Universidad de las Ciencias Informáticas

Facultad 3

“Guía para la optimización de servidores de bases de datos de PostgreSQL”

Trabajo de Diploma para optar por el título de Ingeniero en Ciencias Informáticas.

Autores:

Mariluz Hernández Perdomo Enrique José González Fernández

Tutora:

Ing. Eidy Marien Carbó Peña

Año 50 del Triunfo de la Revolución

2008-2009

Declaración de Autoría

Declaramos ser autores de la presente tesis y reconocemos a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste firmamos la presente a los _ días del mes de junio del año 2009.

Mariluz Hernández Perdomo

Enrique José González Fernández

_____

Firma del Autor(a)

_____

Firma del Autor(a)

Ing. Eidy Marien Carbó Peña

_____

Firma del Tutor(a)

Dedicatoria

A mis padres… porque les debo todo lo que soy.

A mis abuelos Miriam y Antonio.

A Enrique, por ayudarme y quererme tanto.

Mariluz.

A mis padres y mi hermana quienes son todos para mí.

A Mariluz, sin ti no lo hubiera logrado.

A Lizmet que me ha ayudado siempre en todo lo que ha podido.

A Mercedes y Jorge Luis por quererme como a un hijo.

Enrique.

Agradecimientos

De Mariluz:

A mi madre, por su amor infinito y por estar a mi lado en todo momento.

A mi padre, porque en dondequiera que esté ahora sé que estará orgulloso de mí.

A Marien, por ser una excelente tutora y por preocuparse de que todo saliera bien.

A mis abuelos Miriam y Antonio, por quererme tanto y siempre estar pendientes de mí.

A Riquillo, porque me enseñaste a no llorar por cosas que no lo merecen, porque me has hecho crecer y has sido alguien en quien he podido confiar plenamente mis cosas.

A mis amores Thais, Mary y Ali, por tender sus manos cuando más las necesité y por todos los buenos momentos que pasamos juntas.

A Laura y Enrique, por preocuparse siempre por nosotros.

A Lizmet, por su ayuda y sus buenos consejos.

A mi familia, en particular a Mary, Maribel, Annette, Danay y Cecilia, por ayudar a mis padres en difíciles momentos y permitir que no se afectara mi carrera.

A Gladys, Susi, Pipo y a Yoan, por ser como un hijo para mi mamá y darle el cariño que le falta cuando no estoy a su lado.

A Nahuel (y su socio) por ser una maravillosa persona.

A Josi, por ayudarme cuando lo he necesitado y por todas las miles de carcajadas que me ha provocado.

A Frank y la flaca por ser muy buenos con nosotros, a Yani, Kenia, Tico, Yamisel, Darián.

A todos mis amigos que en algún momento me han ayudado y que me han dado muchos ratos felices.

De Enrique:

A mis padres por enseñarme tanto y por ayudarme siempre en todo.

A Mariluz por todo el tiempo que pasó rectificando mis redacciones.

A Marien quien se tomó esta tesis tan en serio como nosotros.

A Jose y a Thais por estar siempre que nos hicieron falta.

Al Gordo y la Flaca por sus comidas tan ricas.

A todos aquellos que nos ayudaron de una forma u otra y que son muchos para nombrarlos uno a uno.

Resumen

En el presente trabajo los autores proponen una guía de pasos para realizar la optimización de servidores de bases de datos en PostgreSQL, a partir las conclusiones obtenidas luego de analizar la situación actual respecto a esto y la necesidad de mejorar esta actividad. Luego se realizó una investigación o estudio del estado del arte de las formas de optimización en la actualidad realizada por los gestores de bases de datos más conocidos. Esta guía está formada por un grupo de pasos, los que no solo se limitan al marco teórico, sino que además cuentan con consultas SQL que le confieren al administrador de la base de datos una visión práctica de la actividad del servidor y de este modo pueda conocer qué camino seguir a la hora de afinar el servidor. Se muestran los resultados obtenidos luego de aplicar la investigación en el Centro de Tecnologías de Almacenamiento y Análisis de Datos, los cuales fueron satisfactorios para las pruebas realizadas. Finalmente los autores realizan algunas recomendaciones a tener en cuenta por los autores cuando empleen la guía y en caso de dar continuidad a esta investigación.

Índice de contenido

Índice de Tablas

Índice de Figuras

Introducción

En la actualidad existen numerosas aplicaciones o sistemas que se encargan de automatizar las actividades del hombre en las diferentes esferas de la vida. Estas pueden ser software de gestión económica, empresarial, salud, educación, entre otras que tengan la necesidad de hacer persistir sus datos a lo largo del tiempo, para consultar dicha información con múltiples objetivos, para obtener reportes, como apoyo a otros sistemas, para la toma de decisiones o por el simple hecho de almacenar datos históricos. Esto es posible gracias a las tecnologías de almacenamiento existentes en las ciencias de la computación, que van desde los simples ficheros cuando no es necesario almacenar grandes volúmenes de datos, o los conocidos XML, hasta las más extendidas y potentes bases de datos.

Las bases de datos comienzan a surgir en la década de los 50. En sus inicios existía una asociación estática de los ficheros a los programas y contaban con problemas como la redundancia, inconsistencia de los datos y dependencia entre ellos, excesivo mantenimiento, poca flexibilidad ante los cambios, baja productividad, limitación de recursos compartidos y medidas de seguridad difíciles, entre otros. Estos problemas se han erradicado en gran medida gracias al empeño que le han dedicado múltiples empresas y desarrolladores a lo largo de varios años, evidenciándose en la robustez cada vez mayor de las aplicaciones.

Como resultado de la evolución de las bases de datos se tiene que hoy en día estas sean mucho más robustas, seguras, flexibles y manejables en gran medida en la mayor parte de los casos. Con el objetivo de introducir información en las bases de datos, controlarlas y accederlas surgieron los sistemas gestores de bases de datos. Estos facilitan la administración o gestión de los datos a los administradores cuyas tareas principales son:

- Administrar la estructura de la base de datos.
- Administrar la actividad de los datos.
- Establecer el diccionario de datos.

- Asegurar la confiabilidad de los datos.
- Garantizar la seguridad de la base de datos.

A través de la fundamental tarea de administrar el sistema gestor de bases de datos es esencial la optimización de estas, ya que tiene gran importancia a la hora de explotar un servidor, puesto que es en ese momento que se encuentra en funcionamiento cuando se puede comprobar si la configuración que tiene es la adecuada en dependencia de la velocidad de respuesta y su efectividad, y a partir de esto se podrá realizar cambios para que mejoren las características del sistema y este cuente con un rendimiento óptimo para los usuarios y la aplicación. Esta actividad la mayoría de las veces es realizada por personas jóvenes e inexpertas, los que no cuentan con una guía adecuada para realizarla, que además les provea de resultados prácticos de la actividad de la base de datos, adquiriendo un conocimiento útil a la hora de tomar decisiones para mejorar el rendimiento del servidor. Durante la labor de la administración de la base de datos, estos se enfrentan con problemas como:

- En los diseños de bases de datos no se aplican reglas de normalización, lo que conlleva a tener redundancia de la información. Otras veces hacen excesivo el uso de estas reglas y esto trae consigo la desnormalización del modelo de datos.
- Mantienen los servidores de bases de datos con la configuración por defecto, lo que provoca en ocasiones un bajo rendimiento de los mismos, debido a que esta configuración predeterminada es para que el programa funcione sobre casi cualquier ordenador con los requerimientos mínimos.
- No hacen uso de tareas de administración como las copias de seguridad, los procedimientos de limpieza de datos, entre otras.
- Tratamiento inadecuado de los índices.
- Diseño inadecuado e ineficiente de las consultas, lo que ocurre muchas veces por el desconocimiento de la arquitectura del Sistema de Gestión de Bases de Datos, o de las técnicas de programación empleadas en dichos casos.

Dado la situación anterior, se deduce que si los administradores contaran con una guía para llevar a cabo la optimización de los servidores en PostgreSQL, se mejoraría notablemente el rendimiento en estos y la rapidez de respuesta de los mismos, además se sentirán más cómodos y confiados al realizar esta tarea tan importante para con el rendimiento de los servidores de bases de datos.

A partir de lo anteriormente planteado surge el siguiente **problema científico de la investigación:**

- Las insuficiencias de las guías existentes para la optimización de servidores de bases de datos de PostgreSQL está afectando la toma de decisiones para mejorar el rendimiento en los servidores de bases de datos de PostgreSQL y la rapidez de respuesta de los mismos.

El **objeto de estudio** de la investigación es: las tecnologías de bases de datos y el **campo de acción**: la optimización de servidores de bases de datos.

El **objetivo general** de la investigación que se plantea para dar solución al anterior problema es:

- Obtener una guía para la optimización de servidores de bases de datos de PostgreSQL.

Este objetivo se desglosa en los siguientes **objetivos específicos**:

1. Realizar la fundamentación teórica de la investigación.
2. Confeccionar una guía para la optimización de servidores de bases de datos de PostgreSQL con enfoque en el diseño, la programación, y la configuración de servidores de bases de datos.
3. Validar la propuesta aplicándola como parte de los servicios que ofrece el Centro de Tecnologías de Almacenamiento y Análisis de Datos (CENTALAD).

La **hipótesis** planteada para esta investigación es:

- Con el desarrollo de una guía para la optimización de servidores de bases de datos de PostgreSQL se facilitará la toma de decisiones para mejorar el rendimiento y la rapidez de respuesta de los mismos.

Para dar cumplimiento a los objetivos se plantean las siguientes **tareas generales** de la investigación:

1. Estudiar el arte de las formas de afinamiento de servidores de bases de datos en Oracle, MySQL y PostgreSQL.
2. Estudiar la arquitectura de PostgreSQL.
3. Identificar las áreas o parámetros a optimizar a través de cada uno de los pasos de la guía.
4. Implementar las consultas para obtener reportes sobre la información en la base de datos.
5. Integrar y probar la solución propuesta en el Centro de Tecnologías de Almacenamiento y Análisis de Datos.

Capítulo 1 Fundamentación Teórica

En el presente capítulo se presenta la fundamentación teórica que se ha llevado a cabo en esta investigación para posteriormente llegar a la solución del problema. Se explican las características de los sistemas gestores de bases de datos así como algunos ejemplos de estos atendiendo el tipo de modelo que emplean. Se enuncian un grupo de conceptos importantes en el área de las bases de datos y que son claves en el desarrollo de este trabajo. Por último se podrá encontrar las tecnologías actuales en este entorno y las tendencias hacia el punto central donde se dirige la investigación.

1.1 Sistemas de Gestión de Bases de Datos

Para introducir el tema de los Sistemas de Gestión de Bases de Datos lo primero es comenzar por algunos conceptos o definiciones manejados sobre las bases de datos. Muchos autores, con los que están de acuerdo los autores de este trabajo, han planteado disímiles definiciones acerca de estas sin embargo todos siguen la misma línea o idea, el autor (Yunta, 2001) por ejemplo, propone que una base de datos es un conjunto de información estructurada en registros y almacenada en un soporte electrónico legible desde un ordenador, donde cada registro constituye una unidad autónoma de información que puede estar a su vez estructurada en diferentes campos o tipos de datos que se recogen en dicha base de datos.

Otros, como (Andrés, 2001), se refieren a estas como un conjunto de datos almacenados entre los que existen relaciones lógicas y que han sido diseñadas para satisfacer los

requerimientos de información de una empresa u organización. En una base de datos, además de los datos, también se almacena su descripción.

El autor Christopher J. Date (Date, 2003) plantea que una Base de Datos es un conjunto de datos que tiene varias propiedades implícitas, entre las que se encuentran:

- Representa algún aspecto del mundo real, llamado *minimundo* o universo de discurso. Las modificaciones del *minimundo* se reflejan en la base de datos.
- Es un conjunto de datos lógicamente coherentes, con un cierto significado inherente. Es necesario aclarar que una colección aleatoria de datos no puede considerarse propiamente una base de datos.
- Una base de datos se diseña, construye y puebla con datos para propósitos específicos. Está dirigida a un grupo de usuarios y tiene ciertas aplicaciones preconcebidas que interesan a distintos usuarios.

Otra de las características de estas es que los datos son usados por varias personas o usuarios así como distintas aplicaciones, de las cuales deben permanecer independientes los datos. Estos son definidos con una descripción única en cada caso.

Para definir, crear y mantener las bases de datos, así como proporcionar y controlar el acceso a estas, son usados los Sistemas de Gestión de Bases de Datos. El autor (Date, 2003) ha definido un Sistema Gestor o Manejador de Bases de Datos (SGBD) como un conjunto de programas que permite a los usuarios crear y mantener una base de datos, por lo tanto el SGBD es un software de propósito general que facilita el proceso de definir, construir y manipular la base de datos para diversas aplicaciones. Pueden ser de propósito general o específico.

La autora (Andrés, 2001) plantea que un SGBD debe ofrecer los siguientes servicios, basándose según lo expresado por Codd, el creador del modelo relacional:

1. Un SGBD debe proporcionar a los usuarios la capacidad de almacenar datos en la base de datos, acceder a ellos y actualizarlos, siendo esta la función fundamental de un SGBD y por supuesto, el SGBD debe ocultar al usuario la estructura física

interna, o lo que es lo mismo, la organización de los ficheros y las estructuras de almacenamiento.

2. Un SGBD debe proporcionar un catálogo en el que se almacenen las descripciones de los datos y que sea accesible por los usuarios. El mencionado catálogo es lo que se denomina diccionario de datos y contiene información que describe los datos de la base de datos (metadatos). Normalmente, un diccionario de datos almacena:

   - Nombre, tipo y tamaño de los datos.
   - Nombre de las relaciones entre los datos.
   - Restricciones de integridad sobre los datos.
   - Nombre de los usuarios autorizados a acceder a la base de datos.
   - Esquema externo, conceptual e interno, y correspondencia entre los esquemas.
   - Estadísticas de utilización, tales como la frecuencia de las transacciones y el número de accesos realizados a los objetos de la base de datos.

   Algunos de los beneficios que reporta el diccionario de datos son los siguientes:

   - La información sobre los datos se puede almacenar de un modo centralizado, lo que ayuda a mantener el control sobre los datos, como un recurso que son.
   - El significado de los datos se puede definir, lo que ayudará a los usuarios a entender el propósito de los mismos.
   - La comunicación se simplifica ya que se almacena el significado exacto. El diccionario de datos también puede identificar al usuario o usuarios que poseen los datos o que los acceden.
   - Las redundancias y las inconsistencias se pueden identificar más fácilmente ya que los datos están centralizados.
   - Se puede tener un historial de los cambios realizados sobre la base de datos.

