.

-Creación de un trigger:

Los triggers se crean con el comando CREATE TRIGGER:

```
CREATE TRIGGER nombre_trigger { BEFORE | AFTER }
{ evento [OR ...] }
ON tabla FOR EACH { ROW | STATEMENT }
EXECUTE PROCEDURE función ( argumentos )
```

Donde evento se refiere a INSERT, DELETE o UPDATE, es decir el evento sobre la tabla que ha iniciado la acción.

El trigger tiene un nombre, que sirve para poder eliminarlo posteriormente:

```
DROP TRIGGER nombre_trigger ON tabla;
```

Una vez se invoca el trigger, éste tiene acceso tanto a los datos originales (para UPDATE y DELETE) como a los datos nuevos (para INSERT y UPDATE). También se puede pedir al trigger que se ejecute antes de que ocurra el evento, para prevenir un cambio no deseado, o para cambiar los datos que van a ser insertados o actualizados. Cuando la sentencia SQL modifica varias filas a la vez, se puede elegir que el trigger se lance para cada fila (ROW) o para todas a la vez (STATEMENT).

Las variables que un procedimiento de un trigger usa para reconocer los datos son:

Variable	Descripción
NEW	Un record que contiene la nueva fila
OLD	Un record que contiene la fila vieja
TG_NAME	Variable que contiene el nombre del trigger que se ha lanzado y ha causado la ejecución del procedimiento de trigger
TG_WHEN	Variable de texto que contiene 'BEFORE' o 'AFTER', dependiendo del trigger
TG_LEVEL	Variable de texto que contiene 'ROW' o 'STATEMENT', dependiendo del trigger
TG_OP	Variable de texto que contiene 'INSERT', 'DELETE', o 'UPDATE', dependiendo del evento que ha lanzado el trigger
TG_RELID	Objeto identificador que representa a la tabla que ha activado el trigger
TG_RELNAME	Nombre de la tabla que el trigger ha lanzado
TG_NARGS	Variable entera que contiene el número de número de argumentos de la definición del trigger
TG_ARGV	Vector de cadenas de caracteres que contienen los parámetros del procedimiento, empezando en cero; Índices inválidos devuelven valores NULL

VENTAJAS DE PROCEDIMIENTOS Y TRIGGERS

-Proporcionan validaciones centrales:

Permiten forzar condiciones para las actualizaciones de las tablas en un único sitio, independiente de las aplicaciones del cliente. Si las condiciones necesitan cambiar, se modifican en único sitio.

-Seguimiento de cambios:

Se pueden usar triggers para crear un seguimiento de auditoría, escribiendo en otra tabla las filas que se actualizan. También permite señalar qué usuario ha realizado el cambio, la fecha,….

-Mejorar la seguridad:

Usando la variable 'current_user' de PostgreSQL, se puede mejorar la seguridad de la base de datos.

-Aplazar borrados:

Se puede utilizar un trigger para marcar las filas para un borrado posterior, y no borrarlas cuando la aplicación lo pide.

-Proporcionar un mapeo para los clientes:

Se pueden utilizar triggers y procedimientos para crear versiones simples de tablas que pueden ser modificados de forma más sencilla por los usuarios.

CURSORES

Cuando se realiza una consulta con una SELECT, los resultados se envían a la aplicación cliente. Si se quiere realizar una consulta con varias filas en un procedimiento, habrá que tratar el resultado de forma especial ya que el resultado tiene estructura de tabla.

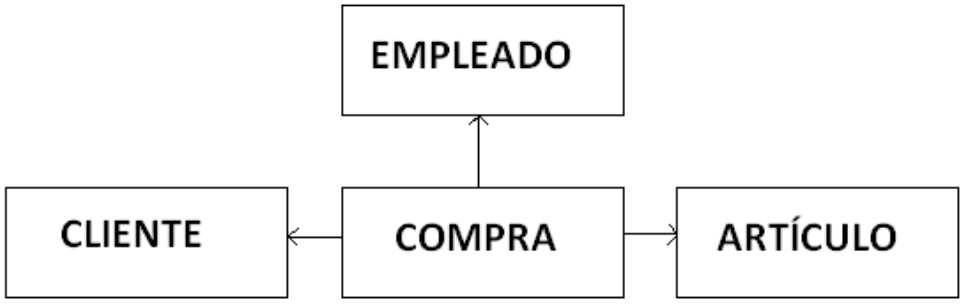
BIBLIOGRAFÍA

- [1] "PostgreSQL Introduction and Concepts"
- [2] "Beggining Databases with PostgreSQL: From Novice to Professional, Second Edition"
- [5] http://wiki.postgresql.org/wiki/Help:Variables
- [6] "PostgreSQL 9.0 High Performance"

TEMA 7 – OPTIMIZACIÓN

SACANDO INFORMACIÓN DE VARIAS TABLAS

En ocasiones se necesita sacar información de más de una tabla. Por ejemplo, una base de datos de una empresa que vende un producto podría tener las siguientes tablas: CLIENTE, EMPLEADO, ARTÍCULO y COMPRA. Lo óptimo es que en cada tabla exista un identificador numérico único para cada fila de las tablas básicas (CLIENTE, EMPLEADO y ARTÍCULO) y cada fila en COMPRA tenga el identificador del cliente, el identificador del empleado, y el identificador del artículo.



Cuando una consulta necesita información cruzando más de una tabla, los nombres de las columnas pueden dar a confusión (por ejemplo si existen en dos tablas una columna con el mismo nombre), por eso SQL permite calificar los nombres de las columnas precediéndolos el nombre de su tabla.

```
SELECT f.nombre FROM amigo f WHERE país='España';
```

El separar la información en estas 4 tablas del ejemplo, permite mantener información detallada de clientes, empleados y artículos, así como incluirlos en una compra tantas veces haga falta sólo usando su identificador. Sin una tabla específica para clientes, habría que introducir la información del cliente en cada compra realizada por él (nombre, teléfono, dirección,… se repetirían), y ante cualquier cambio habría que modificar todas las compras relacionadas. Una estructura de tablas correctamente separadas ayuda a la administración, al mantenimiento de la información, a la eficiencia, a la búsqueda, al almacenamiento compacto, y a la reducción de espacio en disco. Para enlazar correctamente las tablas de nuestro ejemplo, se supone que cada compra tiene un único cliente, un único empleado, y un único artículo.

```
CREATE TABLE cliente (id_cliente INTEGER, nombre CHAR(30), teléfono CHAR(20),
        dirección CHAR(40), código_postal INTEGER, país CHAR(20));
CREATE TABLE empleado (id_empleado INTEGER, nombre CHAR(30), fecha_contrato DATE);
CREATE TABLE artículo (id_articulo INTEGER, nombre CHAR(30), precio NUMERIC(8,2),
        peso FLOAT);
```

```
CREATE TABLE compra (id_compra INTEGER, id_cliente INTEGER, id_empleado INTEGER,
        id_articulo INTEGER, fecha_compra DATE, pago NUMERIC(8,2));
```

Una vez creada la estructura en distintas tablas, hace falta unirlas dentro de una consulta para poder obtener información en conjunto.

SELECT cliente.nombre	Resultado de la consulta
FROM cliente, compra	Tablas que se unen para la consulta
WHERE cliente.id_cliente = compra.id_cliente AND compra.id_compra = 15;	Restricción de la consulta

Como se ha explicado antes, en el caso del ejemplo tanto la tabla cliente como la tabla compra tienen una columna llamada 'id_cliente', por tanto es necesario señalar cuando hace referencia a una tabla y cuando a la otra, o esta consulta devolvería un error por ambigüedad. Para poder obtener información correctamente de varias tablas es importante que cada tabla tenga una columna que sea el identificador único de cada fila (como en el ejemplo es 'id_cliente'). Si además este identificador es numérico, añade las siguientes ventajas:

- Los números son más fáciles de escribir (faltas ortográficas, mayúsculas/minúsculas,…)
- Son independientes de la información, y si ésta cambia, el identificador se mantiene.
- Unir tablas por números es más eficiente que unir por cadenas largas.
- Los números requieren menos espacio en disco.

En algunos casos también puede ser eficiente usar un identificador alfanumérico, por ejemplo en una tabla de provincias con el identificador las letras de matrícula de coche que corresponden a esa provincia. Es eficiente porque son sólo uno o dos caracteres, son únicos, es muy poco probable que cambien, y no requiere mucho espacio en disco.

En el ejemplo, cada compra acepta un único artículo, lo que no es una situación real. Para poder crear compras con 0, 1 ó más artículos, se debería modificar la tabla 'compra' para quitarle la columna 'id_articulo', y crear una nueva tabla 'compra_articulo'. De este modo se ha creado una relación maestro/detalle entre las dos tablas: la tabla 'compra' es el maestro porque tiene la información común a cada compra, y la tabla 'compra_articulo' es el detalle porque contiene información específica de las partes que componen la compra.

```
CREATE TABLE compra (id_compra INTEGER, id_cliente INTEGER, id_empleado INTEGER,
        fecha_compra DATE, pago NUMERIC(8,2));
```

```
CREATE TABLE compra_articulo (id_compra INTEGER, id_articulo INTEGER, cantidad INTEGER);
```

VISTAS

Cuando se tiene una base de datos compleja, o cuando se tienen distintos usuarios con diferentes permisos, se necesita crear una tabla imaginaria, o vista. En el ejemplo anterior, si se quiere permitir a usuarios externos visualizar qué clientes han comprado un cierto artículo, deberían poder ver dos tablas: cliente y compra. Con una vista se puede crear un diseño que

permita mostrar cierta información a los usuarios que acceden, pero no toda la tabla. La sintaxis para crear tablas es:

```
CREATE VIEW nombre_vista AS sentencia_select;
```

Una vez creada se puede consultar como si fuera una tabla, o incluso cruzarla en una consulta con otras tablas, pero no se permite insertar o modificar porque por defecto en PostgreSQL son sólo de consulta. Internamente, cada vez que se ejecuta la consulta PostgreSQL busca en las tablas originales, por lo que la vista siempre está actualizada (no se trata de información copiada).

Para eliminar vistas, se usa un comando similar al que se realiza con las tablas.

```
DROP VIEW nombre_vista;
```

Para modificar una vista se puede hacer con el siguiente comando:

```
CREATE OR REPLACE VIEW nombre_vista AS sentencia_select;
```

Y como las vistas son sólo consultas a otras tablas, eliminarlas o modificarlas no afecta a la información almacenada.

RENDIMIENTO DE LA BASE DE DATOS

El rendimiento suele ser un problema en bases de datos grandes porque los tiempos de respuestas están ligados al tamaño físico de la base de datos. Optimizar bases de datos es una tarea avanzada que requiere de técnicas de diseño y conocimiento de detalles internos del sistema de base de datos. PostgreSQL incluye un optimizador sofisticado que trata de ejecutar las consultas sobre base de datos de la forma más eficiente posible, pero a veces requiere una ayuda adicional del diseñador. Hay formas relativamente fáciles para ayudar a mantener y mejorar el rendimiento de la base de datos PostgreSQL, empezando por conocer cómo funciona la base de datos internamente.

-Monitorización del comportamiento:

Hay dos formas de averiguar lo que PostgreSQL se encuentra haciendo:

- Monitorizando la actividad del sistema operativo:

En Linux, una forma estándar de observar los procesos de usuario de PostgreSQL es usar el comando ‘ps’ para ver los procesos con propietario ‘postgres’.

- Observando las estadísticas que PostgreSQL recolecta internamente: El colector de estadísticas de PostgreSQL tiene varias vistas que muestran las estadísticas internas.

```
SELECT * FROM pg_stat_activity;
```

```
SELECT * FROM pg_locks;
```

-VACUUM:

```
VACUUM [FULL] [FREEZE] [VERBOSE] ANALYZE
[tabla [ (columna [, ... ] ) ] ]
```

El comando VACUUM de PostgreSQL tiene dos usos:

- Recuperar espacio de almacenamiento de la base de datos:
  Con el tiempo una tabla de PostgreSQL irá acumulando filas inactivas, que son filas que ocupan espacio en la base de datos pero que ya no son accesibles. En medio de las transacciones, cuando se insertan o eliminan filas, PostgreSQL las marca como válidas o inválidas pero no las elimina por si se encuentra un fallo o un rollback y tiene que retroceder lo realizado. Por eso se quedan filas no válidas que ocupan espacio, y es este el espacio que VACUMM pretende recuperar.

  El comando VACUUM se ejecuta automáticamente a través de las tablas de la base de datos y marca las filas inválidas como reutilizables cuando se inserten datos. Esto no reduce el espacio utilizado por la base de datos pero se ejecuta de forma muy eficiente sin afectar a los demás usuarios.

  La opción VERBOSE sirve para mostrar estadísticas. La opción FULL hace que recupere todo el espacio libre y esté disponible para el sistema operativo. Esta opción FULL requiere de bloqueos en la base de datos y mucha actividad de disco para reorganizar el diseño de archivos, por lo que puede afectar negativamente al rendimiento. La opción FREEZE es sólo para preparar una base de datos como plantilla, no para usos normales. La opción ANALYZE recalcula varias estadísticas que PostgreSQL usa para planificar sus consultas de base de datos.

- Actualizar las estadísticas del optimizador:
  Como ya se ha visto SQL es un lenguaje declarativo, es decir que solicita un resultado y la base de datos devuelve qué casos lo cumplen. La base de datos tiene que buscar entre las filas de diversas tablas y, según el orden de búsqueda, según la forma de descartar filas, y otras opciones de búsqueda, entonces recuperará los datos de forma más rápida o más lenta.

  Dependiendo de la estructura de la base de datos, de las claves primarias, y del número de filas por tabla, una forma de búsqueda será más fácil que otra. PostgreSQL trata de averiguar qué camino ha de seguir para llevar a cabo la consulta de la forma más rápida. Esto es lo que hace el optimizador, planifica la consulta para una buena ejecución. Este plan se basa tanto en la estructura de la base de datos como en el tamaño de las tablas de la consulta, así como en los índices de las columnas solicitadas. Se puede visualizar el plan para una consulta en particular usando la sentencia EXPLAIN SQL:

```
EXPLAIN [VERBOSE] consulta
```

En una base de datos sencilla, la mayoría de consultas siguen búsquedas secuenciales de las tablas. PostgreSQL estima un coste asociado con cada parte de la consulta e intenta minimizar el total. Además estima el número de filas que devolverá y el tiempo de ejecución necesario. Las estimaciones de coste que usa PostgreSQL se basan en las estadísticas de cada tabla, como el número de filas, que no siempre están actualizadas para la estimación.

- VACUUM desde la línea de comandos:

```
vacuumdb [opciones] base_de_datos
```

Con las siguientes opciones:

Opción	Descripción
-a **--all**	Selecciona todas las bases de datos
-d **--dbname=base_de_datos**	Especifica la base de datos
-t **--table='tabla'**	Selecciona una única tabla
-f **--full**	Selecciona todo
-z **--analyze**	Actualiza las estadísticas del optimizador
-v **--verbose**	Usa el modo 'verbose'
--help	Muestra textos de ayuda
-h **--host=nombre_host**	Especifica el servidor de base de datos
-p **--port=puerto**	Especifica el puerto del servidor de base de datos
-U **--username=nombre_usuario**	Especifica el nombre del usuario que utiliza

- VACUUM desde pgAdmin III:

  También es posible ejecutar VACUUM gráficamente desde pgAdmin III. Haciendo clic con el botón derecho sobre la base de datos deseada, seleccionando 'Mantenimiento', y seleccionando la operación VACUUM.

ÍNDICES

Como se explica en el apartado de rendimiento, PostgreSQL crea un plan para las consultas basado en los costes de seleccionar y buscar datos. Una búsqueda secuencial de todas las filas en una tabla tendrá un alto coste si la tabla es grande. Las bases de datos usan índices para hacer más rápidas las búsquedas de filas que contienen un dado específico, ya que el coste de una búsqueda por índice es mucho menor que en una búsqueda secuencial.

Un índice es simplemente una lista organizada de valores que aparecen en una o más columnas en una tabla. La idea es que si se quiere sólo un subconjunto de filas de una tabla, una tabla pueda usar un índice para determinar qué filas enlaza, en vez de examinar cada todas las filas. Los índices ayudan a la base de datos a separar la cantidad de datos que se necesitan buscar cuando se ejecuta una consulta. Los índices no deben usarse para forzar el orden en consultas, ya que para ordenar las filas en una consulta existe el comando ORDER BY, además que una consulta no siempre devuelve las filas en el orden en que las encuentra. La existencia de un índice con ese orden es para seleccionar filas a la de ejecutar esa consulta, y así que evite buscar en ciertos bloques de filas. Los índices consiguen que los planes para las consultas dejen de ser secuenciales, para enlazar directamente con los datos que forman el índice.

Los índices no se usan en todas las consultas, se utilizan únicamente si PostgreSQL determina que la tabla es grande, y entonces la consulta selecciona sólo un pequeño porcentaje de las filas en la tabla. Para determinar si un índice debe ser utilizado, PostgreSQL debe tener estadísticas sobre la tabla, y estas estadísticas es recolectan mediante VACUUM ANALIZE. Usando las estadísticas, el optimizador sabe cuántas son las filas en la tabla, y puede determinar mejor si los índices deben utilizarse. Las estadísticas también son muy útiles para la determinación de un orden óptimo y para los métodos de unión. La recolección de estadísticas se realiza periódicamente para calcular el cambio de contenido en la tabla.

De hecho, PostgreSQL crea automáticamente un índice para una columna que se defina como clave primaria de la tabla. Esto quiere decir que hacer una consulta de una tabla condicionando la búsqueda a un valor de la clave primaria tiene un coste bajo. Además de los índices de clave primaria, se pueden crear otros adicionalmente mediante el comando SQL CREATE INDEX:

```
CREATE [UNIQUE] INDEX nombre_índice ON tabla ( columna )
```

La opción UNIQUE especifica que la columna que se indexa no contiene entradas duplicadas, creando un error si se intenta añadir o modificar una tabla donde este valor se duplique. Estos índices crean un buen rendimiento en el sistema ya que una búsqueda por este campo es muy rápida. La clave primaria se puede considerar un índice único tanto por su unicidad como por su rapidez de consulta.

Los índices creados correctamente pueden bajar de forma extrema el coste de las consultas, y son la clave para maximizar el rendimiento en una base de datos de PostgreSQL, pero también tienen su parte negativa. Las inserciones y actualizaciones en una tabla con índices serán más lentas porque el índice también se tiene que actualizar a la vez que los datos. Además las estructuras de los índices ocupan espacio físico dentro de la base de datos. Por tanto el diseñador de la base de datos debe seleccionar bien qué tablas y columnas necesitan un índice, valorando los pros y los contras que se han explicado, además de conocer en profundidad la base de datos para crearlos en consultas con mucho coste. Si se eligiera un índice y luego los datos de la búsqueda no coinciden, la consulta tiene que recorrer toda la

tabla en una búsqueda secuencial, por lo que no se ha ganado tiempo. Si por el contrario puede ocurrir que se elija un índice incompleto, es decir, que evite recorrer toda la tabla en la búsqueda pero habiendo elegido un campo más se hubiese acotado mucho más la porción de tabla a buscar. Un ejemplo sería una tabla de clientes con millones de filas, cada una especificando la provincia del cliente y se realizaran muchas consultas donde la provincia está condicionada.

En la mayoría de los casos los índices se han de crear para:

- Tablas con muchas filas y con actualizaciones poco frecuentes.
- Columnas que no son claves primarias o claves ajenas, pero que se usan para uniones complejas con otras tablas en las consultas.
- Columnas en las que se buscará una coincidencia exacta o de un prefijo suyo.

PostgreSQL permite crear índices de subcadenas de una columna, señalando el patrón de las subcadenas. Por ejemplo 'LIKE 'comienzo%'.

```
CREATE INDEX nombre_índice ON tabla ( columna patrón)
```

Los índices no tienen que cubrir la tabla entera, en PostgreSQL se pueden crear índices que satisfacen sólo una parte de la tabla, por ejemplo los que tienen unos valores que se consultan repetidamente.

```
CREATE INDEX nombre_índice ON tabla WHERE condición
```

PostgreSQL proporciona distintos tipos de índices, cada tipo utiliza un algoritmo diferente que se ajusta mejor a distintos tipos de consultas:

- B-tree
- Hash
- GiST
- SP-GiST
- GIN

-B-tree:

Por defecto el comando CREATE INDEX crea índices B-tree, que son los que más se ajustan a las situaciones comunes. Los B-tree pueden manejar consultas por igualdades y por rangos de datos que se puedan ordenar de alguna forma. En particular, el planificador de PostgreSQL usará un índice b-tree cuando la columna indexada está involucrada en una comparación usando uno de estos operadores: '<', '<=', '=', '>=' y '>'. Otras construcciones equivalentes como BETWEEN, IN, IS NULL o IS NOT NULL se pueden implementar con un índice B-tree también. El optimizador también puede usar índices B-tree en consultas con operadores de coincidencia de patrones como LIKE si el patrón es una constante y está al principio de la cadena (por ejemplo 'LIKE 'foo%').

-Hash:

Los índices hash sólo pueden manejar comparaciones simples de igualdad. El planificador de consultas usará un índice hash en el caso de que la columna indexada esté involucrada en una comparación con el operador '='.

```
CREATE INDEX nombre_índice ON tabla USING hash (columna);
```

-GiST:

Los índices GiST no son un de un único tipo de índice, si no una infraestructura dentro de la cual se implementan varios tipos de diferentes estrategias de indexación. Los operadores particulares con los que se puede usar un índice GiST varían en función de la estrategia de indexación. La distribución estándar de PostgreSQL incluye clases de operadores GiST para muchos tipos de datos geométricos de 2 dimensiones, que dan soporte a las consultas indexadas usando las siguientes operaciones: '<<', '&<', '&>', '>>', '<<|', '&<|', '|&>', '|>>', '@>', '<@', '~=' y '&&'.

-SP-GiST:

Igual que los índices GiST, los índices SP-GiST ofrecen una infraestructura que dan soporte a varios tipos de búsquedas. Los índices SP-GiST permiten la implementación de un amplio rango de diferentes estructuras de datos basadas en disco, como Quadtrees, árboles k-n, y árboles sufijo. Como ejemplo PostgreSQL incluye el operador de clases GiST para puntos de dos dimensiones, que dan soporte a consultas indexadas usando estos operadores: '<<', '>>', '~=', '<@', '<^' y '>^'.