- El impacto que puede producir un cambio se puede determinar antes de que sea implementado, ya que el diccionario de datos mantiene información sobre cada tipo de dato, todas sus relaciones y todos sus usuarios.
- Se puede hacer respetar la seguridad.
- Se puede garantizar la integridad.
- Se puede proporcionar información para auditorías.

3. Un SGBD debe proporcionar un mecanismo que garantice que todas las actualizaciones correspondientes a una determinada transacción se realicen, o que no se realice ninguna. Una *transacción* es un conjunto de acciones que cambian el contenido de la base de datos. Una transacción en el sistema informático de un hospital sería dar de alta a un paciente o eliminar un ingreso. Una transacción un poco más complicada sería dar de alta a un paciente y asignarlo en una sala disponible de acuerdo a su diagnóstico y sus características personales. En este caso hay que realizar varios cambios sobre la base de datos. Si la transacción falla durante su realización, por ejemplo porque falla el hardware, la base de datos quedará en un estado inconsistente. Algunos de los cambios se habrán hecho y otros no, por lo tanto, los cambios realizados deberán ser deshechos para devolver la base de datos a un estado consistente.

4. Un SGBD debe proporcionar un mecanismo que asegure que la base de datos se actualice correctamente cuando varios usuarios la están actualizando concurrentemente. Uno de los principales objetivos de los SGBD es el permitir que varios usuarios tengan acceso concurrente a los datos que comparten. El acceso concurrente es relativamente fácil de gestionar si todos los usuarios se dedican a leer datos, ya que no pueden interferir unos con otros. Sin embargo, cuando dos o más usuarios están accediendo a la base de datos y al menos uno de ellos está actualizando datos, pueden interferir de modo que se produzcan inconsistencias en la base de datos. El SGBD se debe encargar de que estas interferencias no se produzcan en el acceso simultáneo.

5. Un SGBD debe proporcionar un mecanismo capaz de recuperar la base de datos en caso de que ocurra algún suceso que la dañe. Como se ha comentado antes, cuando el sistema falla en medio de una transacción, la base de datos se debe devolver a un estado consistente. Este fallo puede ser a causa de un fallo en algún dispositivo hardware o un error del software, que hagan que el SGBD aborte, o puede ser a causa de que el usuario detecte un error durante la transacción y la aborte antes de que finalice. En todos estos casos, el SGBD debe proporcionar un mecanismo capaz de recuperar la base de datos llevándola a un estado consistente.

6. Un SGBD debe proporcionar un mecanismo que garantice que sólo los usuarios autorizados pueden acceder a la base de datos. La protección debe ser contra accesos no autorizados, tanto intencionados como accidentales.

7. Un SGBD debe ser capaz de integrarse con algún software de comunicación. Muchos usuarios acceden a la base de datos desde terminales. En ocasiones estos terminales se encuentran conectados directamente a la máquina sobre la que funciona el SGBD. En otras ocasiones los terminales están en lugares remotos, por lo que la comunicación con la máquina que alberga al SGBD se debe hacer a través de una red. En cualquiera de los dos casos, el SGBD recibe peticiones en forma de mensajes y responde de modo similar. Todas estas transmisiones de mensajes las maneja el gestor de comunicaciones de datos. Aunque este gestor no forma parte del SGBD, es necesario que el SGBD se pueda integrar con él para que el sistema sea comercialmente viable.

8. Un SGBD debe proporcionar los medios necesarios para garantizar que tanto los datos de la base de datos, como los cambios que se realizan sobre estos datos, sigan ciertas reglas. La integridad de la base de datos requiere la validez y consistencia de los datos almacenados. Se puede considerar como otro modo de proteger la base de datos, pero además de tener que ver con la seguridad, tiene otras implicaciones. La integridad se ocupa de la calidad de los datos. Normalmente se expresa mediante restricciones, que son una serie de reglas que la base de datos no puede violar. Por ejemplo, se puede establecer la restricción de que cada médico no puede tener asignados más de diez pacientes. En este caso sería

deseable que el SGBD controlara que no se sobrepase este límite cada vez que se asigne un paciente a un médico.

1.1.1 Arquitectura de un SGBD

Entre varias bibliografías consultadas, varios autores coinciden (Mª. J. Ramos, 2006) en que la arquitectura que emplean por referencia los SGBD es conocida como Arquitectura de tres niveles, la cual ha sido propuesta por el grupo ANSI/SPARC. Estos autores plantean que no se asegura que cualquier SGBD se corresponda exactamente con ella, sin embargo, esta arquitectura se corresponde suficientemente bien con un gran número de sistemas.

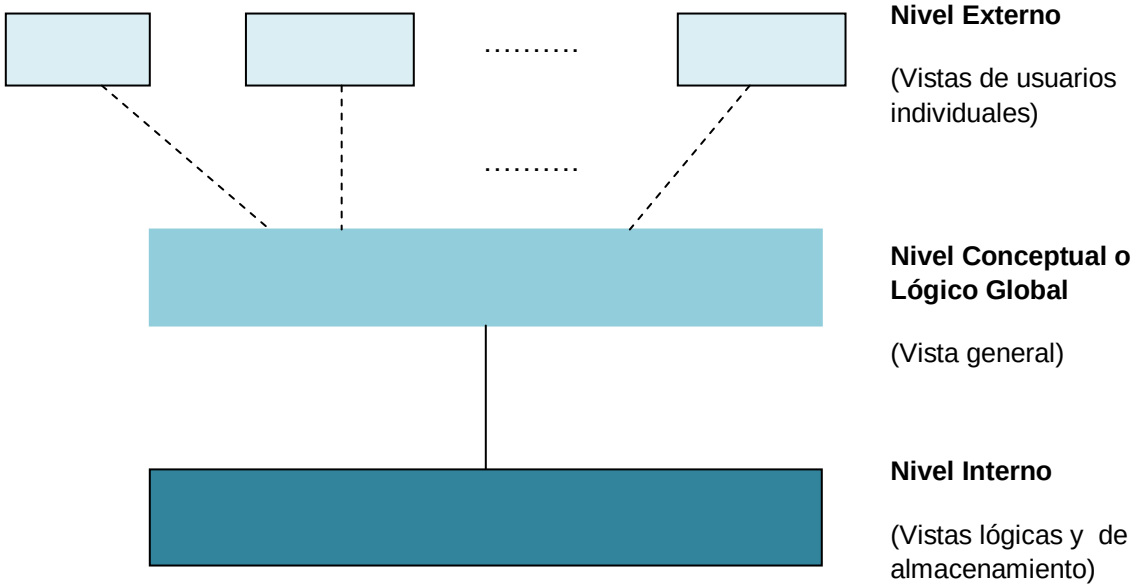


Figura # 1 Arquitectura de un SGBD

El objetivo de esta estructura es formar una separación entre las aplicaciones del usuario y la forma de los datos física. A continuación se describen brevemente cada una de los tres niveles de la arquitectura de los SGBD:

1. **Nivel Interno:** Es el más cercano al almacenamiento físico, o sea, tal y como están almacenados en el servidor. Este nivel describe la estructura física de la base de datos mediante un esquema interno, este esquema se especifica con un modelo físico y describe los detalles de cómo se almacenan físicamente los datos: los archivos que contienen la información, su organización, los métodos de acceso a los registros, los tipos de registros, la longitud, los campos que los componen, etc.

2. **Nivel Conceptual:** Este nivel describe la estructura de toda la base de datos para un grupo de usuarios mediante un esquema conceptual, describe las entidades, atributos, relaciones, operaciones de los usuarios y restricciones, ocultando los detalles de las estructuras físicas de almacenamiento y representa la información contenida en la BD. Pueden usarse modelos de datos de alto nivel o uno de implementación.

3. **Nivel Externo o Vistas:** Es el nivel más cercano a los usuarios, es decir, es donde se describen varios esquemas externos o vistas de usuarios. Cada esquema describe la parte de la base de datos que interesa a un grupo de usuarios siendo en este nivel donde se representa la visión individual de un usuario o de un grupo de usuarios. Se puede usar el modelo de datos de alto nivel o uno de implementación.

Acerca de los sistemas gestores de bases de datos, existen varias definiciones o enfoques que se basan en modelos o estructuras a seguir, por lo que cada sistema gestor se basa en uno de estos modelos, que pueden ser: orientado a objetos, relacional, objeto relacional, multidimensional, entre otros. En los siguientes epígrafes se abordará las características fundamentales de estos modelos, explicándose los más usados y ejemplificándolos con sistemas que los empleen en la actualidad.

1.1.2 Sistemas de Gestión de Bases de Datos Relacionales (SGBDR)

El Modelo de Datos Relacional surgió en 1970 y su creador fue Edgar Codd. Está basado en el algebra relacional y la teoría de conjuntos. Este se caracteriza por disponer que toda la información debe estar contenida en tablas, las cuales están formadas por columnas y filas, y las relaciones entre datos deben ser representadas explícitamente en esos mismos datos. Otras de las características del modelo relacional son:

- Cada tabla es a su vez un conjunto de registros (estos registros constan de varias columnas, campos o atributos), filas o tuplas.
- Cada registro representa un objeto del mundo real.
- Los valores almacenados en una columna deben ser del mismo tipo de dato.
- Todas las filas de una misma tabla poseen el mismo número de columnas.
- No se considera el orden en que se almacenan los registros en las tablas.
- No se considera el orden en que se almacenan las tablas en la base de datos.

Varios autores han expresado lo ventajoso que resulta ser este modelo (Hernández, 1997) (G.W. Hansen, 1997), explicando entre otras **ventajas** que posee este modelo es que provee herramientas que garantizan evitar la duplicidad de registros, que garantiza la integridad referencial, y así al eliminar un registro elimina todos los registros relacionados dependientes y favorece la normalización por ser más comprensible y aplicable.

Como **desventajas** principales ellos plantean que presentan deficiencias con datos gráficos, multimedia, y sistemas de información geográfica y que no se manipulan de forma manejable los bloques de texto como tipo de dato.

Algunos de los SGBD que emplean este modelo son: Oracle, MySQL y Microsoft SQL Server.

1.1.3 Sistemas de Gestión de Bases de Datos Orientadas a Objetos (SGBDOO)

El modelo de bases de datos orientado a objetos es una adaptación a los sistemas de bases de datos. Entre otros, el autor (Manola, 1994) refleja que está basado en el concepto de encapsulamiento de datos y código que opera sobre estos en un objeto. Los objetos estructurados son agrupados en clases. El conjunto de clases está estructurado en subclases y superclases basado en una extensión del modelo entidad-relación. Puesto que el valor de un dato en un objeto también es uno de estos, es posible representar el contenido de los mismos dando como resultado un objeto compuesto.

El modelo orientado a objetos se basa en encapsular código y datos en una única unidad, llamada objeto y el resto del sistema se define mediante un conjunto de mensajes.

Un objeto tiene asociado:

- Un conjunto de variables que contienen los datos del objeto. El valor de cada variable es un objeto.
- Un conjunto de mensajes a los que el objeto responde.
- Un método, que es un trozo de código para implementar cada mensaje. Un método devuelve un valor como respuesta al mensaje.

Nota: El término mensaje en un contexto orientado a objetos se refiere al paso de solicitudes entre objetos sin tener en cuenta detalles específicos de implementación.

Unas de las **ventajas** que posee el modelo orientado a objetos (Moraga, 2002) son la flexibilidad y soporte para el manejo de tipos de datos complejos y la manipulación de datos complejos de forma rápida y ágilmente.

Las **desventajas** de este modelo se basan fundamentalmente en la inmadurez de dicho modelo en el mercado y en la falta de estándares en la industria orientada a objetos.

Ejemplo: ODMG (Object Database Management Group).

A partir de estos dos epígrafes se puede establecer una comparación entre el modelo orientado a objetos y el modelo relacional, mostrando cual es su enfoque y las necesidades que intentan satisfacer cada uno de ellos, ya que son muy comunes estos modelos y que el orientado a objetos sucede al relacional.

SGBDOO	SGBDR
Gestionan objetos en los cuales están encapsulados los datos y las operaciones que actúan sobre ellos. Intentan satisfacer necesidades de aplicaciones más complejas.	Los datos residen en la base de datos. Los procesos se encuentran en las aplicaciones desarrolladas mediante el lenguaje de datos asociado al SGBD. Eficientes para aplicaciones tradicionales de negocios.

Tabla # 1 Comparación entre SGBDOO y SGBDR

La diferencia principal entre los SGBDOO y los SGBDOR radica en que mientras que en una base de datos relacional almacena los datos representados en tablas, en una base de datos objeto-relacional los datos se almacenan como objetos. Un objeto en base de datos orientada a objetos como en programación orientada a objetos es una entidad identificable unívocamente que describe tanto el estado como el comportamiento de una entidad del mundo real. El estado de un objeto es descrito mediante atributos mientras que su comportamiento es definido mediante métodos.

1.1.4 Sistemas de Gestión de Bases de Datos Objeto-Relacionales (SGBDOR)

Un Sistema de Gestión Objeto Relacional contiene dos tecnologías: la tecnología relacional y la tecnología de objetos según lo planteado por: (Villanueva, 2001-2002), debido a que una base de datos objeto relacional es una base de datos que desde el modelo relacional evoluciona hacia una base de datos más extensa y compleja incorporando para obtener este fin, conceptos del modelo orientado a objetos.

Las bases de datos objeto-relacionales tales como Oracle8i y PostgreSQL son compatibles en sentido ascendente con las bases de datos relacionales actuales, por lo que se pueden pasar estas aplicaciones actuales y bases de datos relacionales al nuevo modelo sin tener que reescribirlas. Posteriormente se adaptan las aplicaciones y bases de datos para que utilicen las funciones orientadas a objetos.

Los SGBD que emplean este modelo, pueden construir tipos de objetos complejos, tales como: registros, conjuntos, referencias, listas, pilas, colas y arreglos. Otra de sus características es que se soporta el encadenamiento dinámico y herencia en los tipos de *tupla* o registro. Se pueden compartir varias bibliotecas de clases ya existentes, esto es lo que se conoce como reusabilidad. También da la posibilidad de incluir el chequeo de las reglas de integridad referencial a través de los disparadores. Proporciona un soporte adicional para seguridad y activación de la versión cliente-servidor.