-GIN:

Los índices GIN son índices invertidos que pueden manejar valores que contienen más de una clave, por ejemplo vectores. Igual que GiST y SP-GiST, GIN da soporte a muchos tipos de estrategias de indexación definidas por usuario y a los operadores particulares con los que el índice GIN se puede usar depende en la estrategia de indexación. Como ejemplo, la distribución estándar de PostgreSQL incluye clases de operador GIN para vectores de una dimensión, que dan soporte a consultas indexadas usando estos operadores: '<@', '@>', '=' y '&&'.

OBJETOS GRANDES (IMÁGENES)

Tradicionalmente las bases de datos siempre han estado limitadas a la hora de almacenar datos, tanto por el tipo como por el tamaño, y fue PostgreSQL de los primeros sistemas en aceptar cadenas de miles de caracteres. Actualmente PostgreSQL admite distintas formas de almacenar objetos grandes, y en este apartado se va a ver como se pueden almacenar imágenes.

-Usando enlaces:

Una primera solución es evitar almacenar la imagen como tal en la base de datos, la idea es colocar la imagen en un directorio del servidor de forma manual, en un servidor compartido, o en un servidor web. Entonces la base de datos únicamente tiene que guardar la información

sobre el enlace al archivo de imagen, y el usuario que consulte la imagen (o la aplicación del cliente de forma automática) sólo tiene que acceder al enlace.

La ventaja de usar enlaces es que el tamaño de la base de datos se mantiene al mínimo y las bases de datos se mantienen independientes de la imagen. Además obtener el archivo será mucho más rápido que si estuviera almacenado con las tablas de la base de datos. Y también se podría almacenar el archivo en distintos lugares, según convenga al usuario por temas de espacio y de accesibilidad.

La desventaja de usar enlaces es que no se puede asegurar la integridad referencial en la base de datos. Si la imagen se modifica o se borra, la base de datos no se actualiza automáticamente. Para hacer una copia de seguridad del sistema, se tendrá que acceder al sistema de archivos donde se encuentra la imagen, así como a la propia base de datos. Además hay que asegurarse que el tipo de enlace es accesible para todos los usuarios, por temas de permisos y de tipo de archivos.

-Usando cadenas de texto codificadas:

En PostgreSQL 7.1 el límite de tamaño para un campo se incrementó hasta 1 GB, lo que para efectos prácticos reales es ilimitado. Por tanto una forma de almacenar imágenes es considerarlas como tipo texto en la base de datos.

Las imágenes en general están formadas por datos binarios, que no siempre encajan con tipos de caracteres. Por esto habría que codificar la información de alguna forma, como por ejemplo usando codificación hexadecimal o codificación MIME. Manejar las cadenas de texto extremadamente largas que resultan puede causar problemas por los límites de las aplicaciones del cliente, por los mecanismos de transporte de datos, o por los controladores ODBC. El espacio de almacenamiento necesario para codificar cadenas de texto también será más del doble del necesario que el del archivo binario de la imagen.

-Usando BLOBs:

Uno de los tipos de datos propios de PostgreSQL más usados es BLOB (binary large object: objeto grande binario), que permite almacenar grandes objetos de datos. El tipo BLOB permite crear una aplicación de base de datos que maneje formatos de datos sin estructura definida, como imágenes, es decir que permite almacenar la imagen como dato, o cualquier objeto binario, en PostgreSQL.

PostgreSQL da soporte al identificador de dila ‘oid’, que es el que se maneja para referenciar datos arbitrarios y para transferir los contenidos de cualquier archivo a la base de datos, y extraer un objeto de la base de datos a un archivo.

```
CREATE TABLE imagen
(
        item_id         integer NOT NULL,
        picture         oid,
        CONSTRAINT      imagen_pk               PRIMARY KEY(item_id),
```

```
        CONSTRAINT    imagen_item_id_fk  FOREIGN KEY(item_id) REFERENCES item(item_id)
);
```

PostgreSQL proporciona varias funciones para que las consultas SQL puedan insertar, recuperar y eliminar BLOBs de una tabla.

```
INSERT INTO imagen VALUES (3, lo_import('/tmp/image.jpg'));
```

Los contenidos del archivo especificado se leen en un objeto BLOB y almacenados en la base de datos. Para recuperar un BLOB hay que especificar el archivo donde escribirá el contenido del dato.

```
SELECT lo_export(picture, '/tmp/image2.jpg')
FROM imagen WHERE item_id = 3;
```

Igualmente hay una función para borrar el BLOB de la base de datos, aunque hay que tener especial cuidad porque cualquier referencia al BLOB permanecerá.

```
SELECT lo_unlink(picture) FROM imagen WHERE item_id = 3;
```

Como las operaciones con BLOBs y sus 'oid' están desacoplados, se debe controlar ambos a la vez, y cuando se borra un BLOB se debe configurar la referencia a NULL para prevenir errores.

```
UPDATE imagen SET picture=NULL WHERE item_id = 3;
```

Hay tres cosas a tener en cuenta al importar y exportar BLOBs:

- Como la importación la desarrolla el servidor, los archivos recuperados y almacenados por la importación/exportación deben contener una ruta accesible por el servidor (y no la ruta local).
- Los archivos exportados deben estar localizados en un directorio en el que el usuario de servidor pueda crear archivos.
- Todas las manipulaciones de objetos grandes se deben ejecutar dentro de una transacción SQL.

BIBLIOGRAFÍA

- [1] "PostgreSQL Introduction and Concepts"
- [2] "Beggining Databases with PostgreSQL: From Novice to Professional, Second Edition"
- [6] "PostgreSQL 9.0 High Performance"
- [3] http://wiki.postgresql.org/wiki/Preguntas_Frecuentes

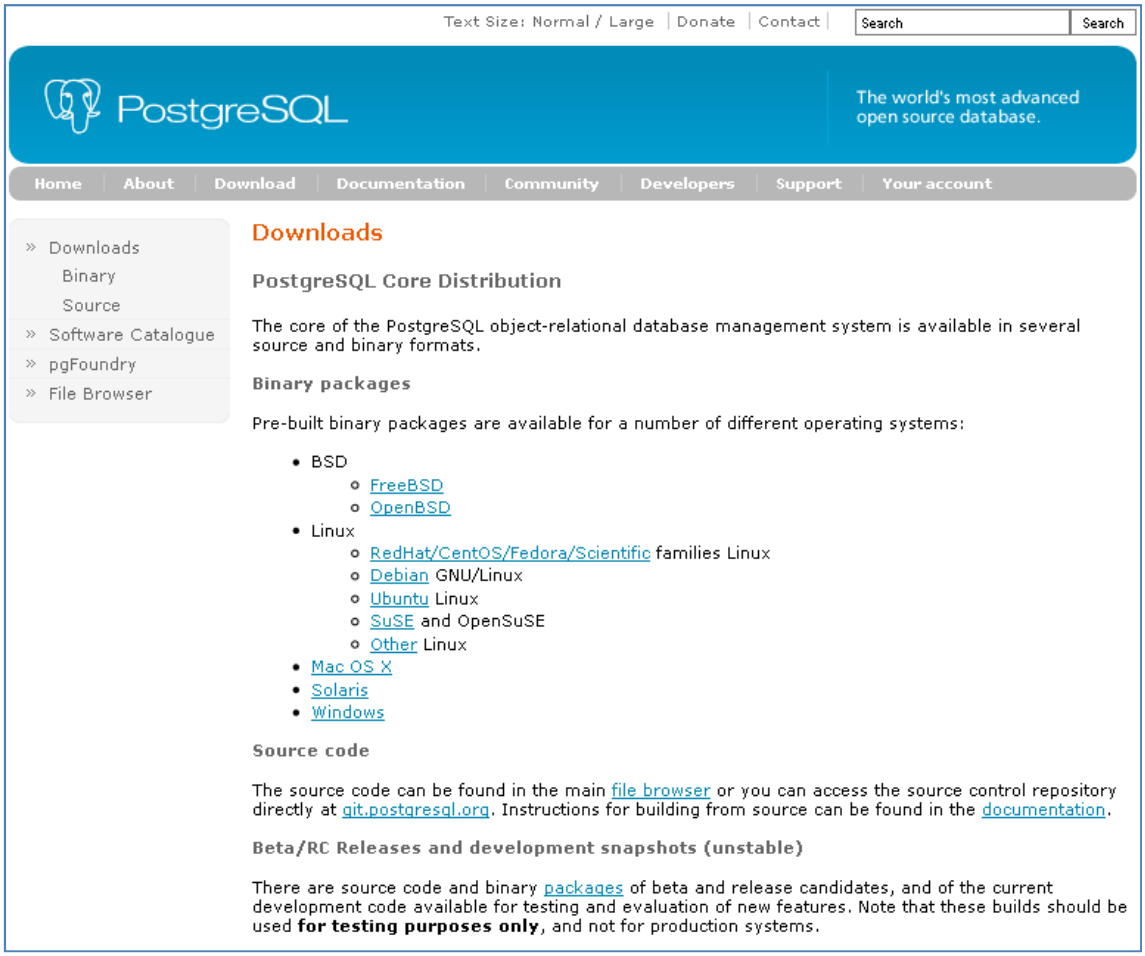
TEMA 8 – CASO DE ESTUDIO

INSTALACIÓN DE POSTGRESQL PARA WINDOWS

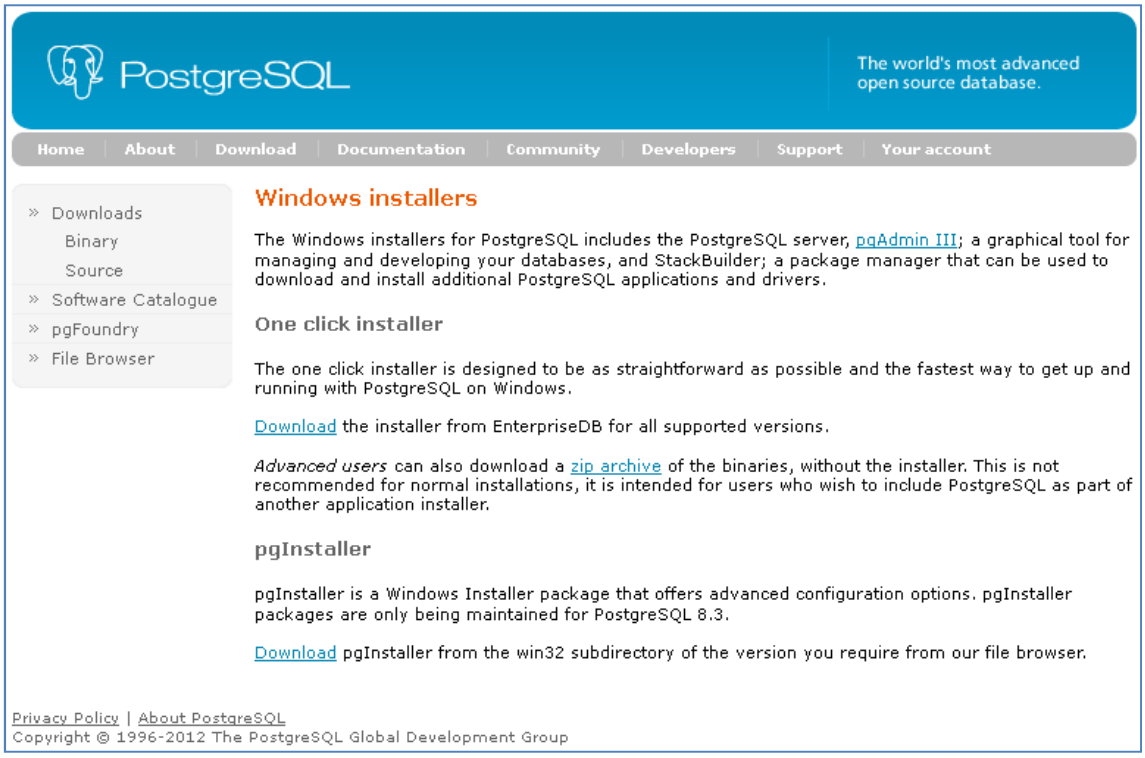
Aunque PostgreSQL ha sido desarrollado para plataformas tipo UNIX, se ha pensado para ser fácilmente portable entre sistemas operativos. Primero se creó la aplicación cliente PostgreSQL para Windows, y desde la versión 7.1 PostgreSQL se puede compilar, instalar y ejecutar como servidor PostgreSQL en Microsoft Windows.

Desde la versión PostgreSQL 8.0 hay disponible una versión nativa para Windows que permite una instalación tanto de servidor como de cliente, que facilita la instalación sobre Windows. Antes de esta versión los usuarios de Windows debían instalar programas adicionales para que diese soporte a algunas funciones UNIX. A partir de la versión 8.4 de PostgreSQL se dispone de un instalador sencillo para Linux, Windows y MacOS que incluye por defecto la última versión del servidor PostgreSQL, los programas clientes por defecto, y el cliente gráfico PgAdmin III.

Para este caso se va a instalar la versión 9.2.0 sobre el sistema operativo Windows XP SP3.

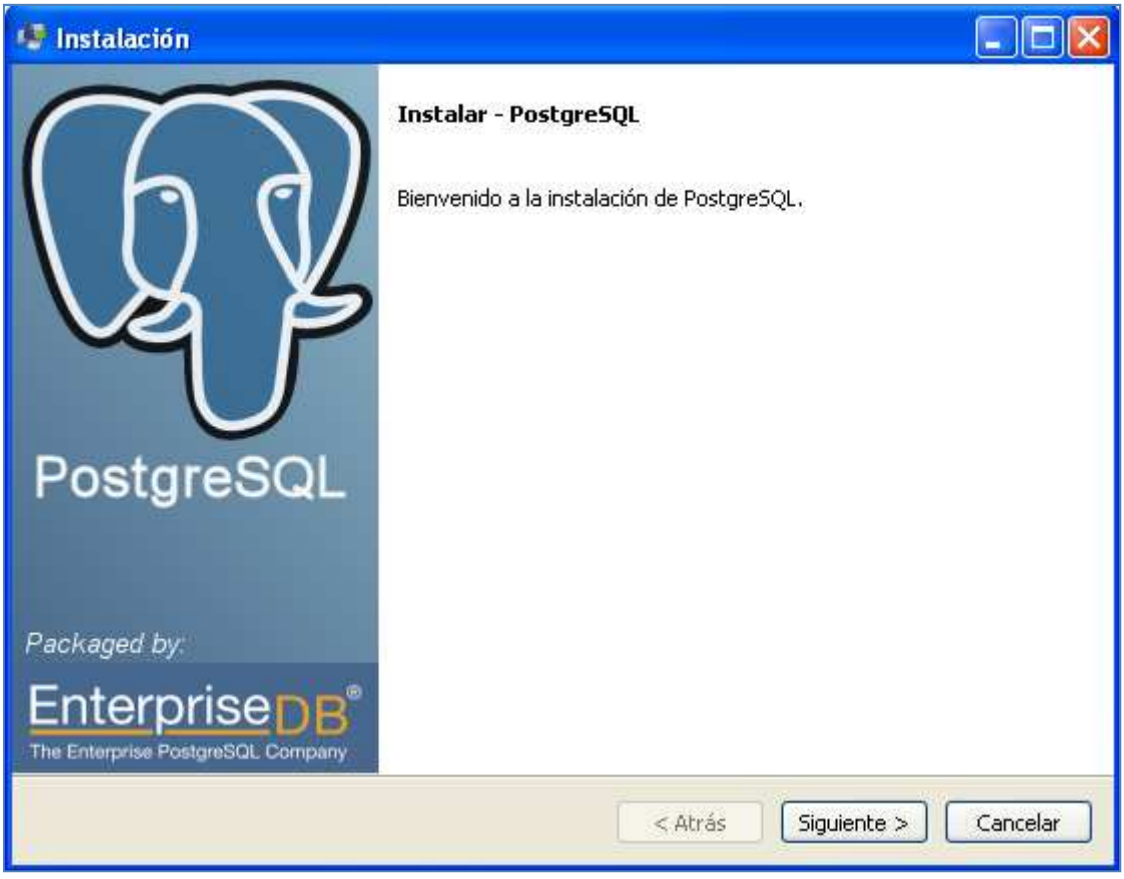


La instalación para Windows se encuentra en la página principal de PostgreSQL, en http://www.postgresql.org/download/.

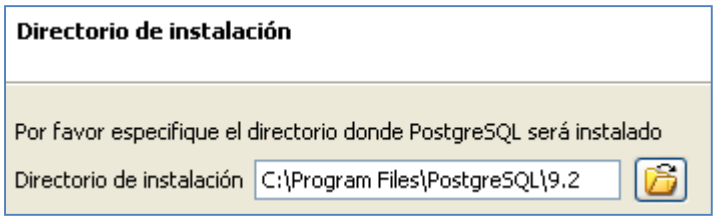


Al descargar la versión disponible, se tiene que ejecutar y seguir los siguientes:

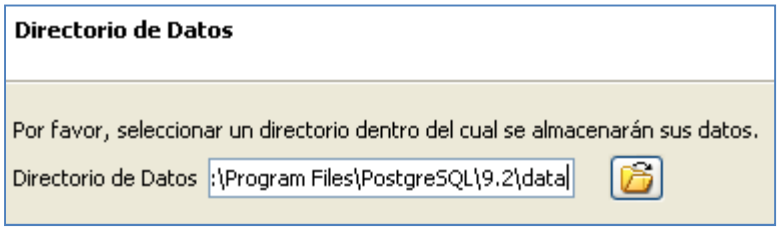
- Seleccionar automática del idioma por parte de PostgreSQL:



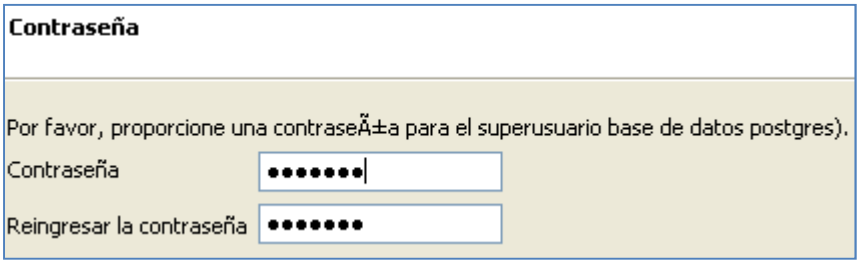
- Elegir la carpeta para la instalación de la aplicación:



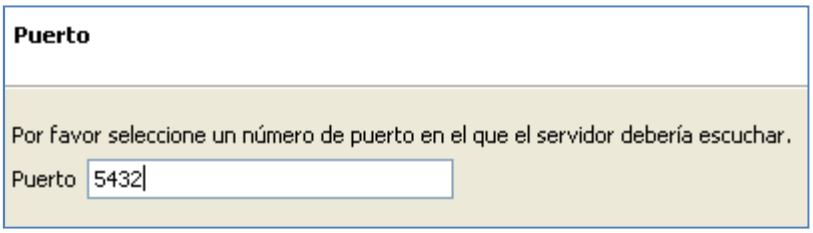
- Elegir la carpeta donde se almacenarán los archivos de bases de datos:



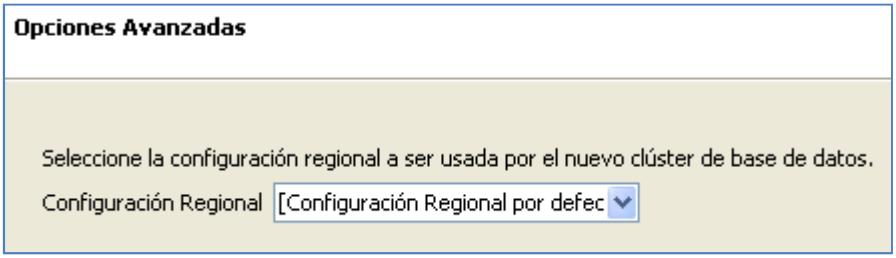
- Elegir una contraseña para el superusuario:



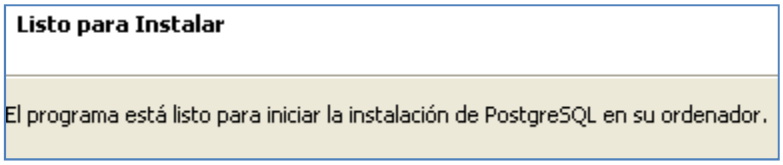
- Elegir el puerto de servidor:



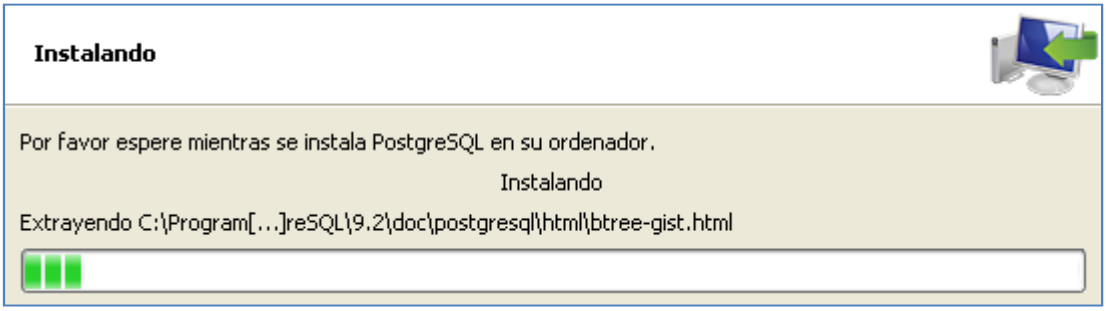
- Elegir la configuración regional:



- Confirmar los pasos anteriores:

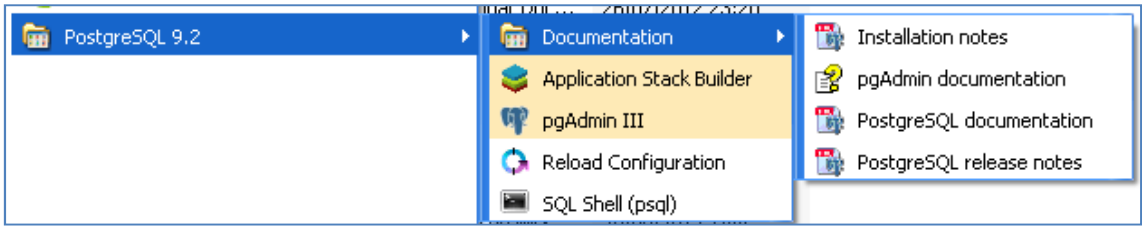


- Esperar mientras se instalan los componentes:



- Instalación terminada:

Una vez terminada la instalación, se pregunta si se desea abrir el programa 'Stack Builder', que sirve para instalar diversos programas adicionales. En este punto se ha instalado los siguientes componentes:



PostgreSQL se debe ejecutar con un usuario que no sea el administrador de la base de datos, para evitar riesgos de seguridad potenciales. De esta forma si se consiguiese tener acceso a PostgreSQL por un error de seguridad, los únicos archivos en riesgo serían los manejados por PostgreSQL pero no todo el servidor. Este usuario es el superusuario para PostgreSQL, que es un usuario de la base de datos con permisos para crear y manejar las bases de datos desde el servidor. La cuenta para PostgreSQL se usa por los clientes que se conectan a la base de datos y es PostgreSQL el que autentifica estas cuentas, por eso usar distintos nombres de usuario y claves de acceso para el superusuario de la base de datos aporta una mayor seguridad.