Varios autores (Mª. J. Ramos, 2006) (G.W. Hansen, 1997) han comentado y analizado algunas de las características que hacen este modelo como ventajoso o no, como las mencionadas anteriormente, planteando además que posee **ventajas** como:

- Existe una mayor capacidad expresiva para los conceptos y asociaciones.
- Se pueden crear operadores asignándole un nombre y existencia de nuevas consultas con mayor capacidad consultiva.

Como **desventaja** principal o inconveniente que tienen los SGBDOR es que aumentan la complejidad del sistema y por tanto ocasionan un aumento del coste asociado.

Unos de los ejemplos de SGBD que usan este modelo son, por ejemplo, PostgreSQL y JDBC (Java Data Base Connectivity). Este último fue pensado para interrelacionar bases de datos relacionales con Java, sin embargo incorpora algunos detalles objeto-relacionales.

1.1.5 Bases de Datos Multidimensionales

En un modelo de datos multidimensional los datos se organizan alrededor de los temas de la organización. La estructura de datos manejada en este modelo son matrices multidimensionales o hipercubos, los cuales consisten en un conjunto de celdas donde cada una se identifica por la combinación de los miembros de las diferentes dimensiones y contienen el valor de la medida analizada para dicha combinación de dimensiones, que sería lo conocido en este tema como “hecho”, según lo planteado por varios autores, entre los que se destaca (Kimball, 2002). A continuación se explican los fundamentales elementos que se manejan en este modelo:

- **Hecho:** Es el objeto a analizar, posee atributos llamados de hechos o de síntesis, y son de tipo cuantitativo. Sus valores (*medidas*) se obtienen generalmente por la aplicación de una función estadística que resume un conjunto de valores en un único valor. Por ejemplo: ventas en dólares, cantidad de unidades en inventario, cantidad de unidades de producto vendidas, horas trabajadas, promedio de piezas producidas, consumo de combustible de un vehículo, etc.

- **Dimensiones:** Representan cada uno de los ejes en un espacio multidimensional. Suministran el contexto en el que se obtienen las medidas de un hecho. Algunos ejemplos son: tiempo, producto, cliente, departamento, entre otras. Las dimensiones se utilizan para seleccionar y agrupar los datos en un nivel de detalle deseado. Los componentes de una dimensión se denominan niveles y se organizan en jerarquías, verbigracia, la dimensión tiempo puede tener niveles día, mes y año.

Tanto los hechos como las dimensiones se guardan en tablas. En la figura # 2 se muestra un ejemplo de un modelo multidimensional para apreciar mejor esta estructura, donde de hechos es la tabla ventas y las dimensiones son almacén, producto y tiempo.

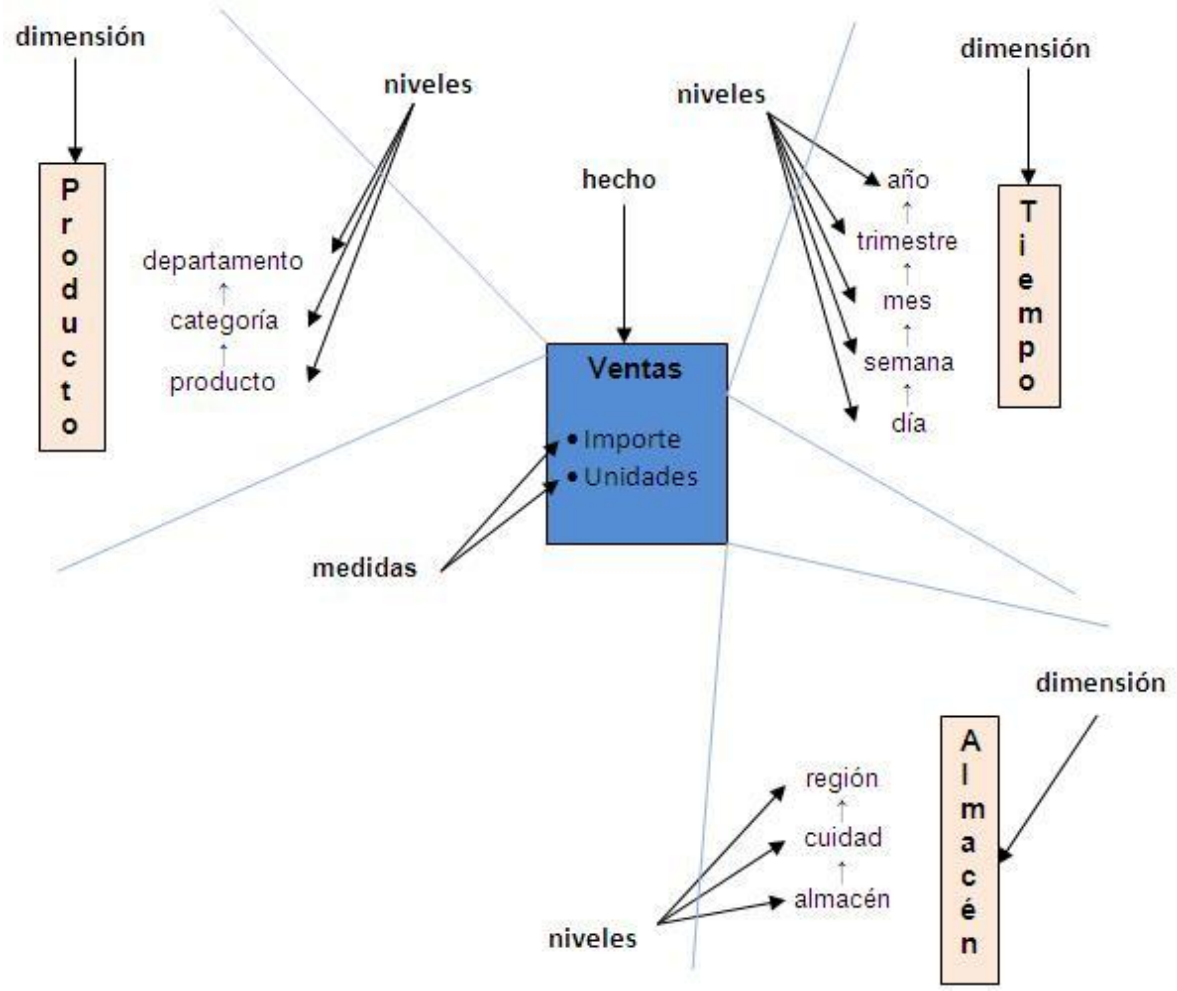


Figura # 2 Modelo Multidimensional

Un modelo multidimensional se puede representar como un esquema en estrella, copo de nieve (*snowflake*) o constelación de hechos (Kimball, 2002) (Chaudhuri, 1997).

- **Esquema en estrella:** Está formado por una tabla de hechos y una tabla para cada dimensión.
- **Esquema copo de nieve:** Es una variante del esquema en estrella que presenta las tablas de dimensión normalizadas.
- **Constelación de hechos:** Son varios esquemas en estrella o copo de nieve que comparten dimensiones.

Ejemplo de cubo: En la figura # 3 se tiene una base de datos que maneja tres dimensiones: países, productos y períodos de entrega (tiempo). Los datos pueden representarse como un cubo de tres dimensiones; cada valor individual de una celda representa la cantidad total de un producto vendido a un país en una fecha determinada.

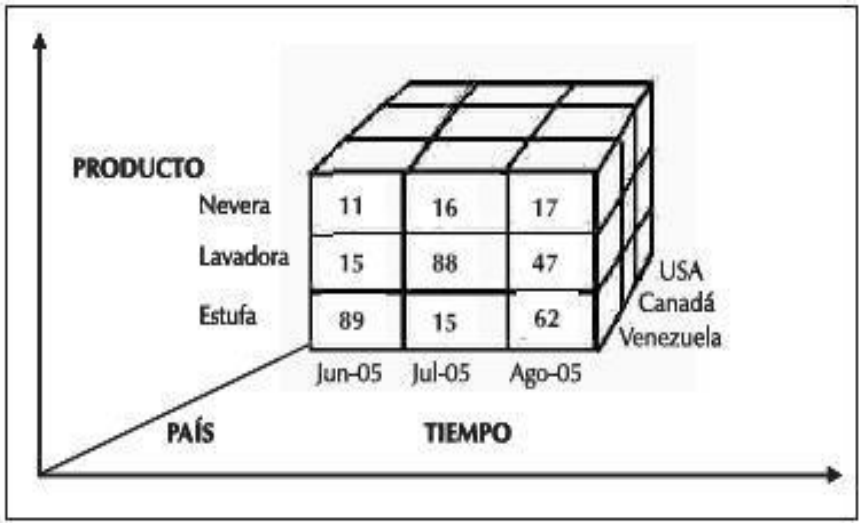


Figura # 3 Cubo de Datos

Para almacenar los datos de las bases de datos multidimensionales es usado la tecnología MOLAP (Multidimensional Online Analitic Process), en español Procesamiento Analítico Multidimensional en Línea (Tamayo & Moreno, 2006).

Como se ha podido observar, cada modelo incorpora características que lo diferencian de los demás modelos, lo cual hace que uno sea mejor que otro en dependencia del fin con que se use. Estos modelos que se han explicado anteriormente son unos de los más usado, pues existen muchos más modelos en la actualidad. A continuación se analizarán algunos de los sistemas más usados en la actualidad y que son los que usan estos modelos hasta aquí mencionados.

1.2 Algunos gestores más usados en la actualidad

Una vez analizado los modelos en los que se basan los SGBD, se propone conocer algunos de los gestores más usados en la actualidad, debido al impacto e importancia que

han tenido en el mundo de la informática. Se estudiaron MySQL, PostgreSQL y Oracle. Es importante destacar que la tendencia hacia la implantación del software libre en el país es creciente, por ende se hace extensivo el uso de PostgreSQL como gestor de bases de datos, debido a que es muy potente, es de código abierto y gratis además, lo cual permite el desarrollo y mejora de este gestor.

1.2.1 Oracle

Oracle es un SGBD del tipo relacional. Surgió a finales de los 70 y fue desarrollado por Oracle Corporation. Es considerado uno de los SGBD más completos actualmente, gracias a características que posee, como:

- Soporte de transacciones.
- Estabilidad.
- Escalabilidad.
- Soporte multiplataforma.

A partir de la versión 10g Release 2, Oracle cuenta con 5 ediciones:

- Oracle Database Enterprise Edition (EE).
- Oracle Database Standard Edition (SE).
- Oracle Database Standard Edition One (SE1).
- Oracle Database Express Edition (XE).
- Oracle Database Personal Edition (PE).

Una de las características que posee Oracle es el Patrón de Consulta, una de las herramientas lógicas más poderosas de SQL. Este permite el reconocimiento de un patrón de consulta mediante su búsqueda por nombre, dirección u otro dato parcialmente recordado. Estos son muy importantes a la hora de realizar consultas, puesto que es muy común que se necesite un texto y no se recuerde cómo fue ingresado.

Otro aspecto importante a destacar en este gestor son los índices, que aumentan la velocidad de respuesta de la consulta y por ello mejoran su rendimiento, optimizando de esta forma su resultado. En Oracle el manejo de estos se realiza de forma inteligente, pues el programador solo crea los índices, sin tener que especificar, explícitamente, cual es el que va a usar.

Las partes principales que componen a Oracle son:

- El Kernel de Oracle.
- Las instancias del Sistema de Datos.
- Los archivos relacionados al sistema de base de datos.

Otra característica de vital importancia es que Oracle es privativo y se requiere de mucho presupuesto para pagar las licencias requeridas para su uso en la implementación de sistemas e implantación de aquellos que lo empleen además. Esto lo pone en desventaja con otros gestores como PostgreSQL, que además de ser una tecnología robusta es gratis y de código abierto.

1.2.2 MySQL

MySQL es un software de fuente abierta, lo cual significa que es posible para cualquier persona usarlo y modificarlo, bajando el código fuente y usarlo sin pagar. Sin embargo esta característica no es para todas las versiones existentes, puesto que ha pasado a ser privativa, lo cual implica gran costo de las licencias necesarias para trabajar con el gestor y esto impida a muchas empresas usarlo, ya sea por no contar con el presupuesto requerido o por problemas con el país al que pertenece la compañía dueña del gestor. Es un SGBD del tipo relacional y multiplataforma ya que funciona en muchas plataformas y/o sistemas operativos como por ejemplo:

- GNU/Linux
- Mac OS X
- OpenBSD

- OS/2 Warp
- Solaris
- Windows 95, Windows 98, Windows NT, Windows 2000, Windows XP, Windows Vista y otras versiones de Windows

MySQL es una base de datos muy rápida en la lectura cuando utiliza el motor no transaccional MyISAM, sin embargo puede provocar problemas de integridad en entornos de alta concurrencia en la modificación. Presenta problemas de baja concurrencia en la modificación de datos y en cambio el entorno es intensivo en lectura de datos en aplicaciones web, lo que hace a MySQL ideal para este tipo de aplicaciones.

Algunas de las características de este sistema son:

- Procedimientos almacenados.
- Disparadores.
- Cursores.
- Soporte a VARCHAR.
- Motores de almacenamiento independientes (MyISAM para lecturas rápidas, InnoDB para transacciones e integridad referencial).
- Transacciones con los motores de almacenamiento InnoDB, BDB Y Clúster; puntos de recuperación (savepoints) con InnoDB, lo cual permite que el usuario pueda escoger los que le sea más adecuado a cada tabla de la base de datos.
- Uso de multihilos mediante hilos del kernel.
- Tablas hash en memoria temporales.

Este es un gestor de bases de datos muy potente, es muy usado en la actualidad, pero lo pone en desventaja el de hecho de no ser libre y gratis en su totalidad ya que muchas empresas, como es el caso de las de nuestro país, se pueden privar del uso de estas versiones privativas.

Sin embargo también posee otras características además de las mencionadas que lo hacen ventajoso, como son:

- Usa tablas en disco b-tree para búsquedas rápidas con compresión de índice.
- Seguridad: ofrece un sistema de contraseñas y privilegios seguro mediante verificación basada en el host y el tráfico de contraseñas está cifrado al conectarse a un servidor.
- Soporta almacenamiento de gran cantidad de datos.
- También proporciona la agrupación de transacciones, pues reúne múltiples transacciones de varias conexiones con el objetivo de incrementar el número de estas por segundo.

1.2.3 PostgreSQL

PostgreSQL es un potente sistema de base de datos objeto relacional, libre (su código fuente está disponible) y liberado bajo licencia BSD. Tiene más de 15 años de activo desarrollo y arquitectura probada que se ha ganado una muy buena reputación por su confiabilidad e integridad de datos. Funciona en todos los sistemas operativos importantes, incluyendo Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64), y Windows. Entre sus principales características se encuentran:

- Soporta casi toda la sintaxis SQL pues tiene soporte total para llaves extranjeras, *joins*, vistas, disparadores y procedimientos almacenados (en múltiples lenguajes).
- Integridad transaccional: Obedece completamente a la especificación ACID (estándar internacional que permite que se cumpla con esta característica).
- Acceso concurrente multiversión, **MVCC Control de Concurrencia Multi-Versión** (Multi-Version Concurrency Control), no se bloquean las tablas, ni siquiera las filas cuando un proceso escribe. Es la tecnología que PostgreSQL usa para evitar bloqueos innecesarios. Mediante el uso de MVCC, este gestor se evita el problema de que procesos lectores estén esperando a que se termine de escribir para poder leer. En su lugar, el gestor mantiene una ruta a todas las transacciones realizadas por los usuarios de la base de datos, siendo capaz entonces de manejar los

registros sin necesidad de que los usuarios tengan que esperar a que los registros estén disponibles.

- Cliente/Servidor: Usa una arquitectura proceso-por-usuario cliente/servidor, la cual es similar al método del Apache 1.3.x para manejar procesos. Existe un proceso maestro que se ramifica para proporcionar conexiones adicionales para cada cliente que intente conectar a PostgreSQL.

- **Write Ahead Logging (WAL)**: La característica de PostgreSQL conocida como Write Ahead Logging incrementa la dependencia de la base de datos al registro de cambios antes de que estos sean escritos en la base de datos. Esto garantiza que en el hipotético caso de que la base de datos deje de funcionar, existirá un registro de las transacciones a partir del cual se podrá restaurar la base de datos. Esto puede ser enormemente beneficioso ya que cualesquiera de los cambios que no fueron escritos en la base de datos cuando dejó de funcionar le servidor, pueden ser recuperados usando el dato que fue previamente registrado. Una vez el sistema ha quedado restaurado, un usuario puede continuar trabajando desde el punto en que lo dejó cuando cayó la base de datos.

- Lenguajes Procedurales: Tiene soporte para lenguajes procedurales internos, incluyendo un lenguaje nativo denominado PL/pgSQL. Este lenguaje es comparable al lenguaje procedural de Oracle, PL/SQL. Otra ventaja de PostgreSQL es su habilidad para usar Perl, Python, o TCL como lenguaje procedural embebido.

- Interfaces con lenguajes de programación: La flexibilidad del API de PostgreSQL ha permitido a los vendedores proporcionar soporte al desarrollo fácilmente para el RDBMS PostgreSQL. Estas interfaces incluyen Object Pascal, Python, Perl, PHP, ODBC, Java/JDBC, Ruby, TCL, C/C++, Pike, etc.

- Herencia de tablas:

- Incluye la mayoría de los tipos de datos SQL92 y SQL99 (INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, DATE, INTERVAL, y TIMESTAMP), soporta almacenamiento de objetos grandes binarios, además de tipos de datos y operaciones geométricas.

- Puntos de recuperación a un momento dado, tablespaces, replicación asincrónica, transacciones jerarquizadas (savepoints), backups en línea.
- Un sofisticado analizador/optimizador de consultas.
- Soporta juegos de caracteres internacionales, codificación de caracteres multibyte.

Según lo expresado en bibliografías oficiales del gestor (Medina, 2006), con PostgreSQL se han aumentado y mejorado sus características y capacidades, aunque el trabajo continúa en todas las áreas incluyendo en sus principales mejoras, entre las que se encuentran:

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

La velocidad del código del motor de datos ha sido incrementada aproximadamente en un 20-40%, y su tiempo de arranque ha bajado el 80% desde que la versión 6.0 fue lanzada. (Medina, 2006)

Como en todo SGBD, en PostgreSQL se realizan ciertas tareas de administración del sistema, que permiten mantenerlo en condiciones favorables al desarrollo y desempeño de las aplicaciones. El Vacuum es una de estas tareas, es un proceso de limpieza que se encarga de:

- Recuperar el espacio de disco ocupado por filas modificadas o borradas.
- Actualizar las estadísticas usadas por el planificador / optimizador.
- Evitar la pérdida de datos muy antiguos debido al rehúso del identificador de transacción.

La frecuencia con que se ejecute este proceso depende de cada instalación y, además, se puede ejecutar en paralelo con operaciones de actualización pero no de definición, se puede ejecutar desde el sistema operativo o desde SQL (Medina, 2006). Este proceso se puede entender mediante la siguiente figura:

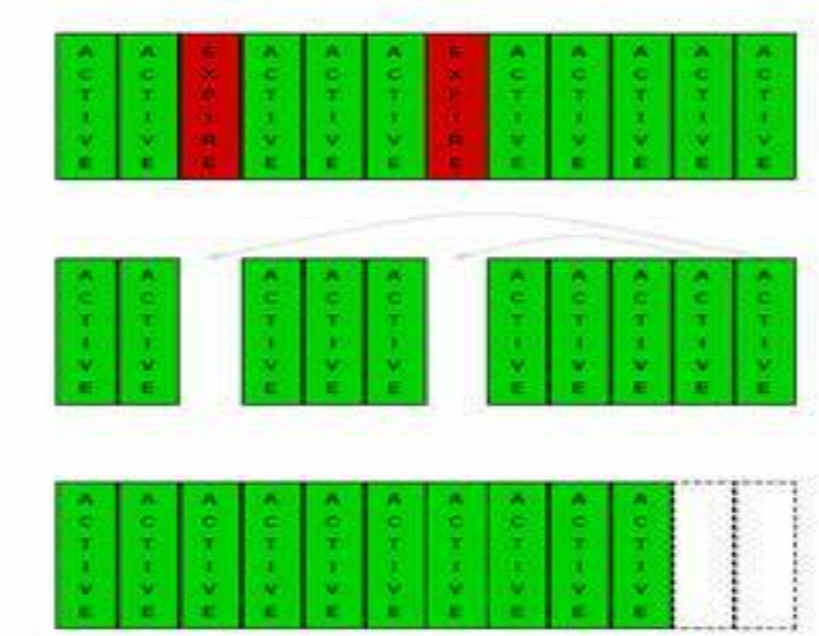


Figura # 4 Funcionamiento de Vacuum

Lo que se representa en esta figura, a grandes rasgos, es un fichero conformado por varios bloques de memoria, algunos de ellos no se usan (señalados en rojo). El Vacuum se encarga de removerlos, sustituyendo su espacio por los siguientes bloques que si son usados habitualmente.

Esto es ventajosos, pues proporciona que la memoria sea usada óptimamente, y puede facilitar la búsqueda de información, así como la rapidez de la ejecución de las consultas. Es recomendado realizarse el Vacuum diariamente, y puede ser ejecutado manualmente

cuando el administrador desee, o también se puede configurar para que sea ejecutado a una hora y fecha determinadas.

A continuación se mencionan algunos de los programas usados para administrar PostgreSQL y una pequeña descripción de cada uno de ellos.

pgAdmin III:

- Herramienta gráfica, permite ver la estructura de las bases de datos, realizar operaciones SQL, ver datos, operaciones de administración.
- Diseñada para ejecutarse en muchos sistemas operativos (Windows, Linux, MacOS).

phppgAdmin III: Es una poderosa herramienta de administración basada en una interfaz Web para bases de datos PostgreSQL.

- Equivalente a la anterior herramienta, pero está realizada en una interfaz web con PHP.
- Tiene la ventaja de que no requiere la instalación en los clientes, así como se puede centralizar la conexión a las bases de datos, impidiendo el acceso desde estaciones de trabajo y, a la vez, facilitando el trabajo a los potenciales usuarios porque no tienen que dedicar tiempo a configurar conexiones.
- Dispone de soporte para procedimientos almacenados, triggers y vistas.

PgAccess:

- Es una interfaz gráfica para el gestor de bases de datos PostgreSQL escrito por Constantin Teodorescu en el lenguaje Tcl/Tk.
- Permite al usuario interactuar con PostgreSQL de una manera similar a muchas aplicaciones de bases de datos para PC, con menús de opciones y diversas herramientas gráficas. Esto significa que el usuario puede evitar la línea de comandos para la mayoría de las tareas.
- Tiene opciones para creación de formularios e informes.

pgExplorer:

- Herramienta de desarrollo para PostgreSQL con una amplia interfaz gráfica.
- Se incluye una vista en árbol de las bases de datos y sus respectivos objetos.
- Permite ingeniería inversa.
- Tiene asistentes para generar sentencias SQL.
- Incluye diseñador gráfico de consultas.

1.3 Aspectos de interés

Es importante conocer algunos conceptos de interés acerca de las bases de datos como la optimización, el monitoreo, el rendimiento, entre otros. Algunos de ellos serán explicados a continuación, debido a su relevancia en la investigación de este trabajo.

1.3.1 Monitoreo

El monitoreo es una de las actividades esenciales a la hora de administrar correctamente un sistema de base de datos. Mediante esta actividad de administración se puede conocer como es el comportamiento del sistema y en qué medida se realiza su actividad.

El monitoreo se puede realizar mediante herramientas del sistema operativo que se esté usando o mediante herramientas y comandos del propio SGBD. Normalmente para monitorear los servidores es necesario el uso de comandos, los cuales son invocados manualmente por el administrador de la base de datos, ya que no abundan herramientas que realicen esta tarea.

El monitoreo, o monitorización como también suele llamarse se centra generalmente en aspectos como: el uso de swap, el uso de los discos y el monitoreo interactivo, que se refiere a eventos ocurridos en el ***postmaster***. Todo esto permite realizar un análisis del rendimiento de los servidores.

Postmaster: Es el proceso inicial que gestiona los accesos multiusuario y multiconexión, levanta la memoria compartida y está al tanto de solicitudes de nuevas conexiones. Lanza procesos de atención de demanda, realizando las operaciones sobre la base de datos a

solicitud de los clientes y realiza el enlazado con los archivos de datos según lo expresado por el autor: (Medina, 2006).

1.3.2 Recolector de Estadísticas

Si se activa el recolector de estadísticas este recoge información de la actividad del servidor. La información que se obtiene tiene que ver con los accesos a las tablas e índices individuales y por páginas, las instrucciones ejecutadas en los procesos servidores lecturas de disco, el uso de la cache, entre otros. La ejecución del recolector produce una ligera sobrecarga en el servidor, la cual puede ser de 0.5 segundos. (Medina, 2006)

1.3.3 Rendimiento

Un Sistema de Gestión de Bases de Datos debe llevar la cuenta de gran cantidad de información referente a la estructura de una base de datos con el fin de efectuar sus funciones de gestión de datos. En una base de datos relacional, esta información está almacenada típicamente en el *catálogo de sistema.*

El catálogo de sistema es una colección de tablas especiales en una base de datos que son propiedad, están creadas y son mantenidas por el propio SGBD. Estas tablas del sistema contienen datos que describen la estructura de la base de datos. Las tablas del catálogo de sistema son automáticamente creadas al crear la base de datos. Generalmente se recogen todas juntas bajo un «id-usuario de sistema» o superusuario especial con un nombre como SYSTEM, SYSIBM, MASTER o DBA.

El acceso de los usuarios al catálogo de sistema es de solo lectura. El SGBD impide a los usuarios actualizar o modificar directamente las tablas del sistema, ya que tales modificaciones destruirán la integridad a la base de datos. En su lugar, el propio SGBD tiene a su cargo insertar, eliminar y actualizar filas de las tablas del sistema cuando modifica la estructura de una base de datos.

Las sentencias DDL (Definition Data Language, en español: Lenguaje de Definición de Datos) tales como CREATE, ALTER, DROP, GRANT y REVOKE producen cambios en las tablas del sistema como consecuencia de sus acciones.

Cada tabla del catálogo de sistema contiene información referente a una sola clase de elemento estructural de la base de datos. Aunque los detalles varían, casi todos los productos SQL comerciales incluyen tablas del sistema que describen cada una de estas cinco entidades como ha sido planteado en otras bibliografías por el autor: (Gamino, 2002).

- **Tablas:** El catálogo describe cada tabla de la base de datos, identificando su nombre, su propietario, el número de columnas que contiene, su tamaño, etc.
- **Columnas:** El catálogo describe cada columna de la base de datos, proporcionando el nombre de la columna, la tabla a la que pertenece, su tipo de datos, su tamaño, si están permitidos los NULL.
- **Usuarios:** El catálogo describe a cada usuario autorizado de la base de datos, incluyendo el nombre, una forma cifrada de la contraseña del usuario y otros datos.
- **Vistas:** El catálogo describe cada vista definida en la base de datos, incluyendo su nombre, el nombre de su propietario, la consulta que define la vista, etc.
- **Privilegios:** El catálogo describe cada grupo de privilegios concedidos en la base de datos, incluyendo los nombres del donante y el donatario, los privilegios concedidos, el objetivo sobre el cual se han concedido los privilegios, etc.

1.3.4 Reindexación

Un elemento muy importante en las bases de datos son los índices, estos permiten que la búsqueda de la información sea más rápida, pues proporcionan un orden e indicador de búsqueda para las consultas.

En algunas ocasiones puede haber dificultades con los índices, ya sean causados por problemas de hardware o de software, o puede que estos sean incorrectos. En estos casos es necesario realizar la reindexación, que es otra tarea de administración, que elimina las páginas muertas, o sea, que no son usadas, y vuelve a reasignar los índices, de manera que estos optimicen la búsqueda de los datos y mejoren el rendimiento de los servidores.

1.4 Optimización

La optimización es un aspecto de suma importancia, ya que esta actividad permite mejorar el rendimiento y que las bases de datos funcionen en las condiciones óptimas de acuerdo a sus características y propósitos. A continuación se explicará cómo se comporta esta en la actualidad y cuáles son las características que lo identifican.

Una metodología bien planeada es la clave para realizar un proceso exitoso de ajuste de rendimiento. Diferentes estrategias de afinamiento varían en su efectividad y sistemas con diferentes propósitos, como los sistemas de procesamiento de datos o los de apoyo a la toma de decisiones, requieren distintos métodos de afinamiento.

El afinamiento de una base de datos es necesario que se comience a realizar desde la fase de diseño, en vez de esperar hasta después de la implementación del sistema. Basado en cuando se comienza a realizar el afinamiento se puede clasificar en:

- **Afinamiento Proactivo:** es cuando el afinamiento se realiza mientras se desarrolla el sistema.

- **Afinamiento Reactivo:** es el que se realiza para mejorar sistemas en explotación.