Para configurar el acceso como cliente, es decir los equipos remotos y usuarios que se pueden conectar al servicio de PostgreSQL, hay que editar el archivo 'pg_hba.conf'. Este archivo contiene muchos comentarios que documentan las opciones disponibles para el acceso remoto. Para permitir que los usuarios desde cualquier equipo dentro de la red local puedan acceder a todas las bases de datos del servidor, habrá que añadir una línea como la siguiente al final del archivo:

```
host    all         all         192.168.0.0/16          trust
```

Que se interpreta como "todos los equipos que su dirección ip empiece con 192.168 pueden acceder a todas las bases de datos". Tras guardar el archivo y reiniciar el servidor PostgreSQL, los usuarios pueden acceder.

COMENZAR SESIÓN DE BASE DE DATOS

PostgreSQL se comunica mediante el modelo de cliente/servidor en el que la parte servidor se encuentra ejecutándose en espera de requerimientos del cliente, y ante estos le devuelve respuestas al cliente.

Cada servidor de PostgreSQL maneja un número de base de datos, y estas bases de datos son las áreas de almacenamiento usadas por el servidor para particionar la información.

CREACIÓN DE LA BASE DE DATOS DE EJEMPLO

Una vez está PostgreSQL correctamente instalado y en ejecución, el siguiente pase es crear una base de datos. En nuestro ejemplo se va a llamar 'tienda', y contendrá las tablas ya conocidas 'cliente', 'empleado', 'articulo' y 'compra'.

Antes de crear una base de datos, una forma de asegurarse que PostgreSQL se está ejecutando en el sistema es ver que en los procesos que se están ejecutando está incluido el proceso 'postmaster.exe'.

El siguiente paseo es señalar qué usuarios son los que:

- Podrán ver datos, insertarlos, o actualizarlos.
- Podrán crear bases de datos.
- Podrán controlar el acceso a los datos.

Para esto existe la utilidad 'createuser', y para eso hay que ejecutar el fichero 'createuser.exe':

```
C:\Program Files\PostgreSQL\9.2\bin>createuser -U postgres –P javi
Ingrese la contraseña para el nuevo rol:
Ingrésela nuevamente:
Contraseña:
```

En este ejemplo 'postgres' es el usuario con permisos para crear usuarios, y 'javi' es el usuario creado.

Para crear la base de datos hay que ejecutar 'createdb.exe':

```
C:\Program Files\PostgreSQL\9.2\bin>createdb -U javi tienda
Contraseña:
```

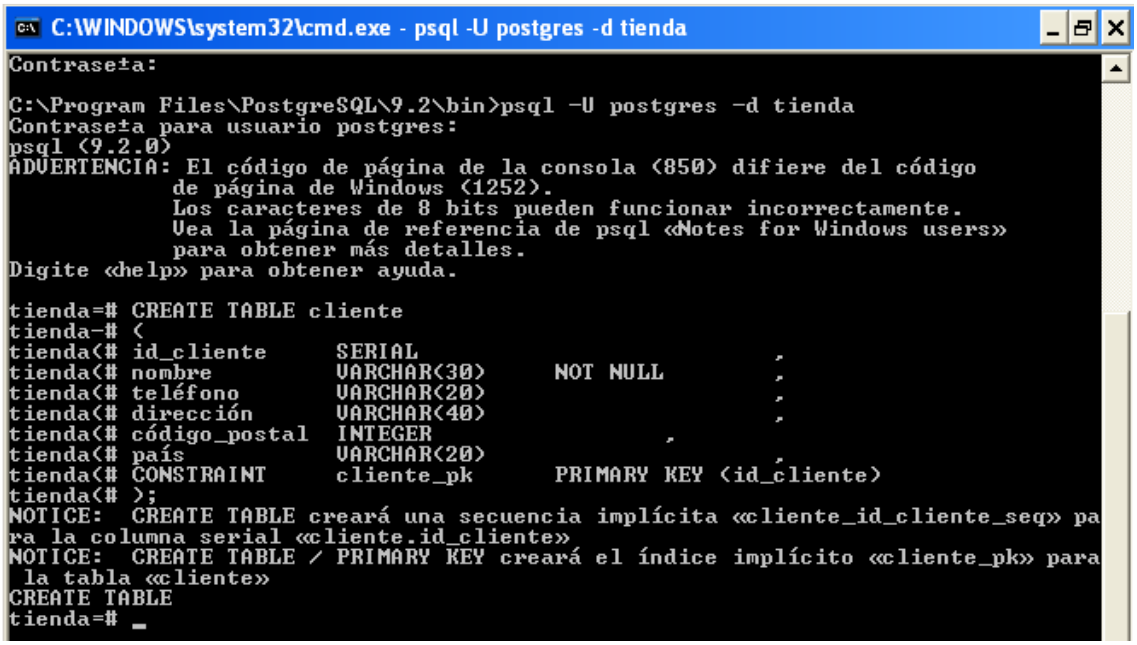
Y ahora ya se puede conectar a la base de datos 'tienda':

```
C:\Program Files\PostgreSQL\9.2\bin>psql -U javi -d tienda
Contraseña para usuario javi:
psql (9.2.0)
ADVERTENCIA: El código de página de la consola (850) difiere del código de página de
        Windows (1252). Los caracteres de 8 bits puede funcionar incorrectamente. Vea la
        página de referencia de psql <<Notes for Windows users>> para obtener más
        detalles.
Digite <<help>> para obtener ayuda.
tienda=#
```

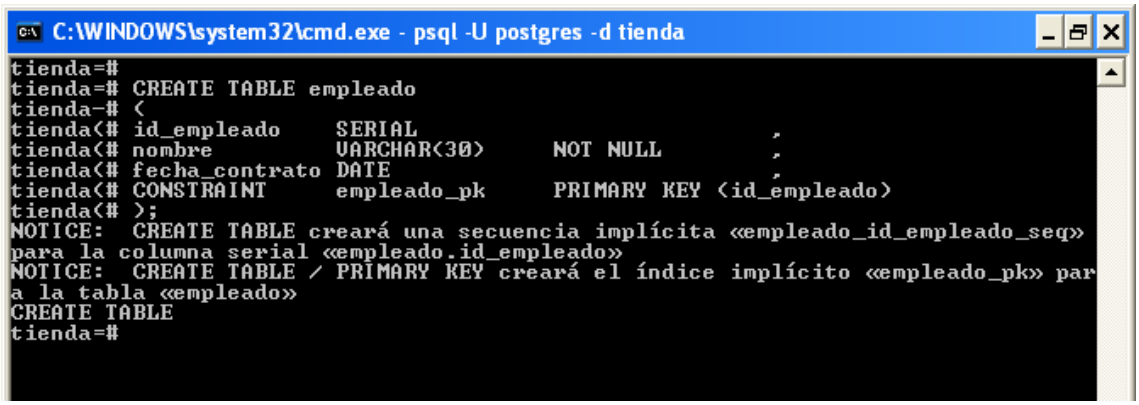
Ahora ya se pueden crear las tablas en la base de datos 'tienda', escribiendo el código SQL que existe para esto.

```
CREATE TABLE cliente
(
id_cliente      SERIAL                      ,
nombre          VARCHAR(30)   NOT NULL      ,
teléfono        VARCHAR(20)                 ,
dirección       VARCHAR(40)                 ,
código_postal   INTEGER                     ,
```

```
país            VARCHAR(20)                 ,
CONSTRAINT      cliente_pk      PRIMARY KEY (id_cliente)
);
```

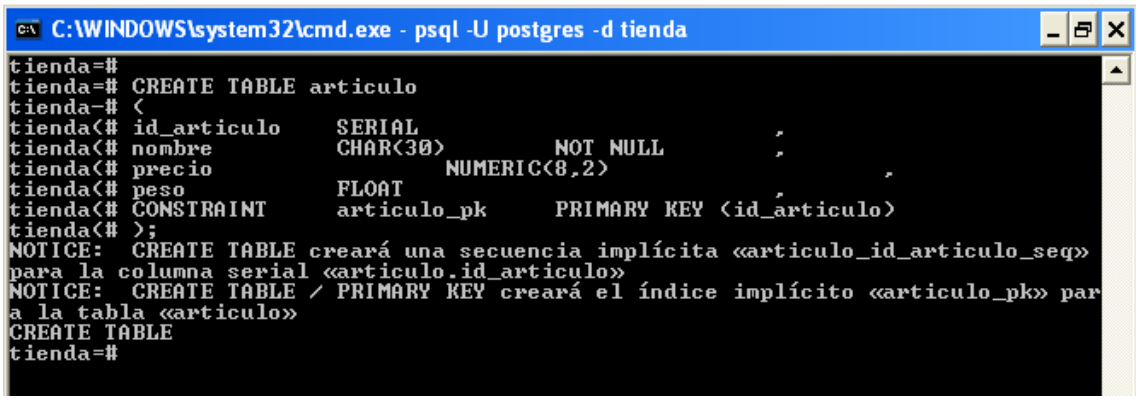



```
CREATE TABLE empleado
(
id_empleado     SERIAL                      ,
nombre          VARCHAR(30)     NOT NULL    ,
fecha_contrato  DATE                        ,
CONSTRAINT      empleado_pk     PRIMARY KEY (id_empleado)
);
```

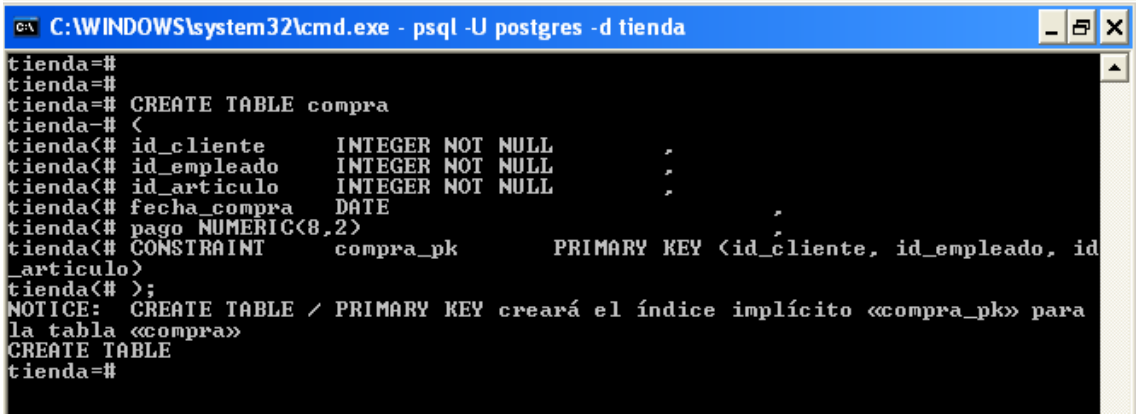



```
CREATE TABLE articulo
(
id_articulo     SERIAL                      ,
```

```
nombre          CHAR(30)        NOT NULL        ,
precio          NUMERIC(8,2)                    ,
peso            FLOAT                           ,
CONSTRAINT      articulo_pk     PRIMARY KEY (id_articulo)
);
```




```
CREATE TABLE compra
(
id_cliente      INTEGER         NOT NULL        ,
id_empleado     INTEGER         NOT NULL        ,
id_articulo     INTEGER         NOT NULL        ,
fecha_compra    DATE                            ,
pago NUMERIC(8,2)                               ,
CONSTRAINT      compra_pk       PRIMARY KEY (id_cliente, id_empleado, id_articulo)
);
```



Una vez creadas las tablas, hay que introducir valores en cada una para poder trabajar con ellas

ACCEDIENDO A LA INFORMACIÓN DE LA BASE DE DATOS DE EJEMPLO

-PSQL:

La herramienta 'psql' ayuda a conectarse a la base de datos, ejecutar consultas, y administrar una base de datos, incluyendo la creación de la base de datos, adición de nuevas tablas y actualización o inserción de datos, mediante comandos SQL. Para conectarse a una base de datos con psql, hace falta llamarlo así:

```
$ psql -d tienda
```

Una vez en marcha, psql está preparado para recibir peticiones:

```
tienda=>
```

Para usuarios con todos los permisos, lo anterior cambia a:

```
tienda=#
```

Las sentencias SQL pueden ser de una o varias líneas, pero siempre acaban con ';'. Además en 'psql' se pueden ejecutar otros comandos internos útiles que no son propios de SQL. Algunas plataformas como 'psql' permiten recuperar el histórico de comandos, y se puede volver a llamar comandos para poder editarlos o ejecutarlos. En 'psql' se consigue con las teclas de flechas arriba y abajo. Para examinar la estructura de la base de datos (nombre y definición de las tablas, funciones, usuarios,…) se ejecuta el comando '\d'.

-ODBC:

Muchas de las herramientas de este apartado, así como algunos de los interfaces de lenguajes de programación para bases de datos, usan la interfaz estándar ODBC para conectarse a PostgreSQL. ODBC define una interfaz común para bases de datos y está basado en las interfaces de programación X/Open y ISO/IEC. ODBC significa Open Database Connnectivity, (Conectividad de Bases de Datos abierta), y no está limitado sólo a clientes de Microsoft Windows. Programas escritos en muchos lenguajes, como C, C++, Ada, PHP, Perl, Python,…, permiten el uso de ODBC, igualmente que muchas aplicaciones, como OpenOffice, Gnumeric, Microsoft Access, Microsoft Excel,….

Para poder utilizar ODBC en una máquina cliente, se necesita una aplicación escrita para interfaz ODBC y un controlador para la base de datos. PostgreSQL tiene un controlador ODBC llamado 'psqlodbc', que se instala en el cliente. A menudo las máquinas de los clientes y el servidor usan sistemas operativos distintos, así como las máquinas de los clientes entre sí, requiriendo una compilación del controlador ODBC sobre distintas plataformas.

El código fuente y una instalación binaria para Windows están disponibles desde la página web del proyecto 'psqlODBC': http://gborg.postgresql.org/project/psqlodbc/. Desde Microsoft Windows, los controladores ODBC se encuentran en 'Herramientas Administrativas', dentro del 'Panel de Control'. Para instalarlo, primero hay que descargárselo desde la página web del proyecto, y después seguir las instrucciones en una sencilla instalación.

-pgAdmin III:

Es una completa interfaz gráfica para bases de datos de PostgreSQL. Esta interfaz se desarrolla más ampliamente en el siguiente punto.

-phpPgAdmin:

Es una alternativa basada en web para manejar bases de datos de PostgreSQL. Es una aplicación escrita en PHP, que se instala en un servidor web, y proporciona una interfaz en navegador web para administrar los servidores de bases de datos. La página web del proyecto es http://phppgadmin.sourceforge.net/.

Con phpPgAdmin se puede:

- Manejar usuarios y grupos
- Crear tablespaces (forma de almacenamiento), bases de datos y esquemas
- Manejar tablas, índices, restricciones, triggers, reglas y privilegios
- Crear vistas, secuencias y funciones
- Crear y ejecutar informes
- Visualizar datos de tablas
- Ejecutar comandos SQL
- Exportar datos de tablas a muchos formatos: SQL, COPY (tipo de comando de SQL), XML, XHTML, CSV (valores separados por comas), 'pg_dump',…
- Importar scripts de SQL, COPY, XML, CSV,…

-Rekall:

Es una aplicación de usuario para bases de datos multiplataforma, desarrollado originalmente por 'theKompany' (http://www.thekompany.com/) como una herramienta para extraer, mostrar y actualizar datos desde diversos tipos de bases de datos. Funciona con PostgreSQL, MySQL, y IBM DB2 con controladores nativos, y con otras bases de datos usando OBDC.

Aunque Rekall no incluye las funcionalidades de administración que puede tener pgAdmin y phpPgAdmin, añade algunas funcionalidades de usuario muy útiles. Por ejemplo, contiene un diseñador visual de consultas y un constructor de formularios para crear aplicaciones de entrada de datos. Además, Rekall permite usar Python para crear scripts, permitiendo construir sofisticadas aplicaciones sobre la base de datos.

Rekall tiene dos versiones, una comercial y otra de código abierto, ambas disponibles desde la página web http://www.rekallrevealed.org/. La versión de código abierto se puede compilar y ejecutar en Linux y otros sistemas bajo el entorno KDE, o con librerías KDE disponibles. La versión comercial añade la versión para Windows y da soporte a conexiones ODBC. Rekall se conecta a PostgreSQL usando un controlador nativo, siendo muy sencillo de utilizar.

-Microsoft Access:

Aunque puede no parecer buena idea manejar una base de datos PostgreSQL desde Access, si ya existe una base de datos sobre Access, se puede utilizar PostgreSQL para almacenar los datos, además de ciertas herramientas que son ventajosas en Microsoft Access.

Aunque PostgreSQL se esté ejecutando en un servidor UNIX o Linux, se puede permitir a los usuarios usar Access u otras aplicaciones para crear listados de datos o formularios de entrada de datos para las bases de datos de PostgreSQL. Es sencillo interactuar Access con el servidor de PostgreSQL usando la interfaz ODBC.

-Microsoft Excel:

Igual que con Microsoft Access, se puede usar Excel para añadir funcionalidades a PostgreSQL. De forma similar a la que se trabaja con Access, se puede incluir información en la hoja de cálculo que se toma desde (o conectada a) un origen remoto de información. Cuando la información cambia, se puede refrescar la hoja de cálculo y visualizar la nueva información. Una vez está enlazado Excel y PostgreSQL, se pueden utilizar herramientas de Excel, como la creación de gráficas para visualizar los datos.

-Otras herramientas:

Existen muchas más herramientas para PostgreSQL, muchas incluidas en el proyecto 'pgfoundry' con la página web http://pgfoundry.org. La página web de GBorg http://gborg.postgresql.org/ también tiene muchos proyectos relacionados con PostgreSQL.
El sitio central de proyectos se encuentra en la página web http://projects.postgresql.org, y existe una lista de herramientas gráficas que dan soporte a PostgreSQL en http://techdocs.postgresql.org/guides/GUITools.

Un monitorizador de sesiones llamado 'pgmonitor' se encuentra en la página web http://gborg.postgresql.org/project/pgmonitor.

BIBLIOGRAFÍA

- [1] "PostgreSQL Introduction and Concepts"
- [2] "Beggining Databases with PostgreSQL: From Novice to Professional, Second Edition"
- [5] http://en.wikipedia.org/wiki/PostgreSQ

APÉNDICE A – LISTADO DE COMANDOS DE SQL EN POSTGRESQL

COMANDOS DE SQL EN POSTGRESQL

ABORT	CREATE INDEX	DROP TYPE
ALTER AGGREGATE	CREATE LANGUAGE	DROP USER
ALTER CONVERSION	CREATE OPERATOR CLASS	DROP VIEW
ALTER DATABASE	CREATE OPERATOR	END
ALTER DOMAIN	CREATE RULE	EXECUTE
ALTER FUNCTION	CREATE SCHEMA	EXPLAIN
ALTER GROUP	CREATE SEQUENCE	FETCH
ALTER INDEX	CREATE TABLE	GRANT
ALTER LANGUAGE	CREATE TABLE AS	INSERT
ALTER OPERATOR CLASS	CREATE TABLESPACE	LISTEN
ALTER OPERATOR	CREATE TRIGGER	LOAD
ALTER SCHEMA	CREATE TYPE	LOCK
ALTER SEQUENCE	CREATE USER	MOVE
ALTER TABLE	CREATE VIEW	NOTIFY
ALTER TABLESPACE	DEALLOCATE	PREPARE
ALTER TRIGGER	DECLARE	REINDEX
ALTER TYPE	DELETE	RELEASE SAVEPOINT
ALTER USER	DROP AGGREGATE	RESET
ANALYZE	DROP CAST	REVOKE
BEGIN	DROP CONVERSION	ROLLBACK
CHECKPOINT	DROP DATABASE	ROLLBACK TO SAVEPOINT
CLOSE	DROP DOMAIN	SAVEPOINT
CLUSTER	DROP FUNCTION	SELECT
COMMENT	DROP GROUP	SELECT INTO
COMMIT	DROP INDEX	SET
COPY	DROP LANGUAGE	SET CONSTRAINTS
CREATE AGGREGATE	DROP OPERATOR	SET SESSION AUTHORIZATION
CREATE CAST	DROP OPERATOR CLASS	SET TRANSACTION
CREATE CONSTRAINT TRIGGER	DROP RULE	SHOW
CREATE CONVERSION	DROP SCHEMA	START TRANSACTION
CREATE DATABASE	DROP SEQUENCE	TRUNCATE
CREATE DOMAIN	DROP TABLE	UNLISTEN
CREATE FUNCTION	DROP TABLESPACE	UPDATE
CREATE GROUP	DROP TRIGGER	VACUUM

SINTAXIS DE SQL EN POSTGRESQL

-ABORT

Aborta la transacción actual.