De estos dos métodos el más efectivo, en gran medida, es el Afinamiento Proactivo, el cual será abordando con mayor profundidad y el cual necesita para su implementación que desde etapas muy tempranas del desarrollo los analistas sean capaces de establecer, junto a los clientes, metas y expectativas realistas de rendimiento, para que durante el análisis y diseño los arquitectos y diseñadores del sistema y la base de datos sean capaces de determinar qué técnicas aplicar para mejorar el rendimiento, así como qué configuración de hardware será necesaria para cumplir con estas metas y expectativas.

Al diseñar un sistema para que tenga buen rendimiento se pueden minimizar sus costos de puesta a punto para la explotación y de administración posterior. Las figuras que se muestran a continuación ilustran el costo relativo de realizar afinamiento durante la vida de una aplicación y los beneficios que esto trae.

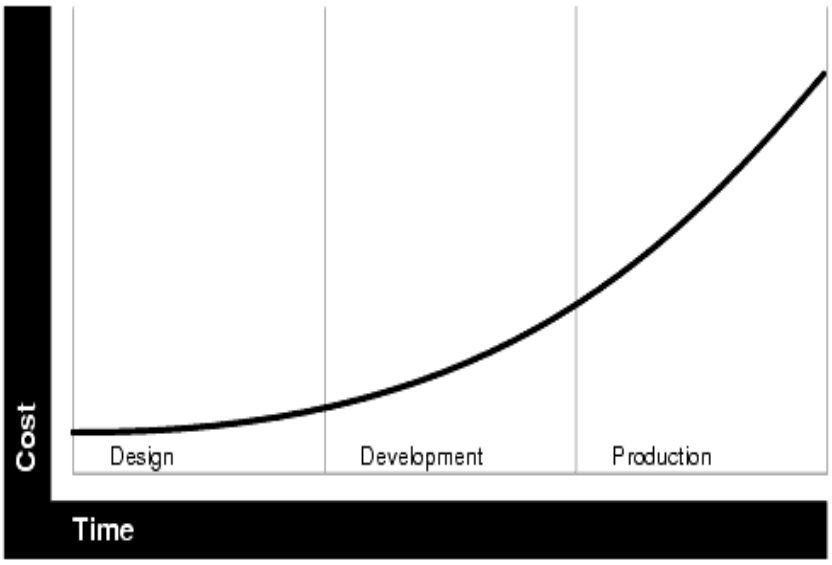


Figura # 5 Costo relativo al optimizar durante el ciclo de vida de un software

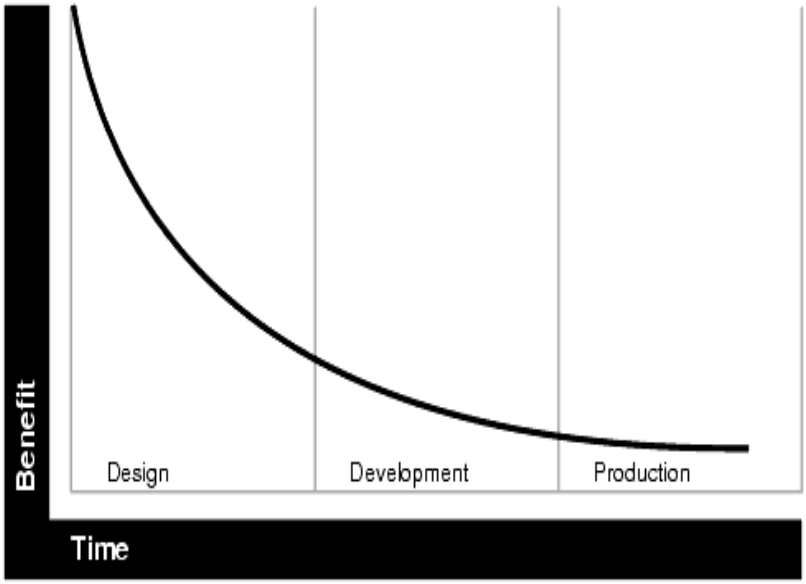


Figura # 6 Beneficio relativo al optimizar durante el ciclo de vida de un software

El proceso de afinamiento no debe comenzar cuando los usuarios de un sistema se empiezan a quejar sobre el rendimiento y los tiempos de respuesta, cuando esto sucede

usualmente es demasiado tarde para implementar algunas de las estrategias de afinamiento más efectivas. Cuando sucede esto si no es posible rediseñar completamente la base de datos, entonces lo único que se puede hacer para mejorar el rendimiento del sistema es reasignar más memoria y mejorar los parámetros de entrada y salida de disco.

Por ejemplo: Hay un banco que tiene un cajero y un gerente y tiene una regla del negocio que especifica que cada retiro de más de $20 debe ser aprobado por el gerente. Si todos los días se forma un larga cola da espera para poder sacar dinero una buena decisión sería contratar más cajeros y aumentar el número de estos hasta 10, pero entonces se provocaría un cuello de botella en la función del gerente y por mucho que tratemos de mejorar el rendimiento del sistema sino se modifica esta regla de aprobar retiros de más de $20 el cuello de botella va a seguir ocurriendo.

Es posible mejorar un sistema en explotación reactivamente comenzando por el paso final e ir subiendo poco a poco a los otros, encontrando y arreglando la mayor cantidad posible de cuellos de botella posibles. Una meta común es hacer que PostgreSQL corra más rápido en la plataforma en la que se encuentra instalado, ya sea afinando los parámetros de rendimiento o añadiendo más o mejor hardware.

Aunque el afinamiento proactivo sea más eficiente y menos costoso que el reactivo, esto no implica que se pueda prescindir de este último porque aunque un sistema este bien diseñado para alcanzar un rendimiento optimo este se degrada con el uso lo que indispensable el mantenimiento de los sistemas en explotación.

Por otra parte es importante conocer que el problema que pueda provocar un mal rendimiento puede ocurrir después de iniciar el sistema, por ello, para comenzar, el rendimiento debe ser monitoreado y evaluado regularmente después de comenzar la operación. En los próximos epígrafes se comenta como es realizada la optimización por algunos de los más usado SGBD actualmente.

1.4.1 Optimización para Oracle

Para realizar un correcto monitoreo para optimizar el servidor, es necesario planificar un diseño de cómo se va a monitorear, y hacerlo en etapas tempranas. Oracle estipula una serie de pasos dirigidos al afinamiento o puesta a punto de sus servidores según lo

investigado en bibliografías como las notas para el afinamiento de bases de datos en Oracle propuestos por el Okinawa International Centre y el Japan International Cooperation Agency, estos son:

1. Optimización de Reglas del Negocio.
2. Optimización del Diseño de la Base de Datos.
3. Optimización del Diseño de la Aplicación.
4. Optimización de los Caminos de Acceso.
5. Optimización de la Memoria.
6. Optimización del I/O y la estructura física.

1.4.2 Optimización para MySQL

La optimización es una tarea compleja, porque requiere un conocimiento de todo el sistema a optimizar. Se podría optimizar sólo algunos aspectos teniendo poco conocimiento del sistema o aplicación, pero cuanto más óptimo se quiera el sistema, más se tiene que conocer acerca del mismo. Para el caso particular de MySQL Sun Microsystems propone centrar la optimización en los siguientes aspectos:

- Optimizar sentencias SELECT y otras consultas.
- Optimizar los posibles bloqueos.
- Optimizar el diseño de datos.
- Optimizar el servidor MySQL
- Optimizar el acceso a disco.

1.4.3 Optimización para PostgreSQL

En la actualidad, la tarea de optimizar se realiza en PostgreSQL cambiando algunos de los parámetros de configuración, a conveniencia de los administradores para con el rendimiento, de manera desorganizada y en muchos casos a ciegas prácticamente. En

otros casos emplean una serie de pasos que no son tan conocidos, que han sido elaborados por Josh Berkus, integrante de la comunidad Internacional de PostgreSQL. Estos pasos son más bien la primera forma que se describía, pero más organizado.

Sin embargo, no poseen alguna técnica o procedimiento que ofrezca una visión práctica del estado del rendimiento actual del servidor, además de que puede resultar un poco esquemático, teniendo en cuenta que el rendimiento varía según múltiples factores en cada uno de los casos, y que la optimización se realiza del mismo modo siempre.

A continuación se mencionan los aspectos o puntos para optimizar en PostgreSQL:

- **Hardware:**

  Aquí se manejan elementos como el CPU, la RAM, la red, entre otros.

- **OS/Filesystem.**

- **Postgresql.conf:**

  Este es el archivo donde se guarda la configuración del PostgreSQL. Aquí se optimizan parámetros como la cache, la memoria compartida, la paginación. En PostgreSQL estos aspectos son vistos de la siguiente forma: shared_buffers, cache_miss, wal_buffers, full_page_writes, effective_cache_size.

- **Diseño de la aplicación:**

  Esto trata sobre optimizar el diseño de las tablas de la base de datos, la forma de indexar, y entra aquí el tema de indexación (visto en el epígrafe superior), el diseño de las consultas, entre otros.

- **Afinamiento de consultas:**

  Este punto es sobre optimizar las consultas SQL realizadas en la base de datos, y es donde toma un valor importante el "*explain analyze*", para conocer el camino de la ejecución de las consultas y como poder mejorarlas a partir de esto.

A partir de un análisis de estos epígrafes se puede apreciar que aun queda mucho por mejorar este proceso de optimización, de modo tal que se pudieran inlcuir nuevos pasos o

aspectos a optimizar y mejorar los que ya existen, haciéndolos de este modo más precisos y eficientes al aplicarlos.

Conclusiones

En este capítulo se ha mencionado brevemente las características de los SGBD así como sus funcionalidades principales. Se han explicado algunos de los tipos de SGBD más usados actualmente, las ventajas y desventajas que ellos proporcionan, así como algunos conceptos importantes a conocer relacionados con el rendimiento y el funcionamiento de los servidores de bases de datos. Se abordó el tema de la optimización en la actualidad, es decir, de qué forma es llevada a cabo. A partir de este análisis parcial se concluye en que:

- No existen abundantes bibliografías sobre la optimización de los servidores de bases de datos.
- No existen guías completas o lo suficientemente explicativas para optimizar las bases de datos.
- Las actividades existentes para optimizar son teóricas y describen casos generales en la mayoría de los casos.
- Los administradores no siempre conocen las estadísticas más comunes de la actividad de la base de datos que les permita conocer donde es realmente necesario optimizar.

Capítulo 2 Descripción de la solución propuesta

En el presente capítulo se muestra una guía práctica para el afinamiento y optimización de servidores de bases de datos de PostgreSQL, el cual será aplicado al Centro de Tecnologías de Almacenamiento y Análisis de Datos (CENTALAD) como parte de los servicios que se ofrecen en este centro. La misma es obtenida como resultado de 2 elementos fundamentales:

1. Bajo rendimiento del servidor de PostgreSQL del proyecto PATDSI y la necesidad de mejorar este para un buen desempeño de la aplicación.
2. La investigación realizada en el Capítulo 1, teniendo en cuenta las prácticas de afinamiento aplicadas por importantes empresas de desarrollo de bases de datos en la puesta a punto de servidores.

En el desarrollo del capítulo se incluirán consultas de ayuda en cada uno de los pasos, las cuales permitirán conocer el estado actual de la actividad de la base de datos en tiempo real en aspectos importantes de conocer a la hora de afinar y poner a punto el servidor. Se muestran correctas prácticas de programación SQL a seguir, teniendo en cuenta la arquitectura de PostgreSQL.

2.1 Pasos para la optimización de PostgreSQL

A partir de las teorías existentes acerca de la optimización de servidores de PostgreSQL, se ha construido una guía de pasos necesarios para realizar esta importante actividad. Algunos de estos planteados hasta el momento, y otros tomados de varios SGBD. De esta forma se unifica el conocimiento para ponerlo en las manos de los administradores

de bases de datos y que estos lo lleven a la práctica, con características adicionales que los ayuden a realizar más eficientemente su trabajo.

En el siguiente epígrafe se muestran estos pasos, así como una descripción más detallada de cada uno de ellos en sus sub epígrafes.

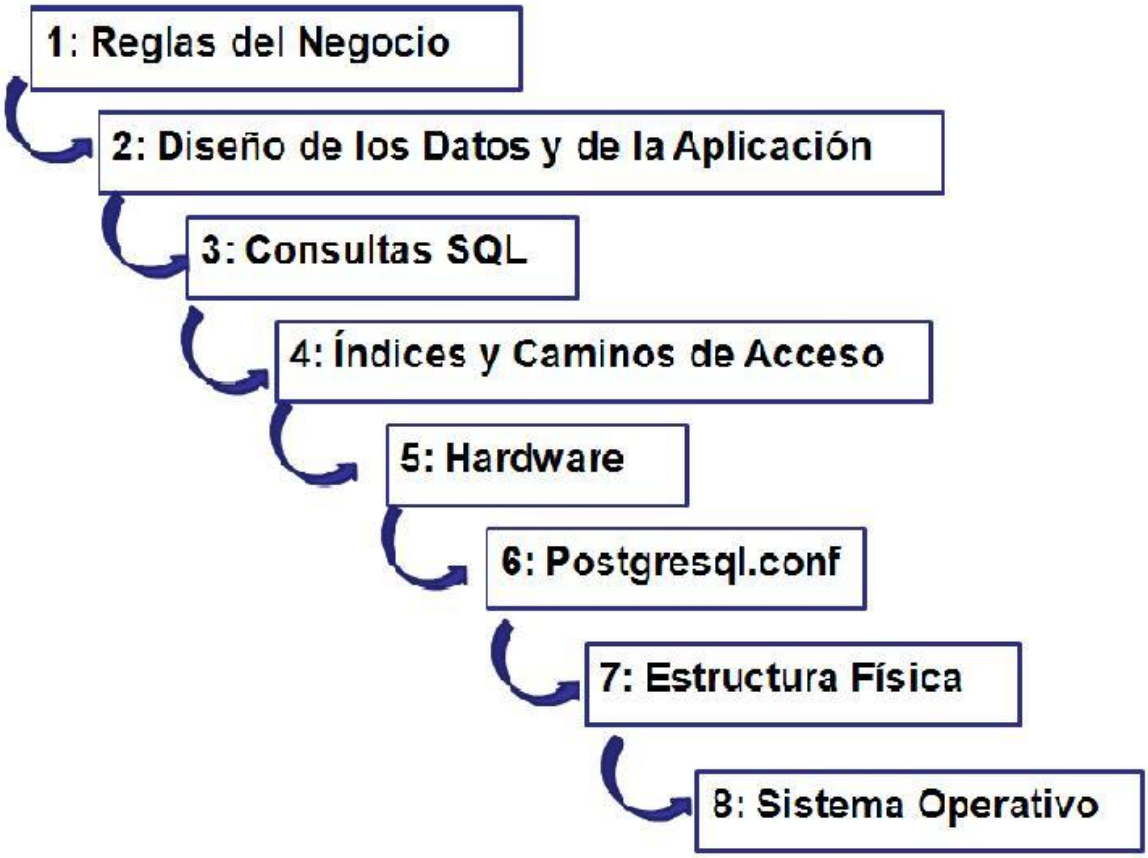


Figura # 7 Pasos de afinamiento en PostgreSQL

2.1.1 Optimizar las reglas del negocio

Este primer paso consiste en determinar las necesidades del negocio y los requerimientos de rendimiento que satisfacen esas reglas. Para un rendimiento óptimo puede ser necesario adaptar las reglas del negocio. Esto conlleva a tener que realizar un profundo análisis a la hora de realizar el análisis y diseño de la aplicación. Los problemas de configuración son irrelevantes a este nivel. De esta forma los planificadores aseguran que

se cumple que los requerimientos de rendimiento se corresponden directamente con lo que concretamente requiere el negocio.