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

-ALTER AGGREGATE

Cambia la definición de una función agregada.

```
ALTER AGGREGATE name ( type ) RENAME TO new_name
```

ALTER AGGREGATE name (type) OWNER TO new_owner

-ALTER CONVERSION

Cambia la definición de una conversión.

ALTER CONVERSION name RENAME TO new_name
ALTER CONVERSION name OWNER TO new_owner

-ALTER DATABASE

Cambia una base de datos.

ALTER DATABASE name SET parameter { TO | = } { value | DEFAULT }
ALTER DATABASE name RESET parameter
ALTER DATABASE name RENAME TO new_name
ALTER DATABASE name OWNER TO new_owner

-ALTER DOMAIN

Cambia la definición de un dominio.

ALTER DOMAIN name { SET DEFAULT expression | DROP DEFAULT }
ALTER DOMAIN name { SET | DROP } NOT NULL
ALTER DOMAIN name ADD domain_constraint
ALTER DOMAIN name DROP CONSTRAINT constraint_name [RESTRICT | CASCADE]
ALTER DOMAIN name OWNER TO new_owner

-ALTER FUNCTION

Cambia la definición de una función.

ALTER FUNCTION name ([type [, ...]]) RENAME TO new_name
ALTER FUNCTION name ([type [, ...]]) OWNER TO new_owner

-ALTER GROUP

Cambio un grupo de usuarios.

ALTER GROUP groupname ADD USER username [, ...]
ALTER GROUP groupname DROP USER username [, ...]
ALTER GROUP groupname RENAME TO new_name

-ALTER INDEX

Cambia la definición de un índice.

ALTER INDEX name action [, ...]
ALTER INDEX name RENAME TO new_name

Donde action puede valer:

OWNER TO new_owner
SET TABLESPACE indexspace_name

-ALTER LANGUAGE

Cambia la definición de un lenguaje procedural.

ALTER LANGUAGE name RENAME TO new_name

-ALTER OPERATOR

Cambia la definición de un operador.

ALTER OPERATOR name ({ lefttype | NONE } , { righttype | NONE }) OWNER TO new_owner

-ALTER OPERATOR CLASS

Cambia la definición de una clase de operador.

ALTER OPERATOR CLASS name USING index_method RENAME TO new_name
ALTER OPERATOR CLASS name USING index_method OWNER TO new_owner

-ALTER SCHEMA

Cambia la definición de un esquema.

ALTER SCHEMA name RENAME TO new_name
ALTER SCHEMA name OWNER TO new_owner

-ALTER SEQUENCE

Cambia la definición de un generador de secuencia.

ALTER SEQUENCE name [INCREMENT [BY] increment] [MINVALUE minvalue | NO MINVALUE] [MAXVALUE maxvalue | NO MAXVALUE] [RESTART [WITH] start] [CACHE cache] [[NO] CYCLE]

-ALTER TABLE

Cambia la definición de una tabla.

ALTER TABLE [ONLY] name [*] action [, ...]
ALTER TABLE [ONLY] name [*] RENAME [COLUMN] column TO new_column
ALTER TABLE name RENAME TO new_name

Donde action puede valer:

ADD [COLUMN] column_type [column_constraint [...]]
DROP [COLUMN] column [RESTRICT | CASCADE]
ALTER [COLUMN] column TYPE type [USING expression]
ALTER [COLUMN] column SET DEFAULT expression
ALTER [COLUMN] column DROP DEFAULT
ALTER [COLUMN] column { SET | DROP } NOT NULL
ALTER [COLUMN] column SET STATISTICS integer

ALTER [COLUMN] column SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
ADD table_constraint
DROP CONSTRAINT constraint_name [RESTRICT | CASCADE]
CLUSTER ON index_name
SET WITHOUT CLUSTER
SET WITHOUT OIDS
OWNER TO new_owner
SET TABLESPACE tablespace_name

-ALTER TABLESPACE

Cambia la definición de un tablespace.

ALTER TABLESPACE name RENAME TO new_name
ALTER TABLESPACE name OWNER TO new_owner

-ALTER TRIGGER

Cambia la definición de un trigger.

ALTER TRIGGER name ON table RENAME TO new_name

-ALTER TYPE

Cambia la definición de un tipo.

ALTER TYPE name OWNER TO new_owner

-ALTER USER

Cambia la cuenta de un usuario de la base de datos.

ALTER USER name [[WITH] option [...]]
ALTER USER name RENAME TO new_name
ALTER USER name SET parameter { TO | = } { value | DEFAULT }
ALTER USER name RESET parameter

Donde option puede valer:

[ENCRYPTED | UNENCRYPTED] PASSWORD 'password'
| CREATEDB | NOCREATEDB
| CREATEUSER | NOCREATEUSER
| VALID UNTIL 'abstime'

-ANALYZE

Recolecta estadísticas desde la base de datos.

ANALYZE [VERBOSE] [table [(column [, ...])]]

-BEGIN

Comienza un bloque de transacción.

BEGIN [WORK | TRANSACTION] [transaction_mode [, ...]]

Donde transaction_mode puede valer:

ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }

```
READ WRITE | READ ONLY
```

-CHECKPOINT

Fuerza un puesto de control al registro de transacciones.

```
CHECKPOINT
```

-CLOSE

Cierra un cursor.

```
CLOSE name
```

-CLUSTER

Agrupa una tabla de acuerdo a un índice.

```
CLUSTER index_name ON table_name
CLUSTER table_name
CLUSTER
```

-COMMENT

Define o cambia el comentario de un objeto.

```
COMMENT ON
{
        TABLE object_name |
        COLUMN table_name.column_name |
        AGGREGATE agg_name (agg_type) |
        CAST (source_type AS target_type) |
        CONSTRAINT constraint_name ON table_name |
        CONVERSION object_name |
        DATABASE object_name |
        DOMAIN object_name |
        FUNCTION func_name (arg1_type, arg2_type, ...) |
        INDEX object_name |
        LARGE OBJECT large_object_oid |
        OPERATOR op (left_operand_type, right_operand_type) |
        OPERATOR CLASS object_name USING index_method |
        [ PROCEDURAL ] LANGUAGE object_name |
        RULE rule_name ON table_name |
        SCHEMA object_name |
        SEQUENCE object_name |
        TRIGGER trigger_name ON table_name |
        TYPE object_name |
        VIEW object_name
} IS 'text'
```

-COMMIT

Confirma la transacción actual.

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

-COPY

Copia datos entre archivo y una tabla.

```
COPY table_name [ ( column [, ...] ) ]
        FROM { 'filename' | STDIN }
        [ [ WITH ]
            [ BINARY ]
            [ OIDS ]
            [ DELIMITER [ AS ] 'delimiter' ]
            [ NULL [ AS ] 'null string' ]
            [ CSV [ QUOTE [ AS ] 'quote' ]
            [ ESCAPE [ AS ] 'escape' ]
            [ FORCE NOT NULL column [, ...] ]
```

```
COPY table_name [ ( column [, ...] ) ]
        TO { 'filename' | STDOUT }
        [ [ WITH ]
            [ BINARY ]
            [ OIDS ]
            [ DELIMITER [ AS ] 'delimiter' ]
            [ NULL [ AS ] 'null string' ]
            [ CSV [ QUOTE [ AS ] 'quote' ]
            [ ESCAPE [ AS ] 'escape' ]
            [ FORCE QUOTE column [, ...] ]
```

-CREATE AGGREGATE

Define una función agregada nueva.

```
CREATE AGGREGATE name (
        BASETYPE = input_data_type,
        SFUNC = sfunc,
        STYPE = state_data_type
        [ , FINALFUNC = ffunc ]
        [ , INITCOND = initial_condition ]
)
```

-CREATE CAST

Define un casting nuevo.

```
CREATE CAST (source_type AS target_type)
        WITH FUNCTION func_name (arg_types)
        [ AS ASSIGNMENT | AS IMPLICIT ]
```

```
CREATE CAST (source_type AS target_type)
        WITHOUT FUNCTION
        [ AS ASSIGNMENT | AS IMPLICIT ]
```

-CREATE CONSTRAINT TRIGGER

Define un nuevo trigger.

```
CREATE CONSTRAINT TRIGGER name
        AFTER events ON
        table_name constraint attributes
        FOR EACH ROW EXECUTE PROCEDURE func_name ( args )
```

-CREATE CONVERSION

Define una conversión nueva.

```
CREATE [DEFAULT] CONVERSION name
        FOR source_encoding TO dest_encoding FROM func_name
```

-CREATE DATABASE

Crea una base de datos nueva.

```
CREATE DATABASE name
        [ [ WITH ] [ OWNER [=] db_owner ]
                [ TEMPLATE [=] template ]
                [ ENCODING [=] encoding ]
                [ TABLESPACE [=] tablespace ] ]
```

-CREATE DOMAIN

Define un dominio nuevo.

```
CREATE DOMAIN name [AS] data_type
        [ DEFAULT expression ]
        [ constraint [ ... ] ]
```

Donde constraint puede valer:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }
```

-CREATE FUNCTION

Define una función nueva.

```
CREATE [ OR REPLACE ] FUNCTION name ( [ [ arg_name ] arg_type [, ...] ] )
        RETURNS ret_type
        { LANGUAGE lang_name
        | IMMUTABLE | STABLE | VOLATILE
        | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
        | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
        | AS 'definition'
        | AS 'obj_file', 'link_symbol'
} ...
[ WITH ( attribute [, ...] ) ]
```

-CREATE GROUP

Define un grupo de usuario nuevo.

```
CREATE GROUP name [ [ WITH ] option [ ... ] ]
```

Donde option puede valer:

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

-CREATE INDEX

Define un índice nuevo.

```
CREATE [ UNIQUE ] INDEX name ON table [ USING method ]
        ( { column | ( expression ) } [ opclass ] [, ...] )
        [ TABLESPACE tablespace ]
```

```
        [ WHERE predicate ]
```

-CREATE LANGUAGE

Define un nuevo lenguaje procedural.

```
CREATE [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
        HANDLER call_handler [ VALIDATOR val_function ]
```

-CREATE OPERATOR

Define un operador nuevo.

```
CREATE OPERATOR name (
        PROCEDURE = func_name
        [, LEFTARG = left_type ] [, RIGHTARG = right_type ]
        [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
        [, RESTRICT = res_proc ] [, JOIN = join_proc ]
        [, HASHES ] [, MERGES ]
        [, SORT1 = left_sort_op ] [, SORT2 = right_sort_op ]
        [, LTCMP = less_than_op ] [, GTCMP = greater_than_op ]
)
```

-CREATE OPERATOR CLASS

Define una clase de operador nueva.

```
CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE data_type
        USING index_method AS
        { OPERATOR strategy_number operator_name [ ( op_type, op_type ) ] [ RECHECK ]
        | FUNCTION support_number func_name ( argument_type [, ...] )
        | STORAGE storage_type
} [, ... ]
```

-CREATE RULE

Define una regla de reescritura nueva.

```
CREATE [ OR REPLACE ] RULE name AS ON event
        TO table [ WHERE condition ]
        DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

-CREATE SCHEMA

Define un esquema nuevo.

```
CREATE SCHEMA schema_name
[ AUTHORIZATION username ] [ schema_element [ ... ] ]
```

```
CREATE SCHEMA AUTHORIZATION username
[ schema_element [ ... ] ]
```

-CREATE SEQUENCE

Define un nuevo generador de secuencia.

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE name
        [ INCREMENT [ BY ] increment ]
        [ MINVALUE minvalue | NO MINVALUE ]
        [ MAXVALUE maxvalue | NO MAXVALUE ]
```

```
        [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
```

-CREATE TABLE

Define una tabla nueva.

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name (
        { column_name data_type [ DEFAULT default_expr ] [ column_constraint [ ... ] ]
        | table_constraint
        | LIKE parent_table [ { INCLUDING | EXCLUDING } DEFAULTS ] } [, ... ]
        )
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace ]
```

Donde column_constraint puede valer:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
        NULL |
        UNIQUE [ USING INDEX TABLESPACE tablespace ] |
        PRIMARY KEY [ USING INDEX TABLESPACE tablespace ] |
        CHECK (expression) |
        REFERENCES ref_table [ ( ref_column ) ]
                [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
                [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

Y table_constraint es:

```
[ CONSTRAINT constraint_name ]
{ UNIQUE ( column_name [, ... ] ) [ USING INDEX TABLESPACE tablespace ] |
        PRIMARY KEY ( column_name [, ... ] ) [ USING INDEX TABLESPACE tablespace ] |
        CHECK ( expression ) |
        FOREIGN KEY ( column_name [, ... ] )
                REFERENCES ref_table [ ( ref_column [, ... ] ) ]
                [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
                [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

-CREATE TABLE AS

Define una nueva tabla a partir de los resultados de una consulta.

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name
        [ (column_name [, ...] ) ] [ [ WITH | WITHOUT ] OIDS ]
        AS query
```

-CREATE TABLESPACE

Define un tablespace nuevo.

```
CREATE TABLESPACE tablespace_name [ OWNER username ] LOCATION 'directory'
```

-CREATE TRIGGER

Define un trigger nuevo.

```
CREATE TRIGGER name { BEFORE | AFTER } { event [ OR ... ] }
```

```
        ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
        EXECUTE PROCEDURE func_name ( arguments )
```

-CREATE TYPE

Define un nuevo tipo de datos.

```
CREATE TYPE name AS
        ( attribute_name data_type [, ... ] )
```

```
CREATE TYPE name (
        INPUT = input_function,
        OUTPUT = output_function
        [ , RECEIVE = receive_function ]
        [ , SEND = send_function ]
        [ , ANALYZE = analyze_function ]
        [ , INTERNALLENGTH = { internal_length | VARIABLE } ]
        [ , PASSEDBYVALUE ]
        [ , ALIGNMENT = alignment ]
        [ , STORAGE = storage ]
        [ , DEFAULT = default ]
        [ , ELEMENT = element ]
        [ , DELIMITER = delimiter ]
)
```

-CREATE USER

Define una nueva cuenta de usuario de base de datos.

```
CREATE USER name [ [ WITH ] option [ ... ] ]
```

Donde option puede valer:

```
SYSID uid
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
| CREATEDB | NOCREATEDB
| CREATEUSER | NOCREATEUSER
| IN GROUP group_name [, ...]
| VALID UNTIL 'abs_time'
```

-CREATE VIEW

Define una vista nueva.

```
CREATE [ OR REPLACE ] VIEW name [ ( column_name [, ...] ) ] AS query
```

-DEALLOCATE

Cancela la asignación de una sentencia preparada.

```
DEALLOCATE [ PREPARE ] plan_name
```

-DECLARE

Define un cursor.

```
DECLARE name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
        CURSOR [ { WITH | WITHOUT } HOLD ] FOR query
        [ FOR { READ ONLY | UPDATE [ OF column [, ...] ] } ]
```

-DELETE

Borra filas en una tabla.

```
DELETE FROM [ ONLY ] table [ WHERE condition ]
```

-DROP AGGREGATE

Borra una función agregada.

```
DROP AGGREGATE name ( type ) [ CASCADE | RESTRICT ]
```

-DROP CAST

Borra un casting.

```
DROP CAST (source_type AS target_type) [ CASCADE | RESTRICT ]
```

-DROP CONVERSION

Borra una conversión.

```
DROP CONVERSION name [ CASCADE | RESTRICT ]
```

-DROP DATABASE

Borra una base de datos.

```
DROP DATABASE name
```

-DROP DOMAIN

Borra un dominio.

```
DROP DOMAIN name [, ...] [ CASCADE | RESTRICT ]
```

-DROP FUNCTION

Borra una función.

```
DROP FUNCTION name ( [ type [, ...] ] ) [ CASCADE | RESTRICT ]
```

-DROP GROUP

Borra un grupo de usuarios.

```
DROP GROUP name
```

-DROP INDEX

Borra un índice.

```
DROP INDEX name [, ...] [ CASCADE | RESTRICT ]
```

-DROP LANGUAGE

Borra un lenguaje procedural.

```
DROP [ PROCEDURAL ] LANGUAGE name [ CASCADE | RESTRICT ]
```

-DROP OPERATOR

Borra un operador.

```
DROP OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
        [ CASCADE | RESTRICT ]
```

-DROP OPERATOR CLASS

Borra una clase de operador.

```
DROP OPERATOR CLASS name USING index_method [ CASCADE | RESTRICT ]
```

-DROP RULE

Borra una regla de escritura.

```
DROP RULE name ON relation [ CASCADE | RESTRICT ]
```

-DROP SCHEMA

Borra un esquema.

```
DROP SCHEMA name [, ...] [ CASCADE | RESTRICT ]
```

-DROP SEQUENCE

Borra una secuencia.

```
DROP SEQUENCE name [, ...] [ CASCADE | RESTRICT ]
```

-DROP TABLE

Borra una tabla.

```
DROP TABLE name [, ...] [ CASCADE | RESTRICT ]
```

-DROP TABLESPACE

Borra un tablespace.

```
DROP TABLESPACE tablespace_name
```

-DROP TRIGGER

Borra un trigger.

```
DROP TRIGGER name ON table [ CASCADE | RESTRICT ]
```

-DROP TYPE

Borra un tipo de datos.

```
DROP TYPE name [, ...] [ CASCADE | RESTRICT ]
```

-DROP USER

Borra una cuenta de usuario de base de datos.

```
DROP USER name
```

-DROP VIEW

Borra una vista.

```
DROP VIEW name [, ...] [ CASCADE | RESTRICT ]
```

-END

Confirma la transacción actual.

```
END [ WORK | TRANSACTION ]
```

-EXECUTE

Ejecuta una sentencia preparada.

```
EXECUTE plan_name [ (parameter [, ...] ) ]
```

-EXPLAIN

Muestra el plan de ejecución de una sentencia.

```
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

-FETCH

Recupera las filas de una consulta mediante un cursor.

```
FETCH [ direction { FROM | IN } ] cursor_name
```

Donde direction puede estar vacío o puede valer:

```
NEXT
PRIOR
FIRST
LAST
ABSOLUTE count
RELATIVE count
count
ALL
FORWARD
FORWARD count
FORWARD ALL
BACKWARD
BACKWARD count
BACKWARD ALL
```

-GRANT

Define privilegios de acceso.

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIGGER }
    [,...] | ALL [ PRIVILEGES ] }
    ON [ TABLE ] table_name [, ...]
    TO { username | GROUP group_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { { CREATE | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
    ON DATABASE db_name [, ...]
    TO { username | GROUP group_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespace_name [, ...]
    TO { username | GROUP group_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTION func_name ([type, ...]) [, ...]
    TO { username | GROUP group_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE lang_name [, ...]
    TO { username | GROUP group_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    TO { username | GROUP group_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

-INSERT

Crea filas nuevas en una tabla.

```
INSERT INTO table [ ( column [, ...] ) ]
        { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) | query }
```

-LISTEN

Espera ante una notificación.

```
LISTEN name
```

-LOAD

Carga o recarga un archivo de librería compartida.

```
LOAD 'filename'
```

-LOCK

Bloquea una tabla.

```
LOCK [ TABLE ] name [, ...] [ IN lock_mode MODE ] [ NOWAIT ]
```

Donde lock_mode puede valer:

```
ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
| SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

-MOVE

Posiciona un cursor.

```
MOVE [ direction { FROM | IN } ] cursor_name
```

-NOTIFY

Genera una notificación.

```
NOTIFY name
```

-PREPARE

Prepara una sentencia para ser ejecutada.

```
PREPARE plan_name [ (data_type [, ...] ) ] AS statement
```

-REINDEX

Recontruye índices.

```
REINDEX { DATABASE | TABLE | INDEX } name [ FORCE ]
```

-RELEASE SAVEPOINT

Destruye un punto de retorno previamente definido.

```
RELEASE [ SAVEPOINT ] savepoint_name
```

-RESET

Restaura el valor de un parámetro de tiempo de ejecución a su valor por defecto.

```
RESET name
```

```
RESET ALL
```

-REVOKE

Revoca privilegios de acceso.

```
REVOKE [ GRANT OPTION FOR ]
        { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIGGER }
        [,...] | ALL [ PRIVILEGES ] }
        ON [ TABLE ] table_name [, ...]
        FROM { username | GROUP group_name | PUBLIC } [, ...]
        [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
        { { CREATE | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
        ON DATABASE db_name [, ...]
        FROM { username | GROUP group_name | PUBLIC } [, ...]
        [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
        { CREATE | ALL [ PRIVILEGES ] }
        ON TABLESPACE tablespace_name [, ...]
        FROM { username | GROUP group_name | PUBLIC } [, ...]
        [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
        { EXECUTE | ALL [ PRIVILEGES ] }
        ON FUNCTION func_name ([type, ...]) [, ...]
        FROM { username | GROUP group_name | PUBLIC } [, ...]
        [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
        { USAGE | ALL [ PRIVILEGES ] }
        ON LANGUAGE lang_name [, ...]
        FROM { username | GROUP group_name | PUBLIC } [, ...]
        [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
        { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
        ON SCHEMA schema_name [, ...]
        FROM { username | GROUP group_name | PUBLIC } [, ...]
        [ CASCADE | RESTRICT ]
```

-ROLLBACK

Aborta la transacción actual.

```
ROLLBACK [ WORK | TRANSACTION ]
```

-ROLLBACK TO SAVEPOINT

Retrocede a un punto de retorno.

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name
```

-SAVEPOINT

Define un nuevo punto de retorno dentro de la transacción actual.

```
SAVEPOINT savepoint_name
```

-SELECT

Recupera las filas de una tabla o vista.

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
```

```
        * | expression [ AS output_name ] [, ...]
        [ FROM from_item [, ...] ]
        [ WHERE condition ]
        [ GROUP BY expression [, ...] ]
        [ HAVING condition [, ...] ]
        [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
        [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
        [ LIMIT { count | ALL } ]
        [ OFFSET start ]
        [ FOR UPDATE [ OF table_name [, ...] ] ]
```

Donde from_item puede valer:

```
[ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
function_name ( [ argument [, ...] ] )
        [ AS ] alias [ ( column_alias [, ...] | column_definition [, ...] ) ]
function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
from_item [ NATURAL ] join_type from_item
        [ ON join_condition | USING ( join_column [, ...] ) ]
```

-SELECT INTO

Define una nueva tabla a partir de los resultados de una consulta.

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
        * | expression [ AS output_name ] [, ...]
        INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table
        [ FROM from_item [, ...] ]
        [ WHERE condition ]
        [ GROUP BY expression [, ...] ]
        [ HAVING condition [, ...] ]
        [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
        [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
        [ LIMIT { count | ALL } ]
        [ OFFSET start ]
        [ FOR UPDATE [ OF table_name [, ...] ] ]
```

-SET

Cambia un parámetro de tiempo de ejecución.

```
SET [ SESSION | LOCAL ] name { TO | = } { value | 'value' | DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { time_zone | LOCAL | DEFAULT }
```

-SET CONSTRAINTS

Configura los modos de configuración de restricciones para la transacción actual.

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

-SET SESSION AUTHORIZATION

Configura el identificador de la sesión de usuario y el identificador de usuario actual de la sesión actual.

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION username
```

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

-SET TRANSACTION

Configura las características de la transacción actual.

```
SET TRANSACTION transaction_mode [, ...]
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]
```

Donde transaction_mode puede valer:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED
        | READ UNCOMMITTED }
READ WRITE | READ ONLY
```

-SHOW

Muestra el valor de un parámetro de tiempo de ejecución.

```
SHOW name
SHOW ALL
```

-START TRANSACTION

Comienza un bloque de transacción.

```
START TRANSACTION [ transaction_mode [, ...] ]
```

Donde transaction_mode puede valer:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED
        | READ UNCOMMITTED }
READ WRITE | READ ONLY
```

-TRUNCATE

Vacía una tabla.

```
TRUNCATE [ TABLE ] name
```

-UNLISTEN

Para la espera ante una notificación.

```
UNLISTEN { name | * }
```

-UPDATE

Actualiza filas de una tabla.

```
UPDATE [ ONLY ] table SET column = { expression | DEFAULT } [, ...]
        [ FROM from_list ]
        [ WHERE condition ]
```

-VACUUM

Recoge datos y opcionalmente analiza una base de datos.

```
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ table [ (column [, ...] ) ] ]
```

BIBLIOGRAFÍA

• [2] “Beggining Databases with PostgreSQL: From Novice to Professional, Second Edition”

APÉNDICE B – COMPARATIVA DE SISTEMAS DE GESTIÓN DE BASES DE DATOS

SYBASE

-Nombre (Año fundación):

- Sybase (1984)

-Ficha técnica:

- Creado en 1984 por Mark Hoffman y Bob Epstein
- Enfocado a inteligencia empresarial
- Multiplataforma

-Características:

- Consultas hasta 100 veces mas rápida que un sistema gestor de base de datos tradicional
- Seguridad tanto en autentificación para usuarios como en servidores

-Ventajas:

- Menor costo
- Facilidad de uso
- Escalabilidad
- Flexibilidad

-Desventajas:

- No es adecuado para transacciones on-line.
- Limitaciones de programación

POSTGRESQL

-Nombre (Año fundación):

- PostgreSQL (1982)

-Ficha técnica:

- Distribuida bajo licencia BSD
- Ultima versión PostgreSQL 9.1.2
- Escrito en c
- Multiplataforma

-Características:

- Código libre

- Es una base de datos 100% ACID.
- Joins, vistas, triggers
- Interfaz para programar en C, Java, .Net, Perl, Python, Ruby, Tcl, ODBC, PHP, ...
- Cuenta con herramientas de diseño

-Ventajas:

- Funciona con grandes cantidades de datos
- Alta concurrencia con varios usuarios accediendo al mismo tiempo al mismo sistema
- Ahorro de costes de operación
- Estabilidad

-Desventajas:

- Es más lento en inserciones y actualizaciones que mysql

-Empresas que lo utilizan:

- Yahoo!
- Geni.com
- OpenStreetMap
- Afilias
- Sony Online
- BASF
- Reddit
- Skype
- Sun xVM
- MusicBrainZ
- International Space Station
- Heroku
- Instagram

NEXUSDB

-Nombre (Año fundación):

- Nexusdb (1986)

-Ficha técnica:

- Creado por NexusDB Pty Ltd
- Última versión 3.10
- Escrito en Delphi, C#
- Sistema operativo de Windows
- Licencia comercial

-Características:

- El motor de base de datos para Delphi
- Motor de base de datos ultra rápida cliente/servidor
- Herramientas entuitivas para desarrolladores
- Compatibilidad con sql
- Transporte conectividad en tiempo real

-Ventajas:

- Estabilidad
- Rendimiento
- Flexibilidad
- Motor de base de datos ultra rápida cliente/servidor
- Estabilidad con el soporte de transacciones

-Empresas que lo utilizan:

- Burger King
- IBM
- Motorola
- Honda
- metro group
- Shell
- forotime
- KPMG

SQL SERVER

-Nombre (Año fundación):

- Microsoft SQL Server (1989)

-Ficha técnica:

- Desarrollado por Microsoft
- Última versión 2012
- Plataforma para Microsoft

-Características:

- Soporte de transacciones
- Procedimientos Almacenados
- permite administrar información de otros servidores de datos

-Ventajas:

- Escalabilidad

- Seguridad
- Estabilidad

-Desventajas:

- Solo permite alojar un máximo de 64 GB.
- Requiere de un sistema operativo de Windows
- No se puede instalar en servidores Linux

-Empresas que lo utilizan:

- Ministerio de Fomento del Gobierno de España
- NASDAQ
- CEPSA
- Jettainer
- Xerox
- Compañía Mediterránea de Navegación

VOLTDB

-Nombre (Año fundación):

- VoltDB (2010)

-Ficha técnica:

- Desarrollado por VoltDB Inc.
- Última versión 2.8
- Escrito en Java, C++.
- Plataforma Linux y MacOS
- Licencia GNU comercial

-Características:

- VoltDB es una base de datos relacional que soporta SQL de acceso desde el interior de scripts compilados almacenados en java
- Alto rendimiento en memoria
- Completa compatibilidad ACID
- Los datos se almacenan en memoria en vez de hacerlo en disco
- Protección de datos en transacciones concurrentes

-Ventajas:

- Residente de memoria
- Particiona la base de datos junto con un motor SQL
- Puede procesar hasta 45 veces más transacciones por segundo que MySQL, Oracle y PostgreSQL
- Se distingue por ofrecer rendimiento a aplicaciones de alta velocidad

-Desventajas:
- Se debe ejecutar en servidores con mucha memoria
- Solo se ejecuta en Linux o MacOS

-Empresas que lo utilizan:
- Cowen
- Shopzilla
- JasperLabs
- Cloudera
- Booyah!
- SAKURA Internet

FIREBIRD

-Nombre (Año fundación):
- Firebird

-Ficha técnica:
- Desarrollado por Firebird Project
- Última versión 2.5.1
- Multiplataforma
- Licencia IPL,IDPL
- Escrito en C++

-Características:
- Ejecutable pequeño, con requerimientos de hardware bajos.

-Ventajas:
- Buena Seguridad Basada en usuarios/roles
- Soporte de transacciones ACID y claves ajenas

-Desventajas:
- Es mediamente estable.

PROGRESS DATABASE

PROGRESS SOFTWARE

-Nombre (Año fundación):
- Progress Database (1981)

-Ficha técnica:
- Plataforma Windows

-Características:
- Bases de datos en servidor central
- Recuperación primaria de datos

-Ventajas:
- Respaldo de de base de datos

-Desventajas:
- Es mediamente estable.

-Empresas que lo utilizan:
- Coca-Cola

LUCIDDB

-Nombre (Año fundación):
- LucidDB (-)

-Ficha técnica:
- Autor Eigenbase Fundation
- Última versión 0.9.4
- Escrito en java, c++
- Licencia GPL 2

INFORMIX

-Nombre (Año fundación):
- IBM Informix (1980)

-Ficha técnica:
- Desarrollo por IBM
- Última versión 11.7
- Programado en C, C++
- Multiplataforma
- Licencia propietaria

-Características:

- Basado en SQL
- Lenguaje de cuarta generación
- Dispone de herramientas graficas
- Cumple con los niveles de seguridad

-Ventajas:

- Reduce los costes de administración
- Soporta requisitos de procesamiento de transacción online
- Maximiza operaciones de datos para el grupo de trabajo y para la empresa total

-Desventajas:

- No es recomendable utilizarlo con aplicaciones que exigen un gran rendimiento
- No tiene soporte para tipo de datos VARCHAR (son datos con una longitud fija de 2000 caracteres)

-Empresas que lo utilizan:

- IBM
- WAL-MART

INTERBASE

-Nombre (Año fundación):

- InterBase (1981)

-Ficha técnica:

- Desarrollado por Embarcadero Techonologies
- Versión 10(XE)
- Multiplataforma
- Licencia Propietaria

-Características:

- Tiene muchos años de experiencia
- Código Abierto
- Mantenimiento prácticamente nulo
- Tráfico de red reducido

-Ventajas:

- Escalabilidad

-Desventajas:

- Conexión a internet

MYSQL

-Nombre (Año fundación):

- MySQL (1995)

-Ficha técnica:

- Desarrollado por Sun Microsystems.
- Última versión 5.5.27
- Programado en C, C++
- Multiplataforma
- GPL o uso comercial

-Características:

- Amplio subconjunto del lenguaje SQL
- Operaciones de Indexación Online
- Particionado de Datos

-Ventajas:

- Conectividad segura
- Disponibilidad en gran cantidad de plataformas y sistemas
- Soporte de transacciones
- Escalabilidad, estabilidad y seguridad

-Desventajas:

- La principal desventaja es la gran cantidad de memoria RAM que utiliza para la instalación

-Empresas que lo utilizan:

- Alcatel Lucent
- Zappos
- FAO
- Universidad de Kent
- Banco de Finlandia
- Policía Nacional de Suecia
- Wikipedia
- Drupal
- OpenLDAP
- Big Fish Games
- Symantec
- S2 Security Corporation
- Booking.com

SQLITE

-Nombre (Año fundación):

- SQLite (2000)

-Ficha técnica:

- Diseñado por Richard Hipp
- Última versión 3.7.14
- Programado en C.
- Multiplataforma.
- Dominio Público

-Características:

- Consistencia De base datos
- ACID

-Ventajas:

- Aislamiento
- Durabilidad
- Puede implementarse en sistemas operativos con pocos recursos como Android, Blakberry, Google Chrome,…
- Simplicidad y sencillez

-Desventajas:

- El modelo tradicional de utilizar un proceso servidor ofrece mayor protección ante aplicaciones que utilizan la base de datos y que pudieran tener fallos de programación

-Empresas que lo utilizan:

- Adobe Photoshop
- Mozila Firefox
- Skipe

DB2

-Nombre (Año fundación):

- IBM DB2 (1983)

-Ficha técnica:

- Desarrollado por IBM

- Ultima versión 10.2
- Multiplataforma
- Licencia privada

-Características:

- DB2 es el producto principal de la estrategia de IBM para servidor de bases de datos relacionales

-Ventajas:

- Permite agilizar el tiempo de respuestas de la consulta
- Tablas de resumen
- La mayoría de los que usan equipos IBM utilizan BD2 porque tiene Soporte técnico

-Desventajas:

- Según la puntuación en un artículo de la revista VAR, Microsoft SQL Server se anoto un 38%, IBM 10%, Oracle 21%, Infromix 9% y Sybase 8%

ORACLE

-Nombre (Año fundación):

- Oracle Database (1979)

-Ficha técnica:

- Desarrollado por Oracle Corporation
- Última versión 11g
- Multiplataforma
- Licencia privada

-Características:

- Incluye una herramienta de administración gráfica muy intuitiva y cómoda de manejar
- Optimiza el de modelos de datos

-Ventajas:

- Multiplataforma: Soporta bases de datos de todos los tamaños
- Soporta Cliente/Servidor

-Desventajas:

- Costo de mantenimiento alto
- Requiere conocimientos propios de Oracle que son poco estándar

-Empresas que lo utilizan:

- General Motors
- HP
- Toyota
- Philips
- Mercado Libre
- Boing

BIBLIOGRAFÍA

- [8] http://es.scribd.com/doc/81543140/cuadro-comparativo
- [5] http://en.wikipedia.org/wiki/Sybase
- [5] http://en.wikipedia.org/wiki/Postgresql
- [9] http://www.nexusdb.com/support/index.php?q=node/506
- [5] http://en.wikipedia.org/wiki/Nexusdb
- [5] http://en.wikipedia.org/wiki/Microsoft_SQL_Server
- [10] http://www.microsoft.com/spain/sql/productinfo/casestudies/default.mspx
- [5] http://en.wikipedia.org/wiki/VoltDB
- [5] http://en.wikipedia.org/wiki/Firebird_%28database_server%29
- [5] http://en.wikipedia.org/wiki/Progress_Software
- [5] http://en.wikipedia.org/wiki/LucidDB
- [5] http://en.wikipedia.org/wiki/Informix_Corporation
- [5] http://en.wikipedia.org/wiki/Informix
- [5] http://en.wikipedia.org/wiki/Interbase
- [5] http://en.wikipedia.org/wiki/MySQL
- [11] http://www.mysql.com/customers/
- [5] http://en.wikipedia.org/wiki/SQLITE
- [5] http://en.wikipedia.org/wiki/IBM_DB2

APÉNDICE C – PGADMIN III

¿QUÉ ES?

La herramienta pgAdmin es una aplicación que ofrece el grupo de PostgreSQL para la administración de bases de datos PostgreSQL e incluye:

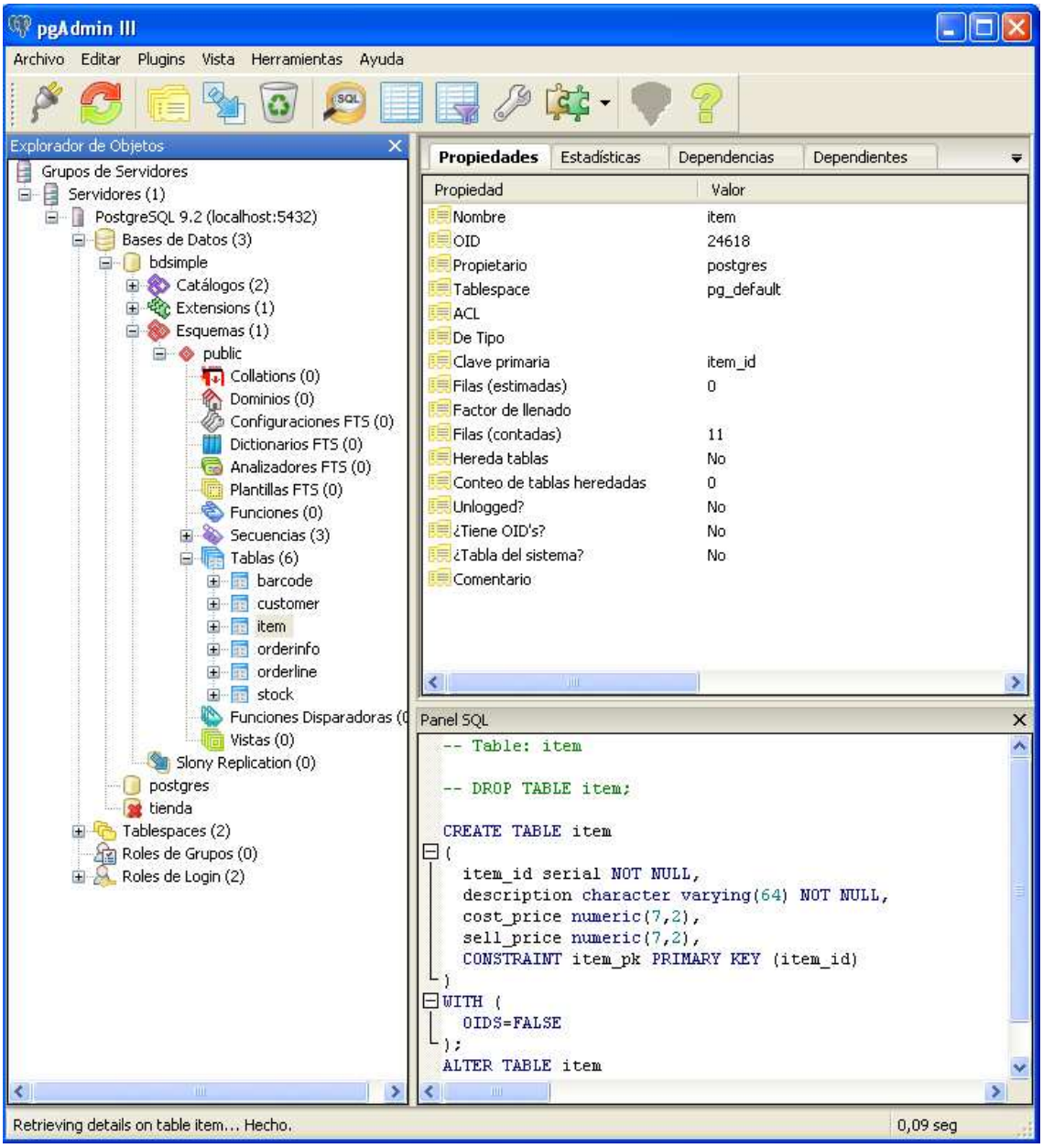
- Interfaz administrativa gráfica
- Herramienta de consulta SQL (con un EXPLAIN gráfico)
- Editor de código procedural
- Agente de planificación SQL/Shell/batch

PgAdmin se diseña para responder a las necesidades de la mayoría de los usuarios, desde simples consultas SQL hasta desarrollar bases de datos complejas. La interfaz gráfica soporta todas las características de PostgreSQL y hace simple la administración. Está disponible en más de una docena de idiomas y para varios sistemas operativos, incluyendo Windows, Linux, FreeBSD, MacOS y Solaris.

El paquete pgAdmin, gratuito y de código abierto, es una poderosa plataforma para administrar y desarrollar bases de datos de PostgreSQL, y la página web del proyecto es http://www.pgadmin.org. El primer prototipo, llamado pgManager, fue desarrollado para PostgreSQL 6.3.2 en 1998, y en unos meses más tarde renombrado a pgAdmin. En 2002 salió la siguiente versión, llamada pgAdmin II. La versión actual es pgAdmin III, y está programada en C++ (las anteriores estaban programadas en Visual Basic). Al ser un programa gratuito, se puede descargar fácilmente desde su web http://www.pgadmin.es, así como integrado en las versiones actuales para Windows de PostgreSQL.

La herramienta pgAdmin III ofrece muchas funcionalidades:

- Crear y borrar tablespaces (forma de almacenamiento), bases de datos, tablas y esquemas.
- Ejecutar SQL mediante una ventana de consultas.
- Exportar los resultados de las consultas SQL a archivos.
- Gestionar copias de seguridad, y restaura bases de datos enteras o tablas individuales.
- Configura usuarios, grupos, y privilegios.
- Visualiza, edita e inserta datos en tablas.



INSTALACIÓN

Con pgAdmin III la instalación es mucho más sencilla que con las versiones anteriores, ya que requerían que el controlador de ODBC para PostgreSQL estuviese instalado para acceder a la base de datos, pero esta dependencia ya no existe. La versión para Windows de PostgreSQL ya incluye una versión de pgAdmin III para ser instalada en un servidor Windows, pero aun así se puede descargar desde http://www.pgadmin.org/pgadmin3/download.php.

Antes de usar pgAdmin III por primera vez, hay que asegurarse que se pueden crear objetos en la base de datos, ya que pgAdmin III crea objetos propios en la base de datos almacenados en el servidor. Para que pgAdmin III pueda utilizar todas sus funciones de mantenimiento, se

necesita acceder con un usuario que tenga privilegios totales sobre la base de datos (un superusuario), o habrá un error.

Se puede manejar varios servidores de base de datos a la vez desde pgAdmin III, por lo que es importante crear una conexión al servidor en la primera ejecución. Desde el menú 'Archivo', opción 'Añadir servidor', se obtiene una ventana donde habrá que escribir los parámetros del servidor. Una vez creada la conexión correctamente, ya es posible conectarse al servidor de base de datos y navegar por la base de datos, tablas, y otros objetos.

Una de las herramientas más útiles es la de poder restaurar información. Esta herramienta se lleva a cabo con la utilidad 'pg_dump'. Se puede recuperar y restaurar tablas individuales o una base de datos entera. Tiene opciones para controlar cómo y dónde se crea el archivo de restauración, y qué método usará.

VENTANA PRINCIPAL

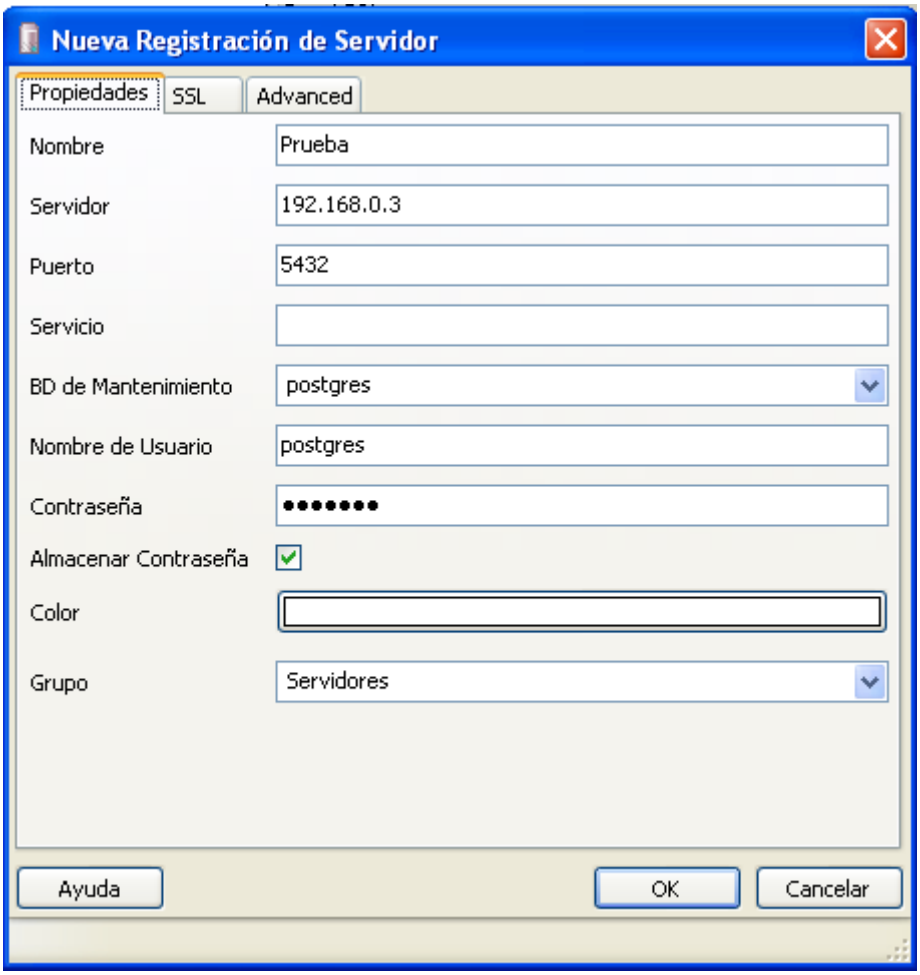
Una vez abierto pgAdmin III, la ventana principal muestra la siguiente estructura de la bases de datos:

- Barra de menú con las distintas funcionalidades de la herramienta.
- Barra de herramientas (que actuarán sobre los objetos seleccionados).
- Explorador de objetos: árbol con las bases de datos definidas y su contenido.
- Panel de detalle: pestañas de Propiedades, Estadísticas, Dependencias y Dependientes del objeto seleccionado.
- Panel SQL: sentencias SQL generadas mediante ingeniería inversa sobre el objeto seleccionado.