Cuando se definen las reglas del negocio, primero se debe mantener un nivel de abstracción lo más alto posible en vez de describir en detalles cada una de las reglas y los requerimientos de rendimiento. Luego se debe determinar el tipo de sistema que se va a emplear de acuerdo a los requerimientos del sistema. Los tipos de sistema pueden ser los siguientes:

- **Sistema de Procesamiento de Transacciones en Línea** (OLTP por sus siglas en inglés): Las aplicaciones OLTP se caracterizan por la creación de muchos usuarios, actualizaciones o recuperación de registros individuales. Las bases de datos OLTP se perfeccionan para actualización de transacciones. Las aplicaciones OLTP y bases de datos de este tipo tienden a ser organizados alrededor de procesos específicos (como órdenes de entrada).

- **Sistema de Apoyo a la Toma de Decisiones** (DSS): Es una de las herramientas más emblemáticas del Business Intelligence ya que, entre otras propiedades, permiten resolver gran parte de las limitaciones de los programas de gestión. Entre sus características principales están: Rapidez en el tiempo de respuesta, cada usuario dispone de información adecuada a su perfil, disponibilidad de información histórica, entre otras. Destacar que los DSS suelen requerir (aunque no es imprescindible) un motor OLAP subyacente, que facilite el análisis casi ilimitado de los datos para hallar las causas raíces de los problemas/pormenores de la compañía.

- **Sistema de Procesamiento Analítico en Línea** (OLAP): OLAP describe una clase de tecnologías diseñadas para mantener específicamente el análisis y acceso a datos y viene a ser un sinónimo con vistas multidimensionales de los datos del negocio. Estas vistas multidimensionales se apoyan en la tecnología de bases de datos multidimensionales. Estas vistas multidimensionales proporcionan la base técnica para cálculos y análisis requeridos por las Aplicaciones del Negocio Inteligente.

Se ha podido apreciar hasta aquí el vínculo entre estas 3 tecnologías. Las características de los sistemas OLAP y OLTP pueden evidenciarse mediante la siguiente tabla, donde se presentan a modo de comparación las características de ambos.

OLTP (Relacional)	OLAP (Multidimensional)
• **Atomizado**	• Resumido
• **Datos Presentes**	• Datos históricos
• **Orientadas al proceso**	• Orientadas al tema
• **Un registro a la vez**	• Muchos registros a la vez

Tabla # 2 Comparación entre sistemas OLAP y OLTP

La próxima actividad a realizar en este paso es determinar los posibles cuellos de botella que podrían existir a partir de las reglas del negocio en el mismo.

2.1.2 Optimizar el diseño de los datos y de la aplicación

Cuando se diseñan los datos se debe determinar primeramente cuáles de ellos deben persistir en la aplicación, después se debe considerar de estos qué relaciones y atributos de cada una de estas son más importantes, y por último estructurar la información de forma tal que cumpla con los requerimientos del sistema.

Después de concluido este proceso se debe normalizar los datos para evitar redundancias, y por tanto con las excepción de las llaves primarias todos los datos deben estar almacenados solamente una vez en la base de datos (con la normalización se trata de minimizar la redundancia, pues tratando de eliminarla totalmente el diseño puede complejizarse en gran medida y afectar el rendimiento del sistema). Como buena práctica a seguir, se recomienda luego de haber normalizado los datos, desnormalizar por razones de rendimiento, aunque se debe ser cuidadoso a la hora de realizar este proceso y solo desnormalizar donde sea necesario.

Esta tarea de desnormalizar es una práctica común para ganar en rendimiento a la hora de diseñar aplicaciones. Antes de hacer esto se debe estar seguro que el diseño actual

causa o puede causar problemas de rendimiento y que no es la aplicación, la incorrecta configuración del servidor o una red sobresaturada y poco confiable la que está causando los verdaderos problemas. Si se decide modificar el diseño de la base de datos normalizada es necesario tener en cuenta los requerimientos de rendimiento del sistema que se está desarrollando, tenerlos debidamente identificados y documentados es casi tan importante como los requerimientos funcionales, ya que contribuye a que se puedan tomar decisiones para mejorar el rendimiento de la aplicación desde etapas tempranas de su diseño y no tener que realizar parches y modificaciones tardías que puedan comprometer la calidad o el tiempo de desarrollo de la misma al introducir cambios inesperados en esta. Otro aspecto a tener en cuenta es una estrategia de prueba que cuente con las siguientes características: repetitiva, definida, administrada y optimizable en orden de poder garantizar que las mejoras de diseño, con el objetivo de ganar rendimiento, están realmente funcionando y que no se está complicando el diseño de datos en vano.

En determinado momento podría parecer una buena idea para solucionar problemas de rendimiento en una base de datos remover algunas reglas y restricciones de la misma y encargar de chequear estas a la aplicación. En ciertos tipos de sistema, como aquellos que utilizan arquitectura cliente-servidor se verá realmente una mejora en el desempeño global de la solución, al encargar a muchos procesadores lo que antes quizás solo realizaba uno, pero hay que tener mucho cuidado y analizar si realmente lo que se está haciendo no es moviendo el problema de lugar, o si al eliminar algunas reglas y restricciones se está comprometiendo la integridad de los datos e impidiendo que en un futuro los datos almacenados puedan ser utilizados por otra aplicación, algo altamente probable si es un sistema empresarial.

Parte del proceso del diseño de los datos es determinar cuáles atributos deben ser indexados y cuáles no. Por esto se debe tener presente aspectos como cuáles son los campos de mayor frecuencia de acceso, cuáles son utilizados comúnmente como criterio de comparación en consultas *where*, entre otros.

Aunque en PostgreSQL se pueden modelar sistemas del tipo Objeto-Relacional estos no tienen un alto rendimiento y si el sistema está centrado en obtener buenos tiempos de respuesta debe evitarse el uso de este modelo.

Para que la base de datos tenga un buen rendimiento además de tener un buen diseño, es necesario que la aplicación que accede a sus datos lo haga correctamente y trate de minimizar la cantidad de conexiones y transacciones en la medida de lo posible.

Algunas buenas prácticas para el acceso a los datos en PostgreSQL son:

- **Utilizar consultas preparadas:** Establecer una consulta como "consulta preparada" es una posibilidad que da PostgreSQL y que permite, más que tener un procedimiento almacenado, que ahorre una serie de pasos al ejecutar la consulta, lo cual contribuye notablemente a la mejora del rendimiento.
- **Utilizar parseo y planeamiento de caché si está disponible.**
- **Utilizar conexiones persistentes o "*pooling software*":** Esto es debido a que en PostgreSQL las nuevas conexiones son costosas y establecer cientos de nuevas de ellas en pocos segundos puede traer como consecuencia la caída del sistema. Un software de agrupación de conexiones actúa como intermediario entre un grupo de servidores web y el servidor de PostgreSQL, de manera que los servidores de conectan al software y este al servidor PostgreSQL, el cual interpreta esto como una sola conexión, por lo que se nota una mejora en el rendimiento, ya que muchas conexiones pueden ralentizar el servidor de PostgreSQL.

Figura # 8 Descripción del software de agrupación de conexiones

2.1.3 Optimizar las consultas SQL

El diseño de las consultas es otro aspecto a tener en cuenta debido a algunas particularidades de PostgreSQL, las cuales deben ser manejadas por el equipo de desarrollo. Para hacer esto de una manera adecuada es necesario conocer cuál es el ciclo de vida y cómo se ejecuta una consulta en PostgreSQL.

Ciclo de Vida:

- **Transmisión del texto de la consulta al *backend* de PostgreSQL:** En este paso no hay mucho que se pueda mejorar, aunque si se tienen consultas demasiado largas embebidas en el código de la aplicación y que no pueden ser preparadas en el servidor con antelación, lo mejor es guardarlas en la base de datos como un procedimiento almacenado para minimizar el tiempo de transmisión.
- **Parseo del texto:** Una vez que el texto de la consulta llega a PostgreSQL este la convierte en *tokens*. Escribiendo las consultas como procedimientos de almacenado también minimiza el tiempo invertido en este paso.
- **Planeación de la consulta:** Usualmente este es el paso en que PostgreSQL decide cómo se ejecutará la consulta y se puede mejorar de muchas formas, lo que se explicará más adelante en este epígrafe (Query Planner).
- **Ejecución de la consulta:** La velocidad con que se ejecuta una consulta depende del camino que el planificador de consultas haya decidido para ejecutar la misma, de la configuración de hardware que se disponga y de cómo se haya configurado el servidor de PostgreSQL.
- **Transmisión del resultado:** Aunque este paso no se pueda optimizar mucho hay que tener en cuenta que todos los datos que se transmiten son extraídos desde

discos, por lo que en las consultas se debe tratar de evitar solicitar datos innecesarios.

Algunos de los consejos que se ofrecen a cerca de este tema son:

- **Hacer más con cada consulta:** Los sistemas en PostgreSQL tienen un mejor rendimiento si están implementados con algunas consultas de gran tamaño y no con muchas pequeñas.
- **Hacer más con cada transacción.**
- **Conocer las particularidades de rendimiento entre las sentencias equivalentes del lenguaje:** Después de la versión 7.4 usar la sentencia NOT IN es mejor que NOT EXISTS, evitar múltiples *joins* externos para versiones anteriores a la 8.2, asegurarse que los tipo de las llaves y los índices son iguales, evitar tipo de búsquedas de la forma "ILIKE '%josh%".

Query Planner:

El planificador de consultas de PostgreSQL es quien se encarga de decidir cómo será ejecutada determinada consulta. Esto se debe a que SQL es un lenguaje declarativo donde el programador especifica qué va a devolver la consulta, no cómo hacerlo. En SQL existen muchas equivalencias por lo que una consulta no trivial puede ser escrita en formas diferentes debido a estas posibilidades de equivalencia del lenguaje. En dicho lenguaje por ejemplo, los *joins* pueden ser ejecutados en cualquier orden, al igual que el orden en que se evalúan los predicados y las subconsultas pueden ser transformadas en *joins*, por solo mencionar algunas de las equivalencias, todas basadas en el álgebra relacional. Esto también trae como consecuencia, no solo que una consulta pueda ser escrita de muchas formas, sino que además pueda ser ejecutada de varias maneras. El trabajo fundamental del Planificador de Consultas de PostgreSQL es decidir cuál será la manera más eficiente de ejecutar una consulta y hacerlo en un tiempo lo suficientemente rápido; ya que si sumamos el tiempo que se toma en decidir cuál es la mejor forma de ejecutar una consulta, junto con el tiempo que esta toma, y es mayor que el tiempo que se tomaría el primer camino que pueda aparecer, entonces no se está resolviendo ningún problema.

La forma de representar un plan de consulta es en forma de árbol, donde cada nodo es una operación que tiene que realizar el gestor para ejecutar una consulta, por ejemplo: joins, ordenamiento, búsqueda en disco, etc. Las tuplas que serán mostradas en el resultado de la consulta empiezan a subir desde las hojas del árbol (que casi siempre son las búsquedas en disco) hasta la raíz. Para obtener un resultado, el ejecutor de la consulta solo tiene que pedirle al nodo raíz un resultado y este les pasa la petición a sus hijos y así sucesivamente, siempre y cuando el plan escogido sea bueno, la ejecución de la consulta presentará un buen rendimiento.

El planificador escoge el plan basándose en su costo estimado, para realizar esto el planificador asume que lo que más tiempo toma son las búsquedas en disco y por lo general, toma el plan que menos de estas búsquedas requiera, tomando también como pauta, que las búsquedas aleatorias en disco son mucho más lentas que las secuenciales. Para realizar este estimado el planeador utiliza las estadísticas almacenadas en el catálogo del sistema por las sentencias ANALYZE, los VACUUM o el recolector de estadísticas en caso de que esté activado. Como estos estimados son basados en estadísticas, que además no siempre están actualizadas o completas, el planificador puede realizar una mala elección, no saberlo y seguir trabajando bajo la asunción de que escogió el plan más adecuado para determinada consulta.

Existen parámetros que pueden ser alterados para obligar al planificador de consultas a escoger planes alternativos para determinadas consultas, algunos de estos son:

- ***enable_bitmapscan(boolean)*:** Este parámetro habilita o deshabilita el uso de planes que utilicen el escaneo de mapas de bit. Por defecto su valor es *on*.
- ***enable_hashagg(boolean)*:** Este parámetro habilita o deshabilita el uso de planes que utilicen agregaciones hash. Por defecto su valor es *on*.
- ***enable_hashjoin(boolean)*:** Este parámetro habilita o deshabilita el uso de planes que utilicen hash-join. Por defecto su valor es *on*.
- ***enable_indexscan(boolean)*:** Este parámetro habilita o deshabilita el uso de planes que utilicen el escaneo de índices. Por defecto su valor es *on*.

- ***enable_mergejoin(boolean)*:** Este parámetro habilita o deshabilita el uso de planes que utilicen merge joins. Por defecto su valor es *on*.

- ***enable_nestloop(boolean)*:** Este parámetro habilita o deshabilita el uso de planes que utilicen ciclos anidados. Por defecto su valor es *on*. Aunque este valor puede ser cambiado es imposible suprimir completamente el uso los ciclos anidados, ya que existen situaciones donde es imposible evitar su uso, al situar este parámetro en *off,* simplemente se evita que el planeador de ejecución de consultas de PostgreSQL utilice ciclos anidados para ejecutar una consulta si existe otra vía.