Para abrir una conexión con un servidor de base de datos PostgreSQL, hay que seleccionarlo en el explorador de objetos y hacer doble clic. Si no está registrado previamente, habrá que agregarlo.

-Agregar servidor:

Para conectarse a un servidor, se debe agregar los datos del mismo mediante el botón 'Añadir una conexión a un servidor" (icono de enchufe en la barra de herramientas), o la opción de menú de archivo 'Añadir Servidor', con lo que aparecerá la pantalla de 'Nueva registración de Servidor'.

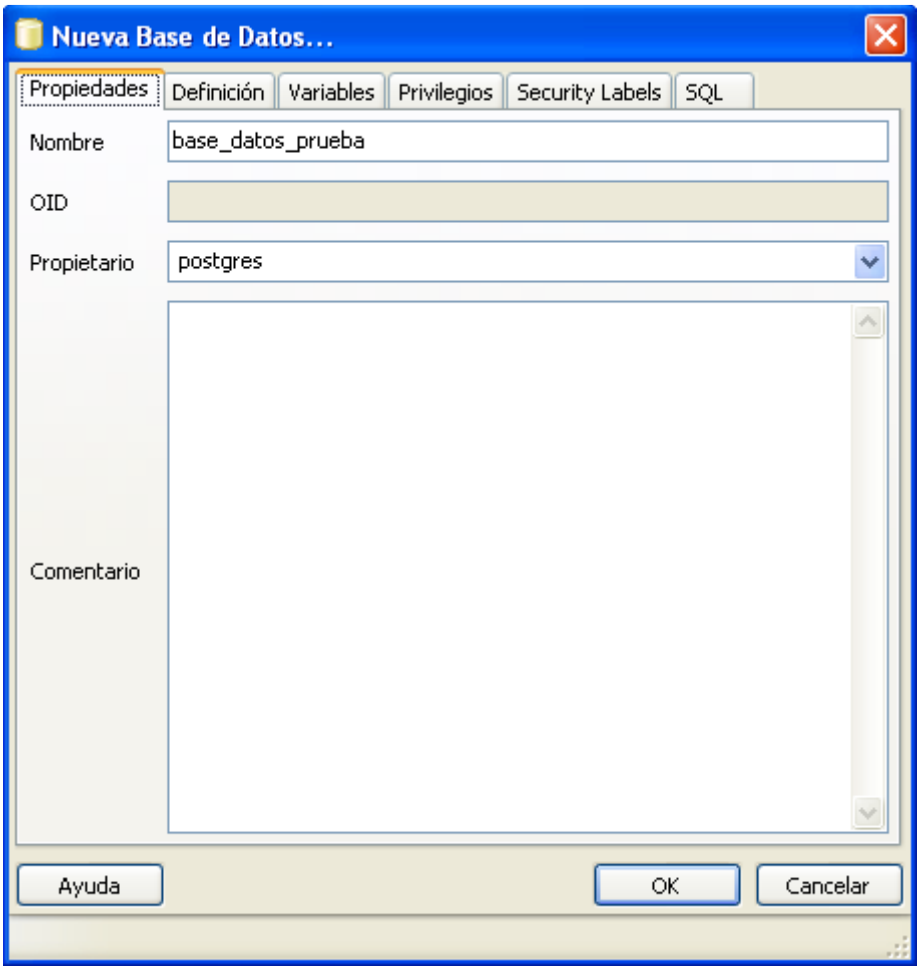


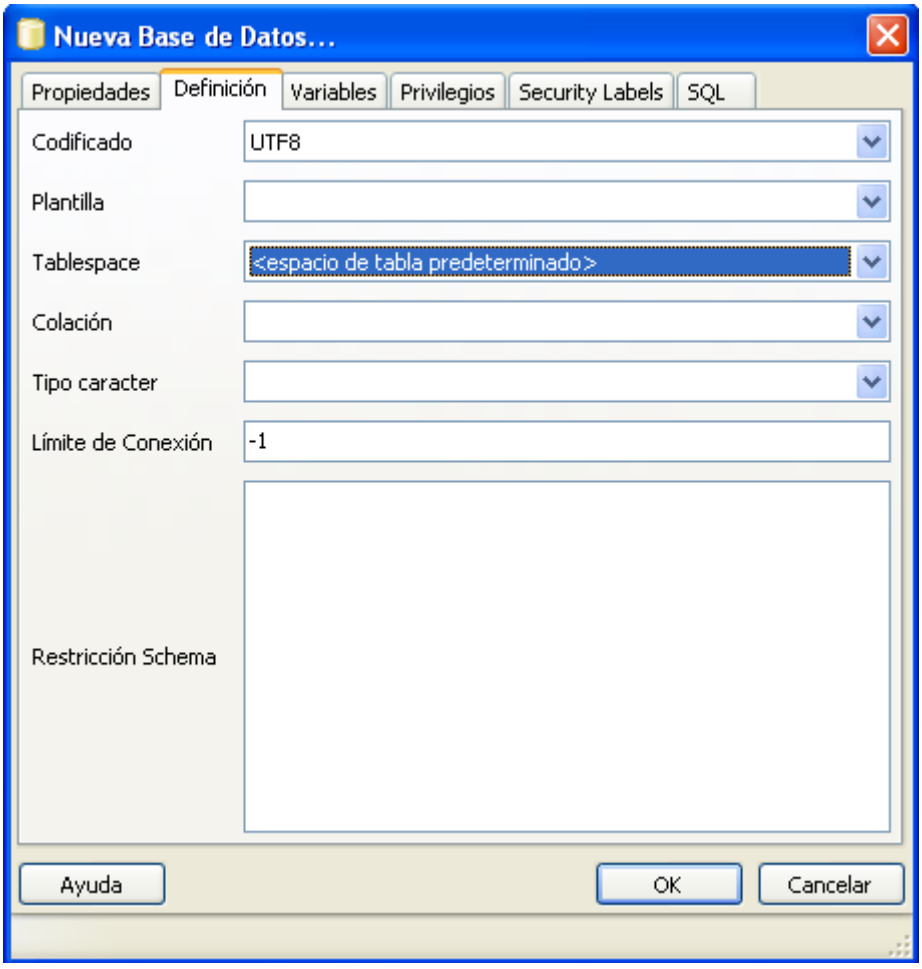
Los campos importantes para rellenar son:

- Nombre: denominación con la que pgAdmin conocerá al servidor.
- Servidor: dirección IP o nombre de host.
- Puerto: número de puerto de escucha del servidor (el estándar para PostgreSQL es 5432).
- Servicio: parámetros para controlar el servicio (depende del sistema operativo).
- BD de Mantenimiento: conexión inicial, contiene adminpack y esquema pgAgent.
- Nombre de Usuario: rol de PostgreSQL para la conexión.
- Contraseña: clave del rol de PostgreSQL para la conexión.
- Almacenar Contraseña: solicita si se graba la contraseña en un archivo de texto para recordarla.
- Color: solicita si se desea un color que marque los objetos de ese servidor en el explorador de objetos.
- Grupo: grupo del usuario.
- SSL: modo de encriptación de la conexión (requiere, prefiere, permitir, desactivar, verify-ca, verify-full).

-Crear base de datos:

Para poder agregar tablas es necesario crear primero una base de datos haciendo clic con el botón derecho sobre 'Bases de datos' en el explorador de objetos, y seleccionar 'Nueva Base de Datos…', o directamente desde el menú editar 'Nuevo Objeto' y 'Nueva Base de Datos…'.



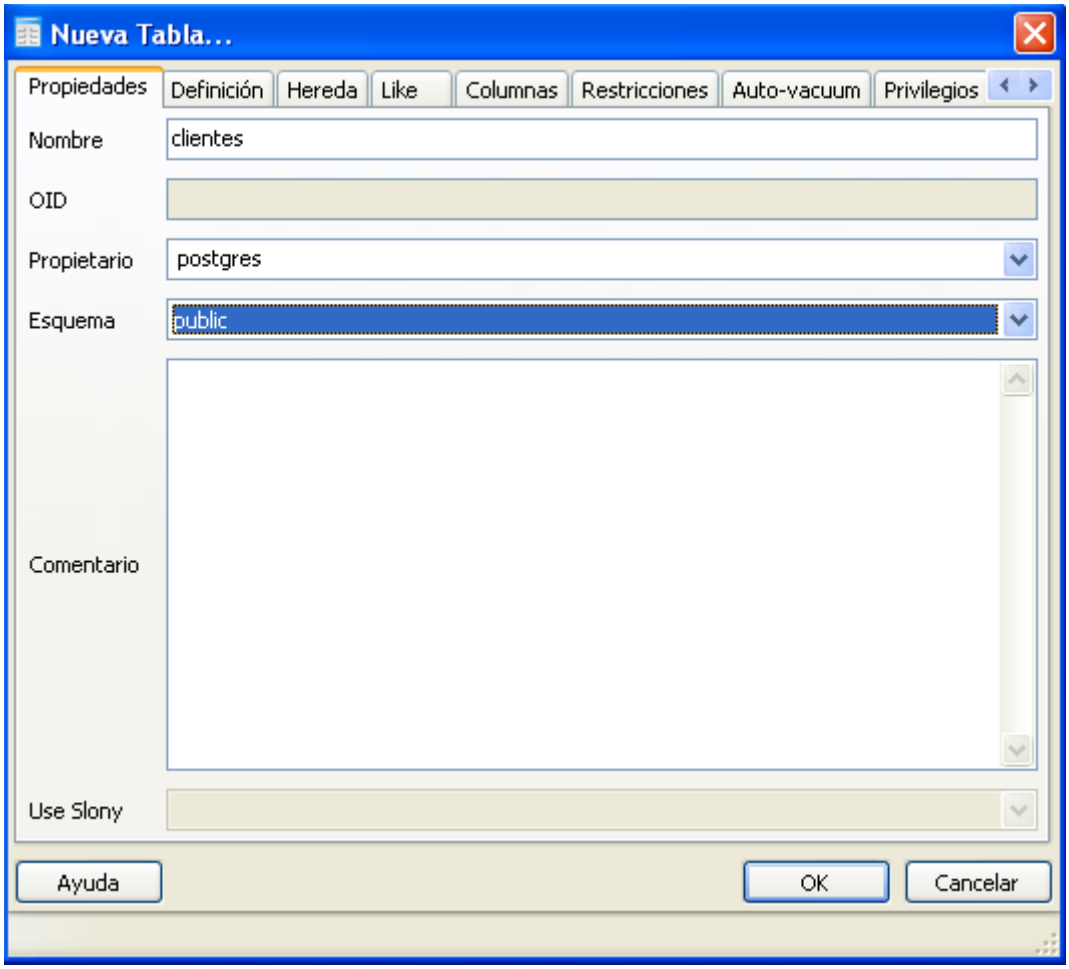


Los campos importantes para rellenar son:

- Nombre: denominación de la base de datos.
- Propietario: usuario (rol) que tendrá derechos especiales sobre la base de datos.
- Codificado: esquema de codificación a utilizar en la base de datos.

-Agregar una tabla:

Para agregar una tabla hay que hacer clic con el botón derecho sobre 'Tablas' en el esquema dentro del explorador de objetos, y seleccionar 'Nueva Tabla…', o directamente desde el menú editar 'Nuevo Objeto' y 'Nueva Tabla…'.

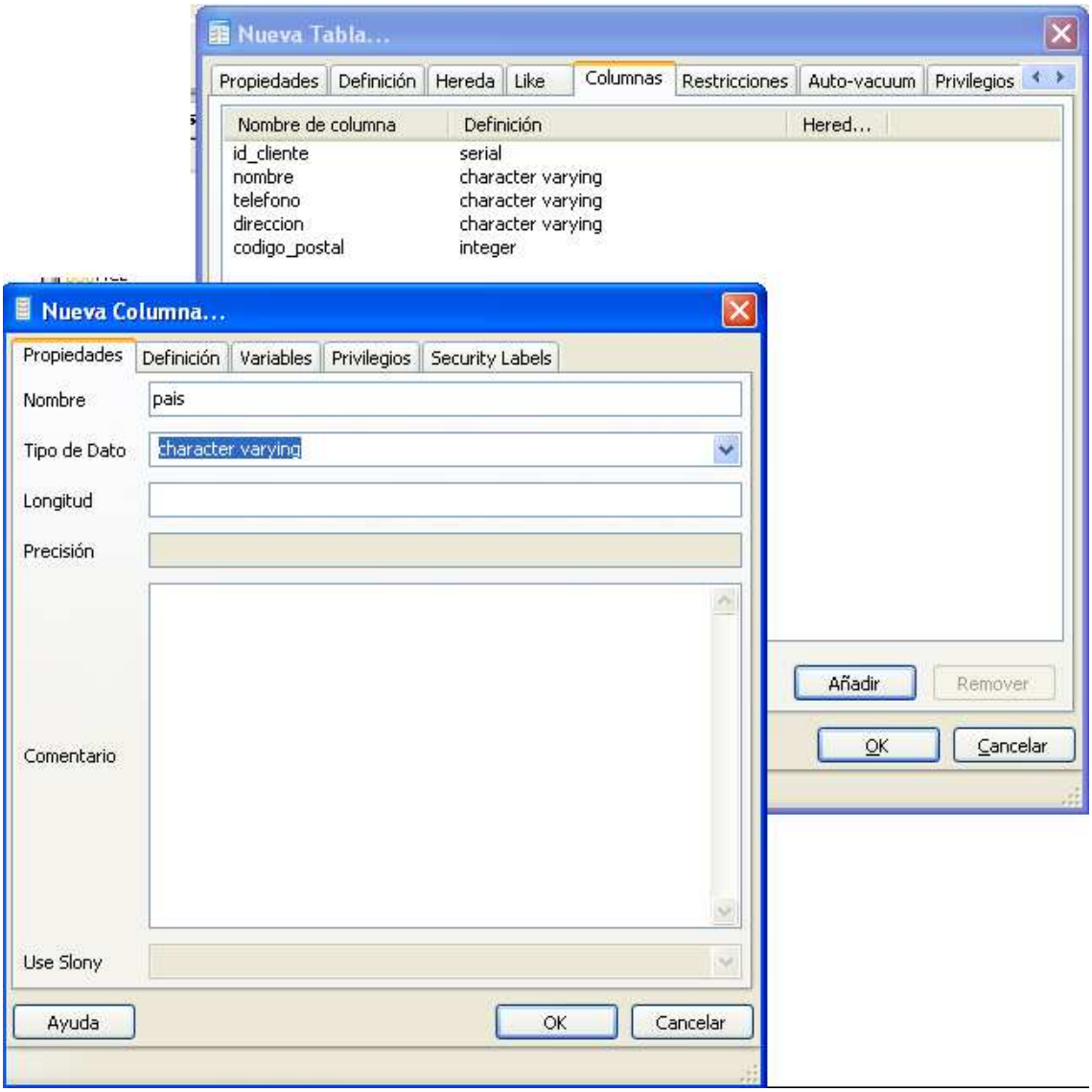


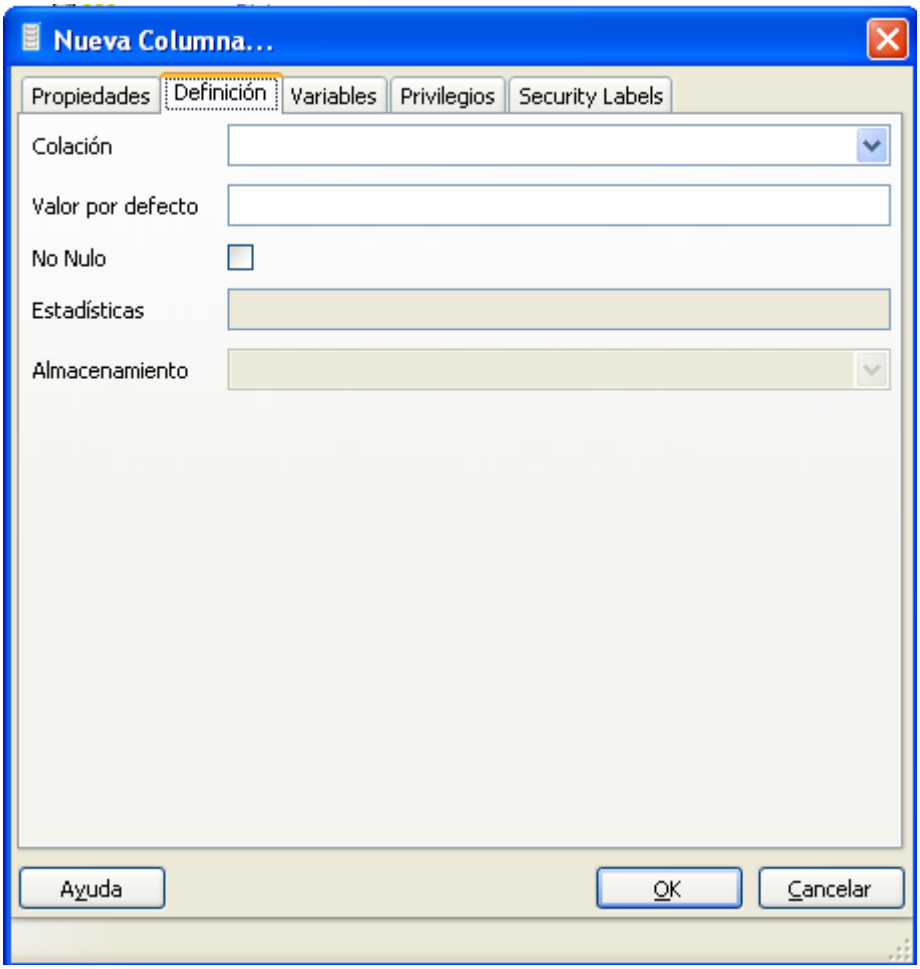
Los campos importantes para rellenar son:

- Nombre: denominación de la tabla.
- Propietario: usuario (rol) que tendrá derechos especiales sobre la tabla.

En la pestaña 'Columnas', para cada columna hay que seleccionar el botón 'Añadir'. Los campos importantes para rellenar son:

- Nombre: denominación de la columna.
- Tipo de datos: tipo de dato de la columna.
- Longitud: para los tipos de datos de longitud variable, la cantidad de dígitos o caracteres totales.
- Precisión: para los tipos de datos numéricos de precisión fija, la cantidad de dígitos decimales.
- Valor por defecto: dato predeterminado que se usará si al ingresar un registro no se especifica ningún valor para la columna.
- No nulo: señala si la columna no puede tomar el valor NULL.

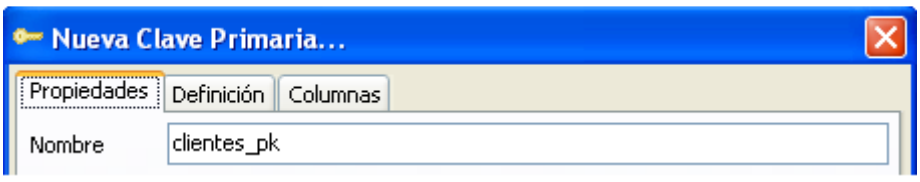

Nueva Tabla...
Propiedades
Definición
Hereda
Like
Columnas
Restricciones
Auto-vacuum
Privilegios
Nombre de columna
Definición
Hered...
id_cliente
serial
nombre
character varying
telefono
character varying
direccion
character varying
codigo_postal
integer
Añadir
Remover
OK
Cancelar
Nueva Columna...
Propiedades
Definición
Variables
Privilegios
Security Labels
Nombre
pais
Tipo de Dato
character varying
Longitud
Precisión
Comentario
Use Slony
Ayuda
OK
Cancelar

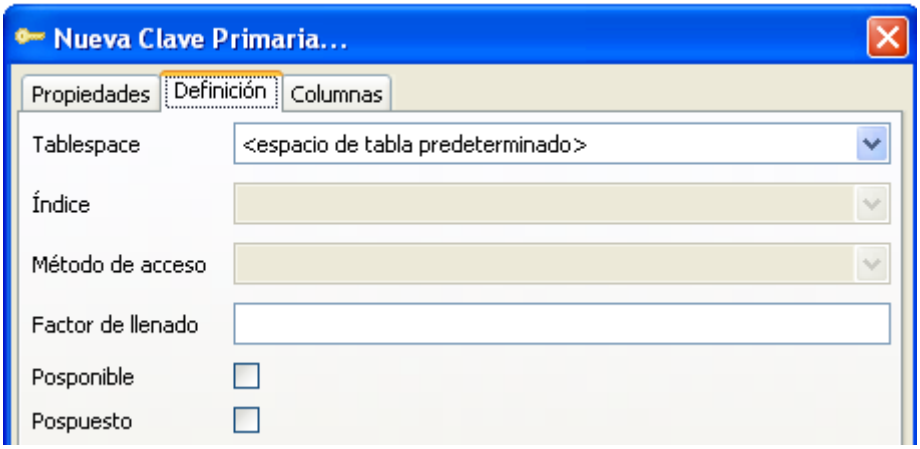


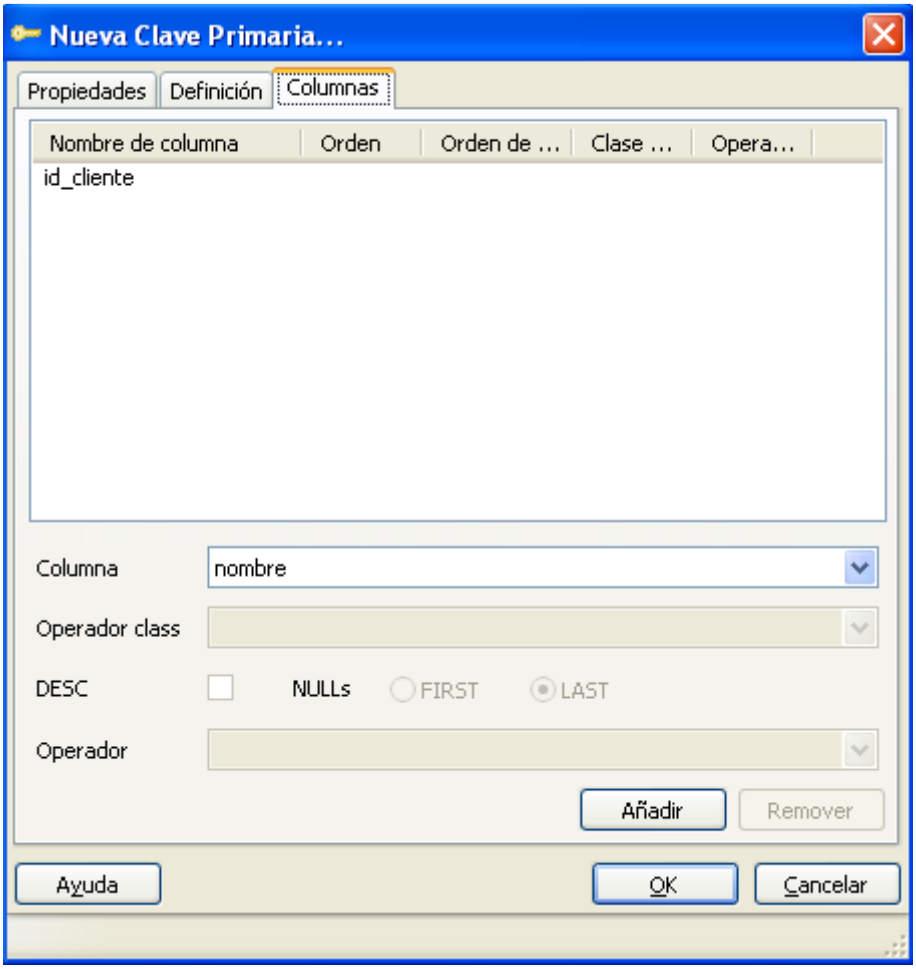
Después de agregar las columnas se pueden agregar las restricciones, desde la pestaña "Restricciones". Para cada restricción hay que seleccionar el tipo de restricción y después el botón 'Añadir'.

Si se agrega una clave primaria, los campos importantes a rellenar son:

- Nombre: denominación de la restricción.
- Columnas: seleccionar la columna o columnas que forman la clave primaria.

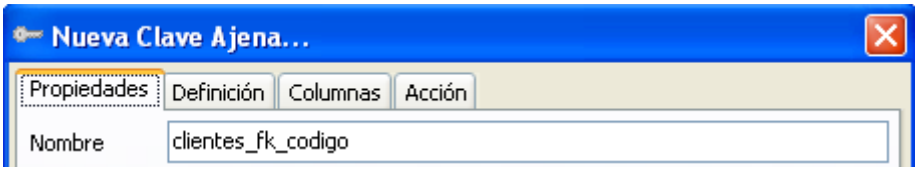


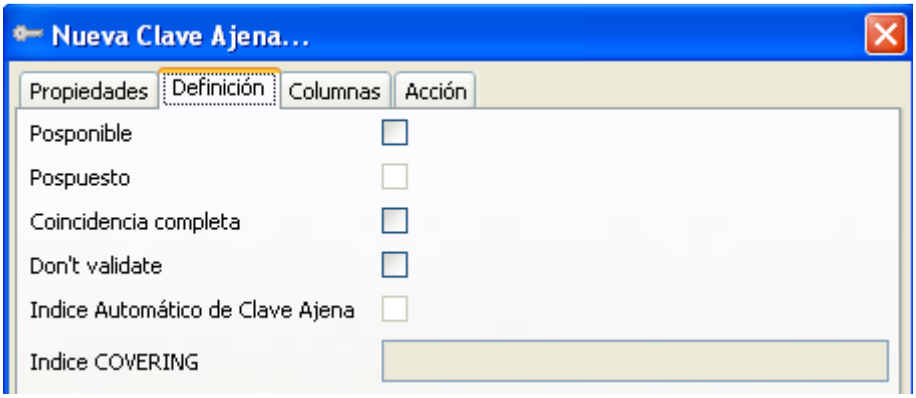


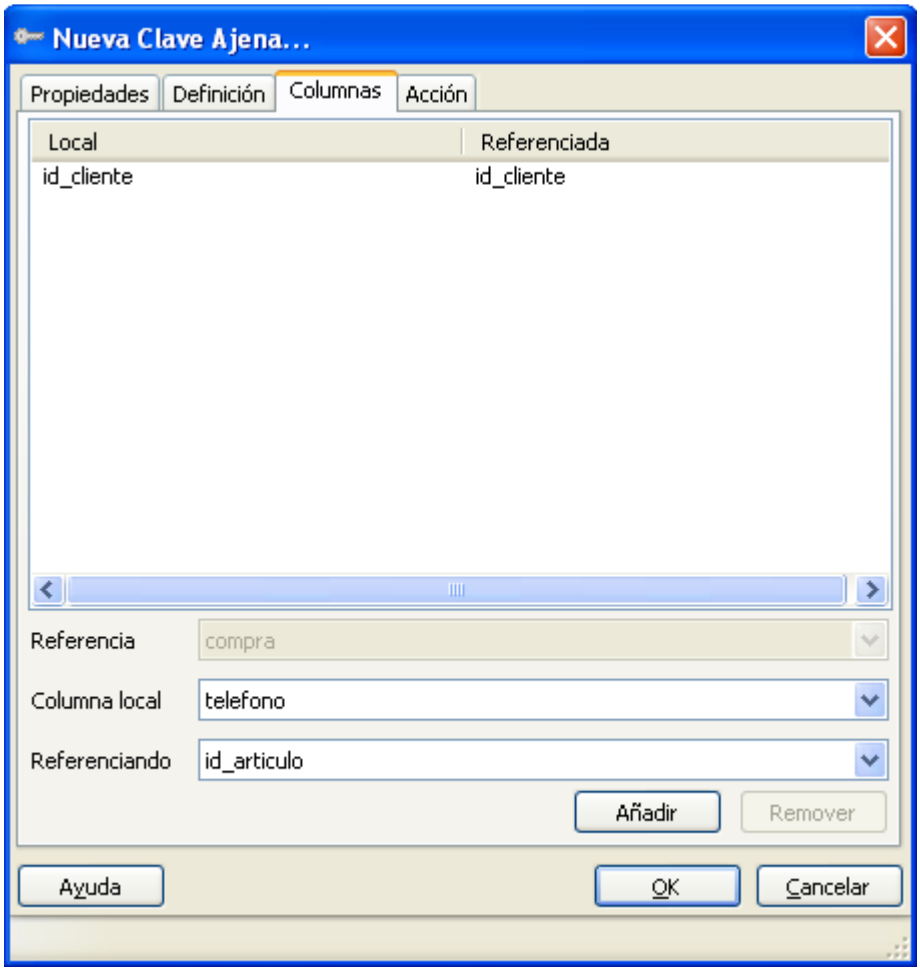


Si se agrega una clave ajena, los campos importantes a rellenar son:

- Nombre: denominación de la restricción.
- Columnas: seleccionar la columna o columnas locales que forman parte de la clave primaria, la tabla ajena a la que hace referencia, y las columnas de la tabla ajena con las que se relacionan las columnas de la tabla local.



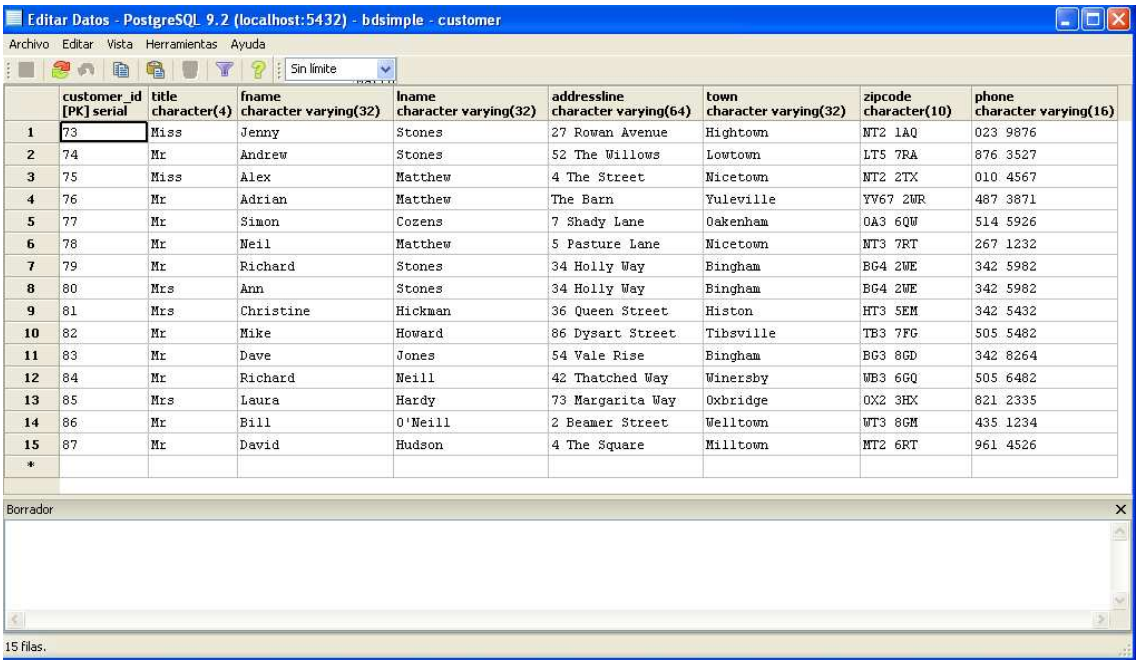




-Herramienta de edición de datos:

El grid de edición permite ver y editar los datos seleccionados en una tabla o vista. Para ello, seleccionar la tabla o vista en el explorador de objetos, y luego presionar el botón 'Ver los

datos del objeto seleccionado' (icono de rejilla o grid) de la barra de herramientas, o directamente por el menú 'Herramientas' 'ver Datos'.



	customer_id [PK] serial	title character(4)	fname character varying(32)	lname character varying(32)	addressline character varying(64)	town character varying(32)	zipcode character(10)	phone character varying(16)
1	73	Miss	Jenny	Stones	27 Rowan Avenue	Hightown	NT2 1AQ	023 9876
2	74	Mr	Andrew	Stones	52 The Willows	Lowtown	LT5 7RA	876 3527
3	75	Miss	Alex	Matthew	4 The Street	Nicetown	NT2 2TX	010 4567
4	76	Mr	Adrian	Matthew	The Barn	Yuleville	YV67 2WR	487 3871
5	77	Mr	Simon	Cozens	7 Shady Lane	Oakenham	OA3 6QW	514 5926
6	78	Mr	Neil	Matthew	5 Pasture Lane	Nicetown	NT3 7RT	267 1232
7	79	Mr	Richard	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982
8	80	Mrs	Ann	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982
9	81	Mrs	Christine	Hickman	36 Queen Street	Histon	HT3 5EM	342 5432
10	82	Mr	Mike	Howard	86 Dysart Street	Tibsville	TB3 7FG	505 5482
11	83	Mr	Dave	Jones	54 Vale Rise	Bingham	BG3 8GD	342 8264
12	84	Mr	Richard	Neill	42 Thatched Way	Winersby	WB3 6GQ	505 6482
13	85	Mrs	Laura	Hardy	73 Margarita Way	Oxbridge	OX2 3HX	821 2335
14	86	Mr	Bill	O'Neill	2 Beamer Street	Welltown	WT3 8GM	435 1234
15	87	Mr	David	Hudson	4 The Square	Milltown	MT2 6RT	961 4526
*								

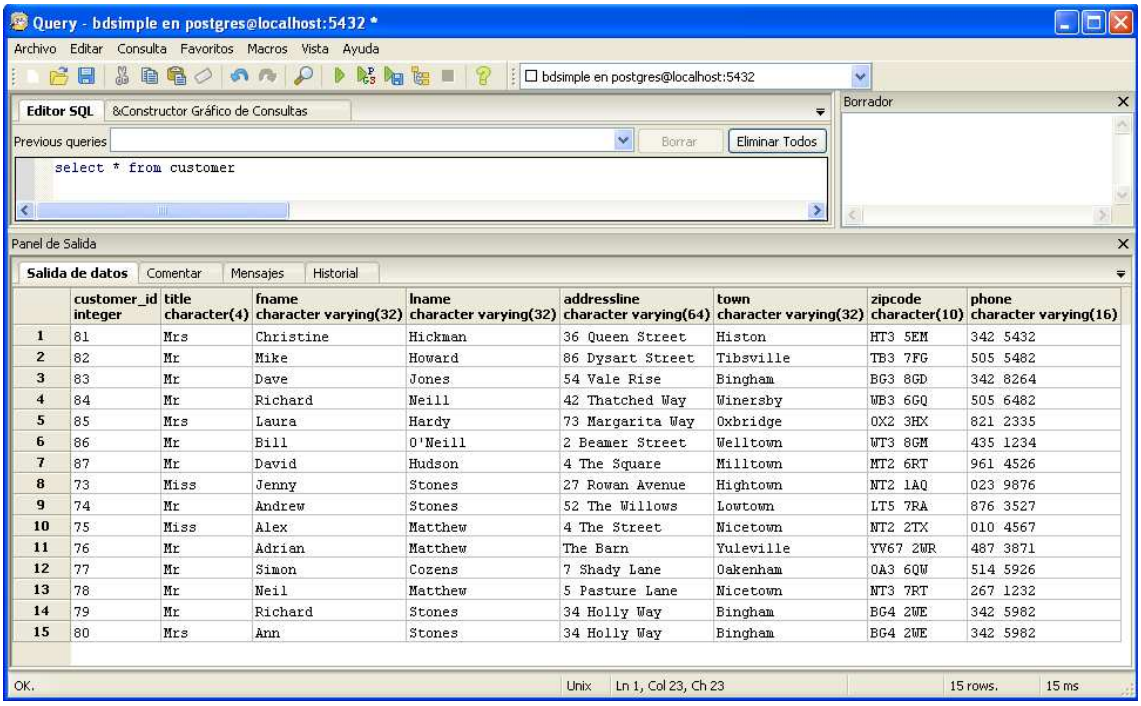
Para editar hay que hacer doble clic en la celda a modificar. Los botones principales en la barra de herramientas son:

- Guardar: almacena los datos modificados.
- Eliminar: borra la fila seleccionada.
- Refrescar: actualiza el grid con los datos más recientes.

Desde el grid sólo se puede actualizar datos (modificar y borrar) si la tabla cuenta con una clave primaria.

-Herramienta de consulta SQL:

Al seleccionar una base de datos, se habilita la herramienta de consulta que permite ejecutar consultas SQL arbitrarias. Hay que hacer clic sobre el botón 'Ejecutar consultas SQL arbitrarias' (icono con un folio con las siglas SQL y un lápiz) de la barra de herramientas, o directamente desde el menú 'Herramientas' 'Herramientas de consulta'.



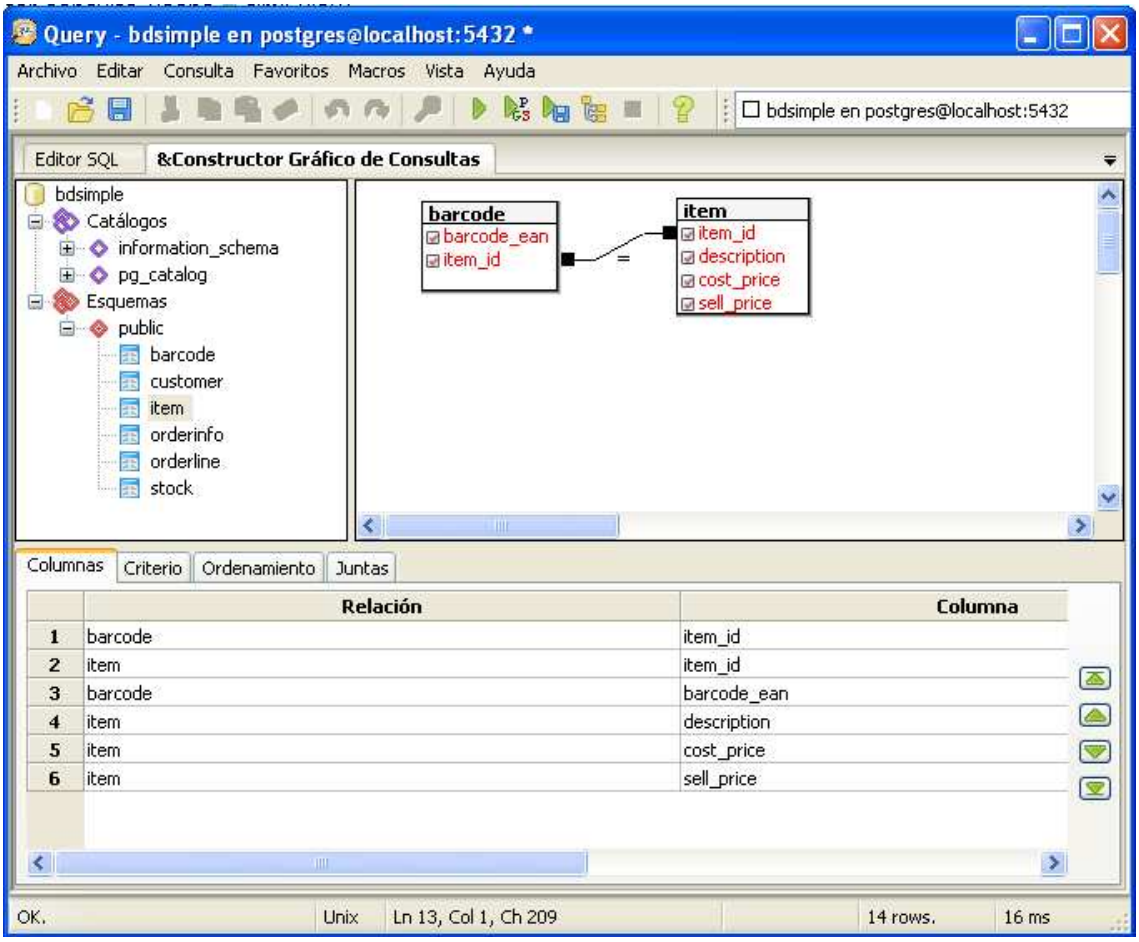
	customer_id integer	title character(4)	fname character varying(32)	lname character varying(32)	addressline character varying(64)	town character varying(32)	zipcode character(10)	phone character varying(16)
1	81	Mrs	Christine	Hickman	36 Queen Street	Histon	HT3 5EM	342 5432
2	82	Mr	Mike	Howard	86 Dysart Street	Tibsville	TB3 7FG	505 5482
3	83	Mr	Dave	Jones	54 Vale Rise	Bingham	BG3 8GD	342 8264
4	84	Mr	Richard	Neill	42 Thatched Way	Winersby	WB3 6GQ	505 6482
5	85	Mrs	Laura	Hardy	73 Margarita Way	Oxbridge	OX2 3HX	821 2335
6	86	Mr	Bill	O'Neill	2 Beamer Street	Welltown	WT3 8GM	435 1234
7	87	Mr	David	Hudson	4 The Square	Milltown	MT2 6RT	961 4526
8	73	Miss	Jenny	Stones	27 Rowan Avenue	Hightown	NT2 1AQ	023 9876
9	74	Mr	Andrew	Stones	52 The Willows	Lowtown	LT5 7RA	876 3527
10	75	Miss	Alex	Matthew	4 The Street	Nicetown	NT2 2TX	010 4567
11	76	Mr	Adrian	Matthew	The Barn	Yuleville	YV67 2WR	487 3871
12	77	Mr	Simon	Cozens	7 Shady Lane	Oakenham	OA3 6QW	514 5926
13	78	Mr	Neil	Matthew	5 Pasture Lane	Nicetown	NT3 7RT	267 1232
14	79	Mr	Richard	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982
15	80	Mrs	Ann	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982

Los botones principales en la barra de herramientas son:

- Ejecutar consulta: ejecuta la consulta SQL introducida.
- Exportar datos consulta: guarda las filas resultantes
- Analizar consulta: muestra el gráfico EXPLAIN con una explicación visual de los nodos del plan de ejecución de la consulta.

-Constructor gráfico de consultas:

PgAdmin ofrece una herramienta de consulta para diseñar consultas virtualmente.