- ***enable_seqscan(boolean)*:** Este parámetro habilita o deshabilita el uso de planes que utilicen búsquedas secuenciales. Por defecto su valor es *on*. Aunque este valor puede ser cambiado es imposible suprimir completamente el uso las búsquedas secuenciales, ya que existen situaciones donde es imposible evitar su uso, al situar este parámetro en *off,* simplemente se evita que el planeador de ejecución de consultas de PostgreSQL utilice búsquedas secuenciales para ejecutar una consulta si existe otra vía.

- ***enable_sort(boolean)*:** Este parámetro habilita o deshabilita el uso de planes que utilicen ordenamientos. Por defecto su valor es *on*. Aunque este valor puede ser cambiado es imposible suprimir completamente el uso los ordenamientos ya que existen situaciones donde es imposible evitar su uso, al situar este parámetro en *off,* simplemente se evita que el planeador de ejecución de consultas de PostgreSQL utilice ordenamientos para ejecutar una consulta si existe otra vía.

Aunque modificar estos parámetros puede ser una buena solución para forzar al planificador de consultas de PostgreSQL a modificar su plan de ejecución, para alguna consulta en específico no se aconseja que se modifiquen los valores por defecto con los que vienen de forma permanente ya que es muy probable que esto traiga consigo más problemas que beneficios. Para mejorar el rendimiento del planificador de ejecución de consulta de forma más permanente es mejor modificar la constantes de costo del planificador, ejecutar ANALYZE con mayor frecuencia, incrementar el valor de predeterminado del parámetro *default_statistics_target* e incrementar la cantidad de estadísticas recogidas para alguna columna en específico.

Constantes de costo del planificador:

Las variables de costo del planificador de consultas de PostgreSQL son variables utilizadas por este para determinar el costo aproximado de los distintos planes de ejecución para una consulta, el plan con menor valor será el escogido. Estas variables no son medidas en ninguna escala, por lo que lo único que interesa es el valor relativo entre ellas, por lo que si todas son multiplicadas o divididas por un mismo factor esto no afectará en ningún modo las decisiones que tomará el planificador. El valor tomado como base es *seq_page_cost* que por defecto viene establecido en 1.0, todos los demás valores están en una escala relativa con respecto a este.

Variar estos parámetros puede traer consigo una mejora notable en el rendimiento del planificador de consultas de PostgreSQL, pero también todo lo contrario por lo que si se van a modificar debe ser después de un profundo estudio y muchos experimentos. Como hasta estos momentos no existe un método bien definido para determinar el valor ideal de estas variables a continuación no se brindara un método para identificar cuáles deben ser estos valores ideales, solamente se proveerá de una forma de cambiar sus valores y que significan cada una de ellos:

- ***seq_page_cost(floating point)*:** Establece el valor de tiempo que se toma el intercambio de una página de disco que es parte de una secuencia de intercambios. El valor por defecto es 1.0.

- ***random_page_cost(floating point)*:** Establece el costo estimado de intercambiar con disco una página que forma parte de una secuencia. El valor por defecto es 4.0. Disminuir este valor con respecto a *seq_page_cost* provocará que el sistema prefiera los escaneos de indexados en lugar de los secuenciales, aumentarlo hará parecer a estos un poco más costosos. Si se cambian los dos valores juntos se está modificando la importancia de acceso a disco relativa al costo de CPU. Estos dos parámetros pueden ser establecidos al mismo valor, que incluso puede ser menor que 1.0 si la base de datos con la que se está trabajando cabe completamente en la memoria RAM, ya que el costo de intercambiar una página con la memoria RAM es mucho menor con el disco duro y también el acceso a páginas que no estén en secuencia no es más lento como ocurre en el caso de los discos duros.

- ***cpu_tuple_cost(floating point)*:** Establece el valor estimado que toma procesar una fila durante una consulta. El valor por defecto es 0.01.
- ***cpu_index_tuple_cost(floating point)*:** Establece el valor estimado que toma procesar una fila de un índice durante un escaneo indexado. El valor por defecto es 0.005.
- ***cpu_operator_cost(floating point)*:** Establece el valor estimado que toma procesar cada operador o función ejecutada durante una consulta. El valor por defecto es 0.0025.

Entender los mecanismos de procesamiento de PostgreSQL es también importante para escribir sentencias SQL. Ya se esté escribiendo una nueva consulta SQL o se esté afinando una consulta con bajo rendimiento en una aplicación que esté en producción, los dos principales puntos a tener en cuenta son el uso de CPU y los procesos de entrada/salida de disco.

Para afinar consultas ya existentes que puedan estar causando problemas de rendimiento, primero se deben encontrar éstas haciendo uso de programas analizadores de log como el *pgfouine,* para luego arreglarlas.

Algunos aspectos que se deben buscar haciendo uso del EXPLAIN ANALIZE para optimizar una consulta que este consumiendo muchos recursos.

- Estimados de conteo de columna incorrectos.
- Escaneos secuenciales.
- Ciclos muy largos.

Leer el **EXPLAIN ANALIZE** es complejo y casi un arte, este es un árbol invertido en el cual se debe tratar de encontrar los problemas en el nivel más profundo.

2.1.4 Optimizar el hardware

A la hora de afinar el hardware hay cuatro aspectos fundamentales que deben ser tenidos en cuenta:

- **CPU:** Cuantos más se tengan mejor, aunque si no se cuenta con abundante presupuesto y las bases de datos del servidor no requieren cálculos complejos, es mejor priorizar otros aspectos como la RAM y los discos. Si se tiene la posibilidad de escoger, lo mejor es escoger CPUs con varios núcleos, cache L2 y arquitectura de 64 bits, ya que permiten una mayor velocidad de acceso y proporcionan un mayor tamaño de la RAM.

- **RAM:** Mientras más se tenga mejor, ya que se podrá contar con una mayor cantidad de cache, por lo que se minimizará el acceso a discos que es cientos de veces más lento que el acceso a RAM. En el caso de los sistemas OLTP, lo que se debe tener en cuenta es tener suficiente RAM para que quepa la base de datos completa, esto debe ser alrededor de dos a tres veces el tamaño en disco de las tablas en el servidor y en los sistemas OLAP y DSS tener suficiente memoria para que quepan los ordenamientos. Para esto se debe conocer principalmente cuál es el mayor *data set* con el que se va a trabajar y cuántos usuarios lo van a hacer.

- **Discos:** Los discos SCSI son los recomendados para tener en un servidor de base de datos, pero son extremadamente caros, por lo que si no se cuenta con un buen presupuesto es mejor utilizar SATA2 que tienen una buena velocidad y son mucho más baratos, por lo que se podrá comprar más y por tanto poder repartir los ficheros de logs y tablas en diferentes discos, evitando así congestiones en un solo disco. Aquí se muestra un árbol de decisión sobre la selección del disco adecuado, en dependencia de las características del sistema, lo que por supuesto está sujeto a cambios y variaciones según el presupuesto con el que se cuente.

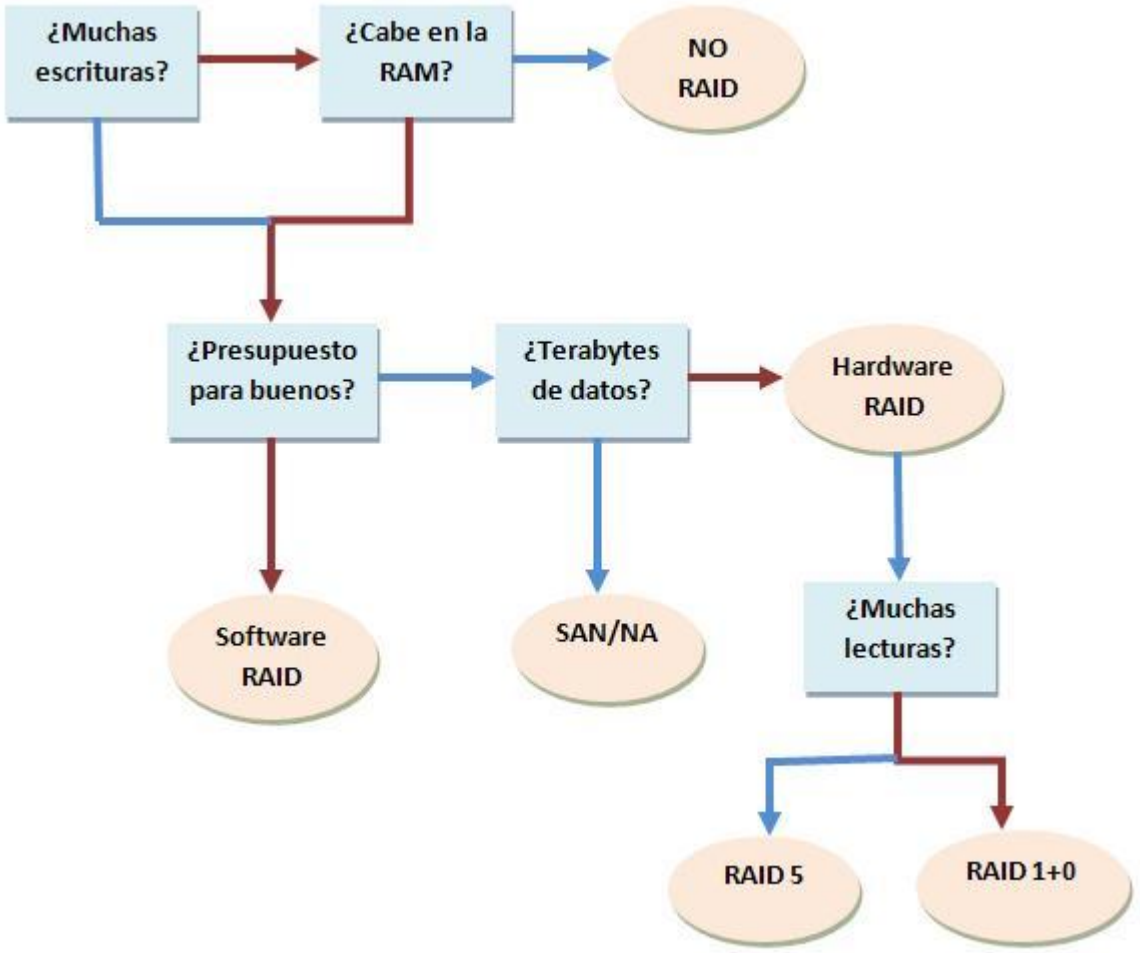


Figura # 9 Árbol de decisión sobre uso de discos

Un sistema de bases de datos que sus datos no quepan en la memoria RAM del servidor, sus momentos de peor rendimiento estarán dados por la velocidad de los discos en que estén almacenados sus datos, por lo que mientras más rápidos sean estos, mejor rendimiento tendrá el sistema. Si no se cuenta con la posibilidad de escoger el disco más rápido que exista en el mercado, pero se tiene cierto margen para escoger teniendo en cuenta el mayor volumen de datos que se pueda escribir o leer en una consulta y haciéndole pruebas de velocidad a los distintos tipos de discos duros existentes, se puede llegar a un equilibrio razonable entre rendimiento y presupuesto.

Para realizar pruebas de rendimiento en discos duro es importante tener en cuenta dos aspectos fundamentales: primero tener la menor cantidad posible de procesos

corriendo en el sistema operativo en el que se va a probar y segundo tener suficiente espacio libre para poder realizar la misma. A continuación se explicará cómo realizar una pequeña prueba de velocidad de acceso a discos duros en un sistema basado en Unix utilizando una herramienta de código abierto bastante simple que se encuentra integrada al sistema operativo utilizado para realizar la misma (Ubuntu 8.10). Esta prueba mide el más simple acceso a disco que existe: una escritura secuencial grande seguida por una lectura secuencial similar. Esta prueba es relevante para PostgreSQL y para los sistemas de bases de datos en general, ya que proporciona una idea de la máxima velocidad que puede tener un escaneo secuencial en una tabla larga del sistema. En el caso de PostgreSQL los escaneos secuenciales reales efectuados a las tablas están alrededor del 30% del máximo arrojado por esta prueba. Para realizar esta prueba lo otro que se necesita conocer es el tamaño de la memoria RAM del sistema. La prueba consta de los siguientes 3 pasos.

1. Se crea un solo fichero que debe tener dos veces el tamaño de la memoria RAM y se sincroniza con el disco, se le pone este tamaño para obtener una velocidad de escritura sostenida con la menor intervención posible de la memoria cache. Como los bloques utilizados por PostgreSQL tienen 8kb un Gigabyte tiene 125000 bloques, por lo que si la memoria del servidor es de 4 GB se deberá ejecutar la siguiente línea en una consola de Linux:

   ```
   time sh –c “dd if=/dev/zero of=disktest bs=8k count =1000000 && sync”
   ```

   Después de ejecutada esta línea se espera el tiempo necesario para que el archivo se copie y se cree y cuando este haya finalizado aparecerá en pantalla unos resultados semejantes a estos:

   ```
   1000000+0 records in
   1000000+0 records out
   8192000000 bytes (8 GB) copied, 147.161 seconds, 55.5 MB/s

   real 4m0.231s
   user 0m0.924s
   sys 0m40.334s
   ```

Donde se puede observar la cantidad de bloques escritos, el tiempo total demorado en realizar la operación y la velocidad del disco entre otras, pero estas son las que interesan. En este ejemplo la velocidad de escritura incluye el la velocidad con la que se escribieron los datos en la memoria cache, pero incluyendo el tiempo de sincronización es realmente menor alrededor de 32.5MB/s (8 GB / 4min).

2. A continuación se escribe otro fichero grande, esta vez de solo el tamaño de la RAM para sacar la información del anterior que pueda estar almacenada todavía en la memoria cache para que la lectura en el próximo paso sea directamente desde el disco duro.

```
dd if=/dev/zero of=ddfile2 bs=8K count=500000
```

3. Por último se lee nuevamente el primer archive que se creó. Como la memoria cache está llena con el segundo archivo esta lectura se realiza directamente desde el disco duro.

```
time dd if=disktest of=/dev/null bs=8k
1000000+0 records in
1000000+0 records out
8192000000 bytes (4 GB) copied, 204.695 seconds, 40.1 MB/s


real 3m24.729s
user 0m1.756s
sys 0m11.951s
```

Esta vez el tiempo dado por la instrucción “dd” es bastante parecido al tiempo real invertido al tiempo real utilizado y la única diferencia es tiempo de almacenamiento en buffer el cual puede ser despreciado.