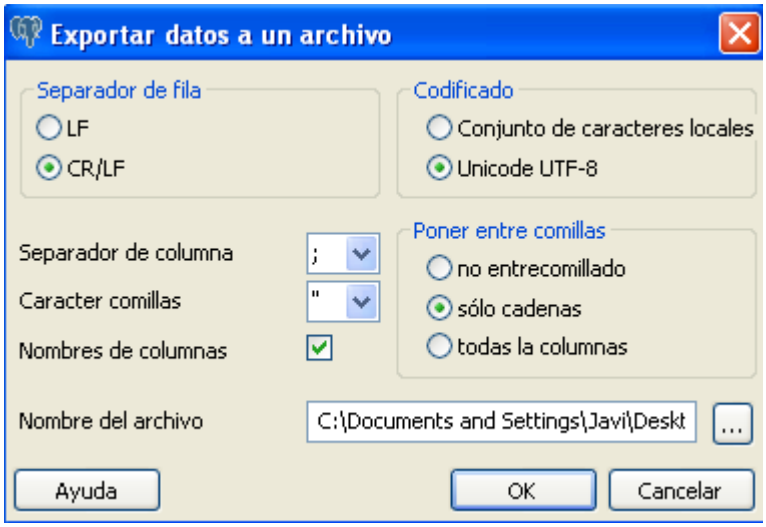


Se compone de las siguientes partes:

- En el panel izquierdo se encuentra la base de datos, con sus esquemas, y cada una con sus tablas.
- En el panel derecho se encuentra la representación gráfica de las columnas.
- En el panel inferior se encuentran las pestañas 'Columnas' (para detallar las columnas que mostrará la SELECT), 'Criterio' (para incluir condiciones WHERE), 'Ordenamiento' (para especificar el orden de salida ORDER), y 'Juntas' (para especificar los tipos de JOIN).

-Herramienta de exportación:

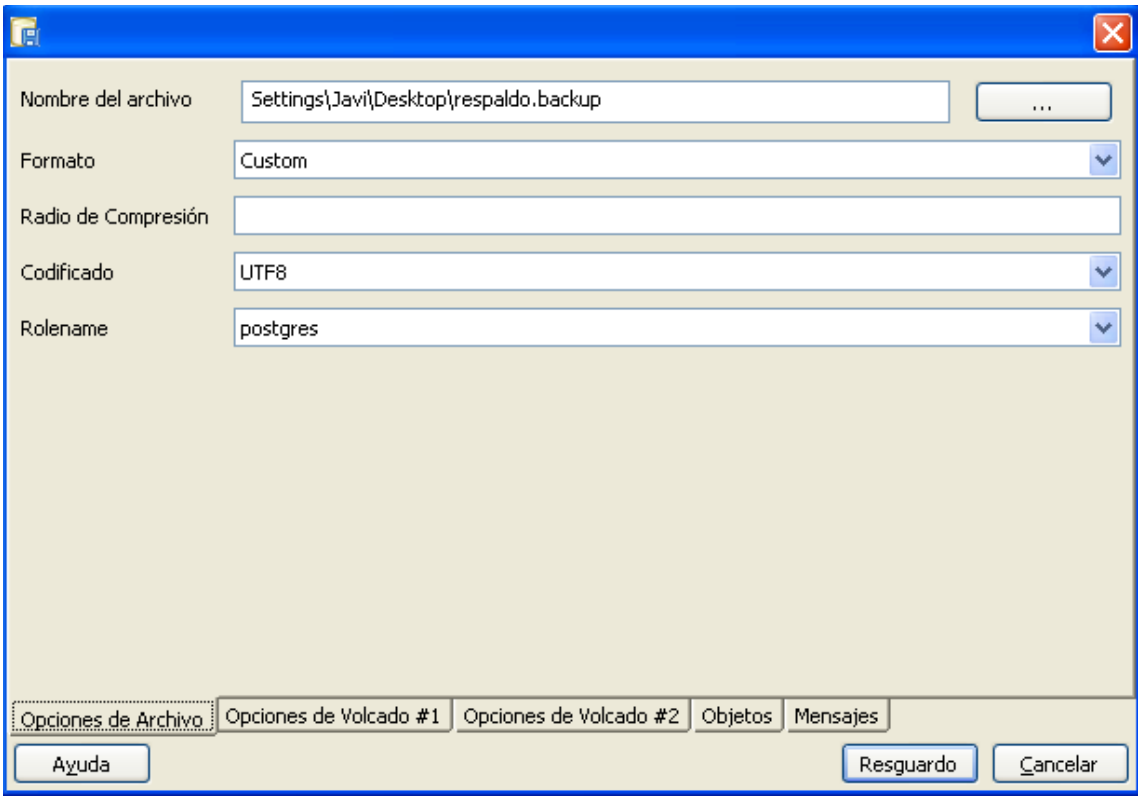
La herramienta de exportación permite exportar datos de la herramienta de consulta. Hay que hacer clic sobre el botón 'Ejecute consulta, escriba resultado a un archivo' (icono de play con un disquete) de la barra de herramientas, o mediante el menú 'Consulta' 'Ejecutar a un archivo':

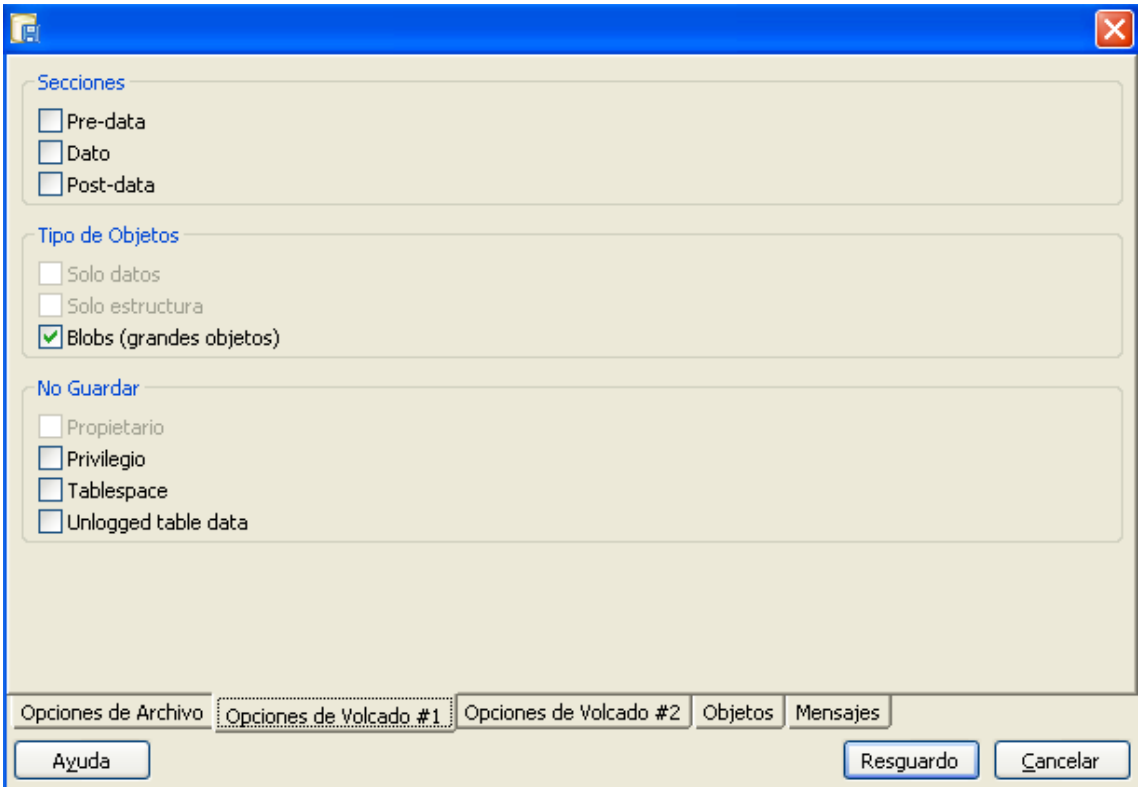


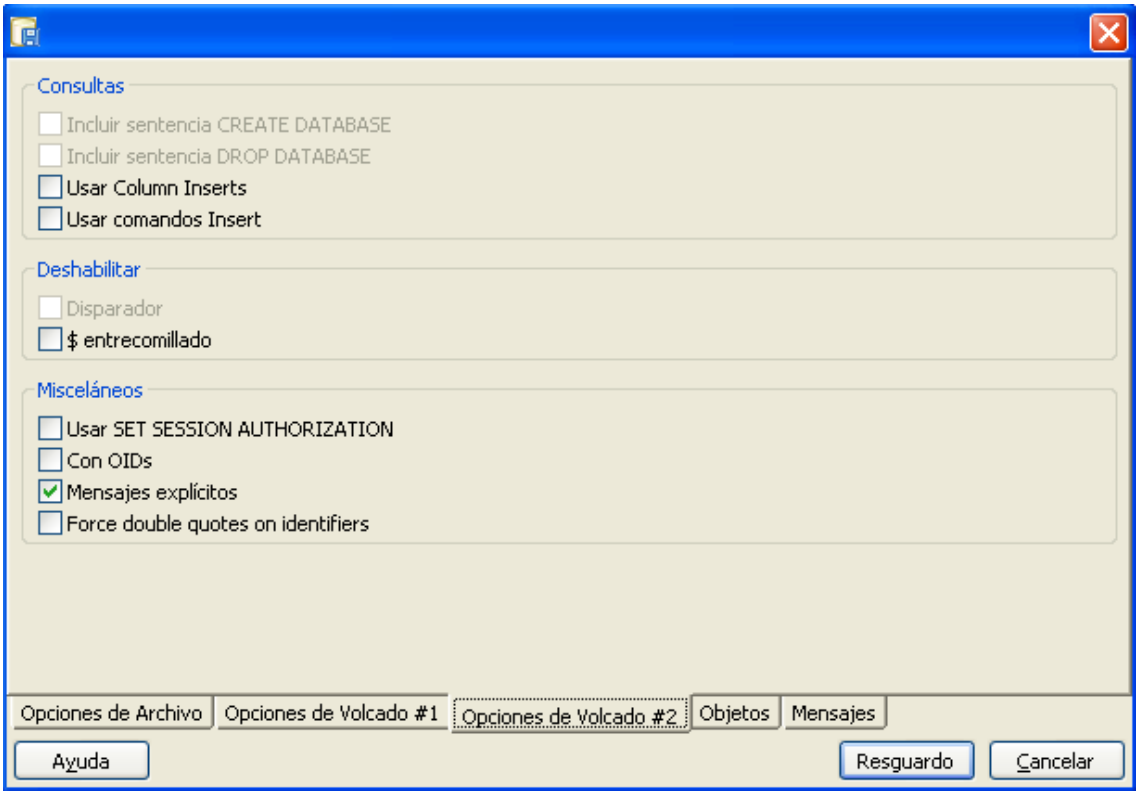
HERRAMIENTAS DE RESGUARDO Y RESTAURACIÓN

-Herramienta de resguardo:

La herramienta de resguardo o backup llama a la herramienta de volcado de PostgreSQL para crear copias de seguridad de los datos. Para utilizarla hay que seleccionar la base de datos en el explorador de objetos, hacer clic con el botón derecho y seleccionar 'Resguardo…'.







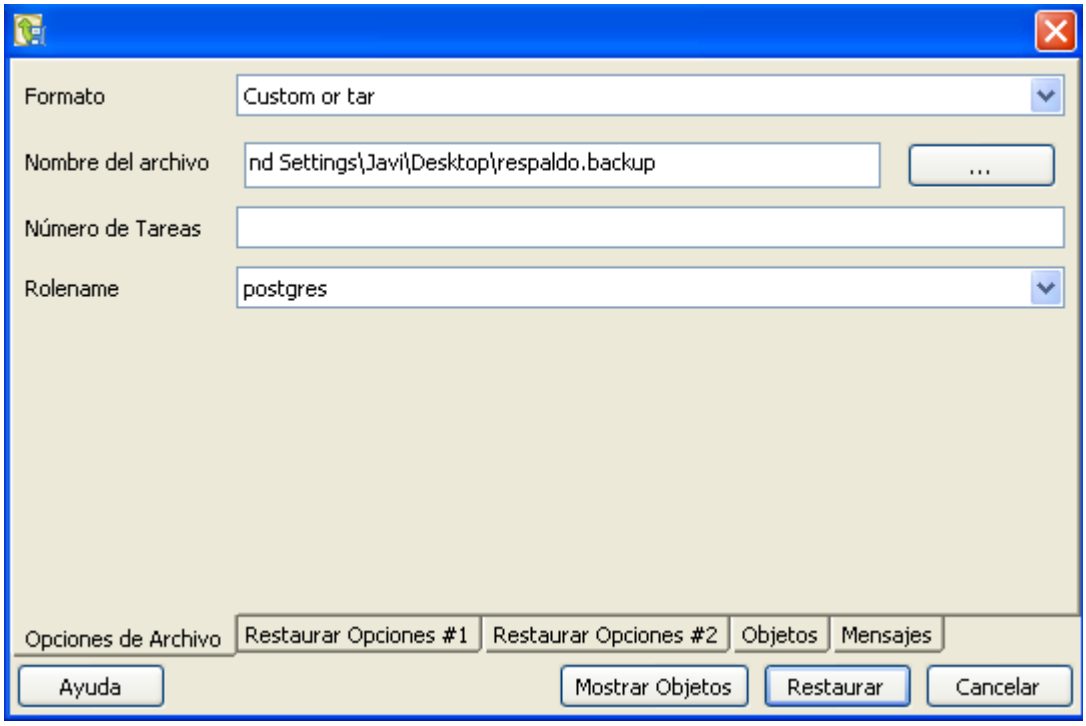
Los campos principales a rellenar son:

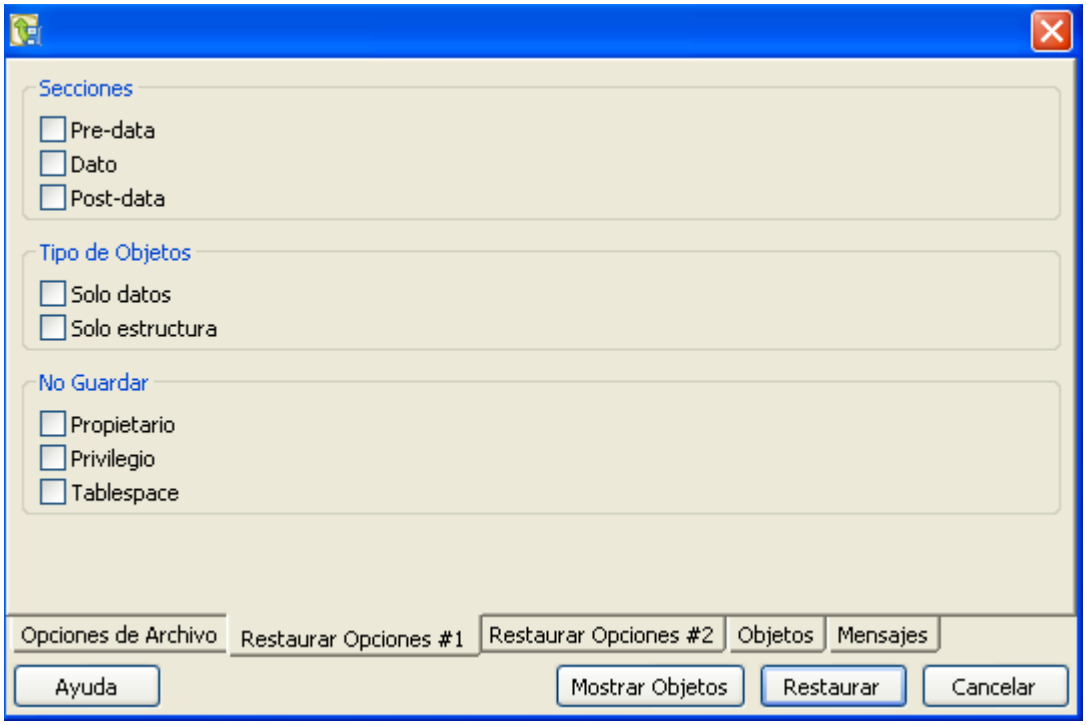
- Nombre del archivo: denominación del archivo de respaldo en el sistema operativo.
- Formato: elegir formato con el que se comprimirá los datos.

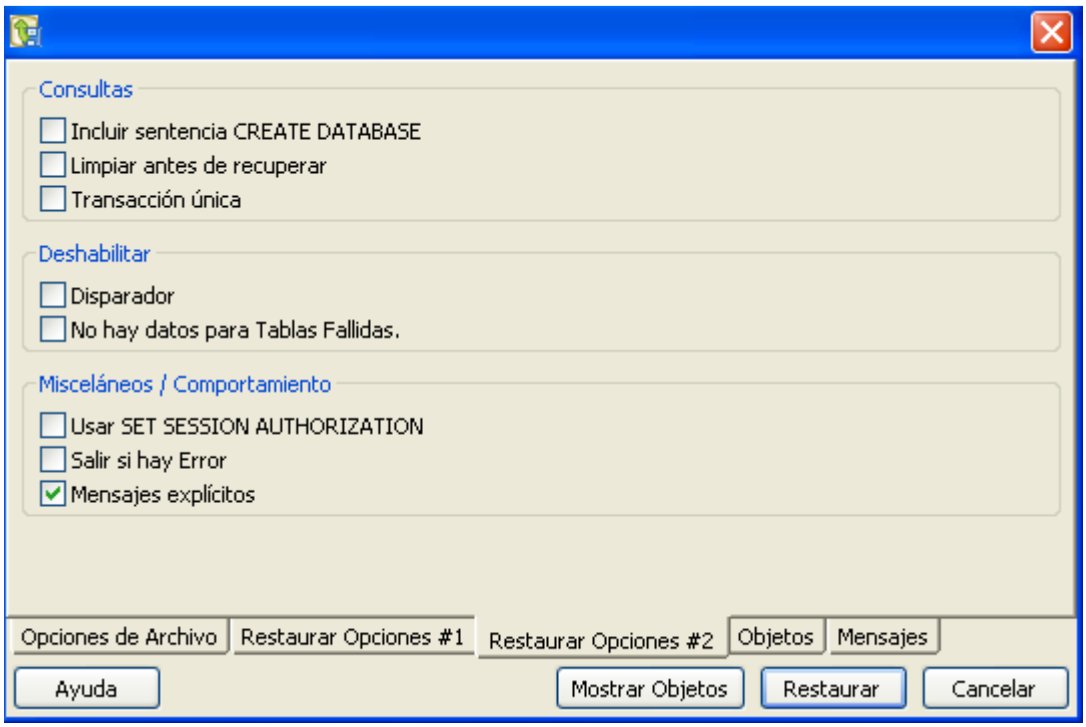
- Tipo de objetos: elegir el tipo de objetos que se respaldan (entre otros los archivos grandes Blobs).
- Consultas: forma en que se almacenarán los datos respaldados (por ejemplo con comandos INSERT).

-Herramienta de restauración:

La herramienta de restauración (restore) de PostgreSQL restaura los datos desde copias de seguridad (archivos de backup). Para utilizarla hay que crear una base de datos en blanco, seleccionarla en el explorador de objetos, clic sobre ella con el botón derecho y elegir 'Restaurar…', o desde el menú 'Herramientas' 'Restaurar'.







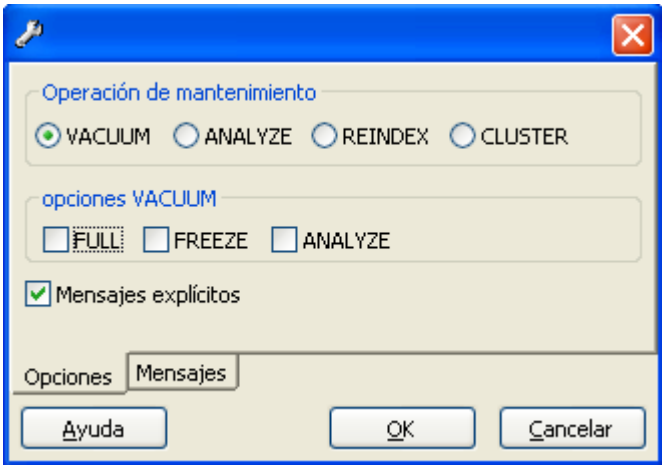
Los campos más importantes a rellenar son:

- Nombre de archivo: ubicación del archivo backup.
- Tipo de objetos: seleccionar si se quiere restaurar únicamente la estructura o los datos.

Al hacer clic sobre el botón 'Restaurar', PostgreSQL leerá el archivo backup restaurando los datos. En caso de éxito, el botón cambia a 'Hecho' para finalizar.

HERRAMIENTA DE MANTENIMIENTO

Esta herramienta ejecuta la tarea de reconstruir las estadísticas sobre la base de datos y tablas, limpiar los datos no utilizados y reorganizar los índices. Para utilizarla hay que hacer clic con el botón derecho en el explorador de objetos, y seleccionar 'Mantenimiento…', o desde el menú 'Herramientas' 'Mantenimiento'.



En opciones de mantenimiento se puede seleccionar:

- VACUMM: para limpiar las filas inválidas.
- ANALYZE: para analizar los datos y recalcular estadísticas.
- REINDEX: para reorganizar los índices.
- CLUSTER: para seleccionar todo el cluster.

En opciones de VACUUM se puede seleccionar:

- FULL: para compactar la tabla.
- FREEZE: para utilizar esta técnica de "enfriamiento" de filas.
- ANALYZE: para analizar los datos y recalcular estadísticas.

BIBLIOGRAFÍA

- [2] "Beggining Databases with PostgreSQL: From Novice to Professional, Second Edition"
- [13] http://www.arpug.com.ar/trac/wiki/PgAdmin

BIBLIOGRAFÍA

-[1] "PostgreSQL: Introduction and Concepts"

Bruce Momjian
Addison-Wesley
2001

-[2] "Beggining Databases with PostgreSQL: From Novice to Professional, Second Edition"

Neil Matthew y Richard Stones
Appress
2005

-[3] http://www.postgresql.org

Página oficial del proyecto PostgreSQL

-[4] http://www.rae.es

Página oficial de la Real Academia Española

-[5] http://wikipedia.org

Enciclopedia libre

-[6] "PostgreSQL 9.0 High Performance"

Gregory Smith
Packt Publishing
2010

-[7] "PostgreSQL 9 Administration Cookbook"

Simon Riggs, Hannu Krosing
Packt Publishing
2010

-[8] http://es.scribd.com/doc/81543140/cuadro-comparativo

Cuadro comparativo de sistemas de bases de datos
Manuel Miranda Buenabad

-[9] http://www.nexusdb.com

Página oficial de la base de datos NexusDB

-[10] http://www.microsoft.com
Página oficial de microsoft

-[11] http://www.mysql.com
Página oficial de la base de datos MySQL

-[12] http://www.computerworld.com.au/
Página de la revista informática Computer World

-[13] http://www.arpug.com.ar/
Página de la comunidad argentina de PostgreSQL