Utilizando la anterior prueba se puede escoger un disco al alcance del presupuesto disponible y que tenga una velocidad que permita realizar la mayoría de las consultas en un tiempo razonable.

- **Red:** Lo más común y óptimo es que en el servidor de base de datos no se ejecuten ninguna de las aplicaciones clientes que utilizan sus servicios, por lo que todo el tráfico de información circula por la red. Si las aplicaciones son desktop, no hay mucho que se pueda hacer para optimizar la red, pero si se trata de una aplicación web una buena práctica sería que en el servidor de la aplicación web se instalaran dos tarjetas de red: una dedicada a las conexiones con los cliente y otra única y exclusivamente a la conexión con el servidor de base de datos para evitar cuello de botellas o congestiones, debido a la gran cantidad de conexiones clientes que en un determinado momento pueda tener.

En caso de no tener restricciones materiales a la hora de adquirir hardware es evidente que todo sería muy fácil, pero como casi nunca ese es el caso, se debe conocer para cada uno de los tipos de sistema que hay cómo influyen cada una de sus partes en el rendimiento y saber cuál de esos aspectos se debe priorizar por encima de los demás en aras de correr un sistema óptimo, de acuerdo a las necesidades que se tengan.

Otros aspectos a tener en cuenta a la hora de hablar de hardware es que la calidad importa y no todos los CPUs son iguales, o las tarjetas de red o las de RAID, un componente o driver inadecuado o de baja calidad pueden destruir el rendimiento de la aplicación.

2.1.5 Optimizar los índices y los caminos de acceso

Los índices son una estructura de datos utilizada para mejorar e incrementar la velocidad de búsqueda de valores en una base de datos. Estos pueden ser creados utilizando una o más columnas de una tabla y en el caso de las bases de datos relacionales los índices son una copia de una parte de la tabla. Su utilización es un aspecto importante a la hora de hablar de rendimiento de una base de datos y una incorrecta estrategia de indexado puede repercutir negativamente en este. Para crear un índice en PostgreSQL se utiliza la siguiente sentencia:

CREATE INDEX <nombre_del_índice> **ON** <nombre_de_la_tabla> (<nombre_de_los_campos>);

Para eliminar un índice:

DROP INDEX <nombre_del_índice>;

Los índices pueden ser creados o eliminados en cualquier momento y una vez creado el índice, PostgreSQL se encarga de actualizarlo cada vez que la tabla sea modificada y a su vez, el planificador de utilizarlos cuando estime que serán más eficientes que una búsqueda secuencial. Mientras más veces se ejecuten sentencias ANALYZE mejor será la información con la que contará el planificador y por tanto mejor será la utilización de los índices.

PostgreSQL tiene cuatro tipos de índices: B-tree, Hash, GIST y GIN. Cada uno de estos utiliza un algoritmo diferente dependiendo del tipo de consulta. Por defecto la sentencia CREATE INDEX crea un árbol de tipo B-tree debido a que este es el que se utiliza más comúnmente. Según el tipo de consulta que sea el planificador de PostgreSQL evalúa que tipo de árbol utilizar:

- **B-tree:** Consultas de rango e igualdad donde se utilicen los operadores **<**, **<=**, **= >=**, **>** y algunas sentencia equivalentes como **BETWEEN**, **IN** y la condición **IS NULL**.

- **Hash:** Solamente soportan comparaciones de igualdad utilizando el operador **=** y no soportan la condición **IS NULL**, para crear un índice Hash se utiliza la siguiente sentencia:

  CREATE INDEX <nombre_del_índice> **ON** <nombre_de_la_tabla> **USING HASH** (<nombre_de_la_columna>);

- **GIST:** No son un solo tipo de índices, sino una infraestructura donde muchas estrategias pueden ser implementadas por los usuarios, por tanto los operadores utilizados por este varían dependiendo de la estrategia implementada. La distribución estándar de PostgreSQL incluye clases de operadores GIST para varios tipo de datos bidimensionales que soportan consultas indexadas para los operadores **<<**, **&<**, **&>**, **>>**, **<<|**, **&<|**, **|&>**, **|>>**, **&>**, **<&**, **~=**, **&&**.

- **GIN:** Son índices invertidos que pueden manipular valores que contengan más de una llave como arreglos. Como GIST GIN soporta estrategias definidas por lo usuarios puede variar su uso dependiendo de la estrategia implementada. La distribución estándar de PostgreSQL incluye clases de operadores GIN para

arreglos unidimensionales que soportan consultas indexadas para los operadores **<@**, **@>**, **=**, **&&**.

Implementar una correcta estrategia de indexado es algo complicado sobre todo si el que la implementa no es un experto en el tema, pero cualquier administrador de base de datos puede mejorar el rendimiento de sus índices haciendo uso de las posibilidades que ofrece el catálogo y las estadísticas de PostgreSQL en su versión 8.3. Los índices que no se usan usualmente y que no forman parte de una restricción causan demora en las actualizaciones, inserciones y borrado de las tablas a las que están ligados. Además hacen que se retarden las operaciones de VACUUM, resguardo y restauración de la base de datos y por último hacen que el planificador de ejecución de consultas se tome más tiempo en analizar qué camino de ejecución tomará para ejecutar las consultas realizadas sobre esas tablas, pues tiene que decidir si utiliza o no los índices. Por lo que uno de los caminos que se puede tomar es deshacerse de estos índices inactivos.

Para eliminar los índices inactivos lo primero que hay que hacer es identificarlos. Si se dispone de tiempo suficiente y el servidor no se encuentra en un estado crítico, lo mejor es ejecutar la función *pg_stat_reset()* para borrar todas las estadísticas almacenadas por el sistema para la base de datos a la que se esté conectado y luego dejar al sistema correr bajo condiciones normales de explotación durante un ciclo completo del negocio. De esta forma se debe evitar que se tomen como índices en uso algunos que pueden haber sido usados en un pasado lejano, pero que ya no se utilizan. Si las prestaciones que el servidor está brindando son muy bajas, su ciclo completo de negocio se toma demasiado tiempo, o simplemente no se quiere esperar, se puede realizar el proceso de afinamiento de los índices sin borrar las estadísticas, lo que quizás no se obtengan los mismos resultados.

Para encontrar lo índices se puede ejecutar la siguiente consulta:

```
SELECT idstat.relname AS table_name,

indexrelname AS index_name,

idstat.idx_scan AS times_used,

pg_size_pretty(pg_relation_size(idstat.relname)) AS table_size,
```

```
pg_size_pretty(pg_relation_size(indexrelname)) AS index_size,
n_tup_upd + n_tup_ins + n_tup_del as num_writes,
indexdef AS definition
FROM pg_stat_user_indexes AS idstat JOIN pg_indexes ON indexrelname = indexname
JOIN pg_stat_user_tables AS tabstat ON idstat.relname = tabstat.relname
WHERE idstat.idx_scan < 100
AND indexdef !~* 'pk'
ORDER BY times_used DESC, idstat.relname
```

Esta consulta es para una base de datos con solo el esquema *public*, en caso de tener más de un esquema se debe usar la siguiente versión de la misma consulta:

```
SELECT
idstat.relname AS table_name,
indexrelname AS index_name,
idstat.idx_scan AS times_used,
pg_size_pretty(pg_relation_size(idstat.schemaname||'.'||idstat.relname)) AS table_size,
pg_size_pretty(pg_relation_size(idstat.schemaname||'.'||indexrelname)) AS index_size,
n_tup_upd + n_tup_ins + n_tup_del as num_writes,
indexdef AS definition
FROM pg_stat_user_indexes AS idstat JOIN pg_indexes ON indexrelname = indexname
JOIN pg_stat_user_tables AS tabstat ON idstat.relname = tabstat.relname
WHERE idstat.idx_scan < 100
AND indexdef !~* 'unique'
```

ORDER BY times_used **DESC**, idstat.relname

2.1.6 Optimizar el fichero postgresql.conf

En el fichero postgresql.conf se encuentran todos los parámetros que determinan el entorno de ejecución del servidor de PostgreSQL, aunque algunos de ellos se pueden fijar mediante las opciones de arranque del servidor. Todos estos parámetros tienen valores por defecto, algunos vinculados al código fuente en */include/pg_config.h* y otros como una opción de configuración en tiempo de compilación.

La configuración por defecto que trae PostgreSQL está diseñada para lograr compatibilidad con casi cualquier configuración de hardware en la cual pueda ser instalado y no para lograr un buen rendimiento, por lo que es altamente probable que con la configuración por defecto un sistema no trabaje ni medianamente bien. Los valores de los parámetros que no se encuentren especificados en el fichero tienen como valor el que le asigna por defecto PostgreSQL y pueden ser consultados en la tabla del catálogo de del gestor *pg_setting.* Cualquier valor puede ser cambiado ya sea con el servidor en marcha utilizando el comando SET o cambiando el valor directamente en el fichero y recargando la configuración con *pg_ctl reload*, aunque hay algunos de estos parámetros que necesitan que el servidor sea reiniciado.

De estas dos formas de variar los parámetros de configuración la más segura y recomendada para optimizar el rendimiento del servidor es la segunda, debido a que hay parámetros que con el uso de la instrucción SET se varían solo para la conexión que ejecuta la instrucción y otros que incluso varían para la transacción que ejecuta la instrucción.

La configuración de los parámetros del fichero postgresql.conf requiere conocer los tipos de datos que aceptan cada uno de ellos. Los tipos que existen son los siguientes:

- **Boolean:** true, false, on, off, yes, no.
- **Integer:** números enteros, 1234.
- **Float Decimal values:** 12.34.

- **Memory/Disk integer:** 1234 o “unidad de computadoras” 512MB, 10GB. Aquí es recomendable evitar los enteros, pues se tiene que conocer entonces la unidad subyacente para entender qué quiere decir.
- **Time:** d, m, s, 30s. No es necesario especificar la unidad, pero esta práctica no es aconsejable.
- **String, o texto:** pg_log.
- **Enums:** (‘WARNING, ‘ERROR’).
- **Listas:** user, public, test.

Las líneas que comiencen con “#” están comentadas, por lo que los valores que estén dentro de ellas no tendrán efecto a menos que se borre el símbolo. Para bases de datos nuevas esto significa que el parámetro tiene asignado su valor por defecto, pero para sistemas que se encuentren ya en funcionamiento esto no es necesariamente cierto, porque en versiones anteriores a la 8.3 comentar un valor en el fichero no restaura el parámetro a su valor por defecto, e incluso en esta versión y la inmediata superior a esta (que se encuentra a punto de salir) los cambios no tienen efecto hasta que no se recarguen los parámetro o se reinicie el servidor. Por último hay que tener en cuenta que si el mismo parámetro está especificado varias veces, el último que se escribe es el que PostgreSQL tomará como válido. A continuación se describen algunos de estos parámetros:

- **Listen addresses:** Por defecto PostgreSQL solo responde a las conexiones desde el mismo servidor. Si se desea que el servidor sea accesible para otros por el protocolo de red estándar TCP/IP, se debe cambiar el valor por defecto que trae el parámetro *listen_adresses*. El método más comúnmente usado es el de establecer el valor de este parámetro de la siguiente forma: *listen_adresses* = ‘*‘ y luego controlar los rangos de dirección en el archivo *pg_hba.conf*.
- **Max connections:** Este parámetro define exactamente lo que su nombre indica, establecer el número máximo de conexiones clientes permitidas en nuestro servidor. Este valor es muy importante para algunos de los otros parámetros que se explicarán a continuación, particularmente *work_mem*, porque hay algunos recursos

de memoria que son asignados por conexión, por lo que el máximo número de clientes que pueden estar conectados es un aspecto a tener en cuenta a la hora de asignar dichos recursos. Generalmente, PostgreSQL con un buen hardware puede soportar algunos cientos de conexiones sin afectar el rendimiento, pero si se necesitan miles se debe usar *connection pooling software,* que en español se puede traducir este término como: software de agrupación de conexiones, para reducir la sobrecarga en el servidor por exceso de peticiones.

- **Shared buffers:** PostgreSQL no intercambia información directamente con el disco. En lugar de esto los datos son solicitados al *shared_buffer_cache*, entonces los *backends* del gestor leen o escriben en estos bloques y finalmente los envían a los discos. Los *backends* que necesitan acceder a las tablas primero buscan los bloques que necesitan en esta caché, si ya están ahí se puede continuar el procesamiento inmediatamente, sino el sistema operativo hace una petición para cargar los bloques. Los bloques son cargados de la caché del kernel o desde un disco duro. Estas operaciones pueden ser costosas.

  La configuración por defecto de PostgreSQL asigna 1000 *shared buffers* donde cada buffer tiene 8 kilobytes. Incrementar el número de bloques hace más probable que la información solicitada por los *backends* sea encontrada sin necesidad de recurrir a costosas solicitudes del sistema operativo. El cambio puede ser hecho con una línea de comando ***postmaster*****,** cambiando el valor "*shared_buffers*" en el fichero de configuración postgresql.conf.

  La configuración de los parámetros de los buffers compartidos determina cuánta memoria es dedicada a PostgreSQL para este utilizarla para el cacheo de los datos. Los valores que trae por defecto son bajos porque algunas plataformas (como las versiones más viejas de Solaris y SGI) si tienen valores muy grandes requieren acciones invasivas como recompilar el kernel. Si el sistema tiene 1 GB o más de RAM, un valor razonable para comenzar de los buffers compartidos podría ser ¼ de la memoria del sistema. Si se cuenta con menos RAM se debe analizar cuidadosamente cuánta RAM necesita el sistema operativo. En estos casos es común que el valor esté alrededor del 15% de la memoria con que se cuenta.

- **Effective cache size:** Este valor debe ser fijado basándose en un estimado de cuánta memoria caché queda disponible en el sistema operativo después de tener en cuenta cuánta memoria es usada por:

  - El sistema operativo.
  - Cuánta para el propio PostgreSQL
  - El resto de las aplicaciones que puedan estar corriendo en el sistema.

  Esto es una guía de cómo la memoria que espera esté disponible en el búfer de caché de sistema operativo, no una asignación. Este valor es usado únicamente por el planificador de consulta de PostgreSQL para saber si es examinar cuáles son los planes que se espera caben en la memoria RAM o no. Si su valor es demasiado bajo, puede que los índices no se utilicen para ejecutar consultas de la manera que se esperaba.

  Establecer el effective_cache_size a ½ de la memoria total es un valor conservador normal y ¾ de la memoria es un poco más agresivo pero sigue siendo un valor razonable. Se puede lograr un mejor estimado observando las estadísticas del sistema operativo en el cual está corriendo PostgreSQL. En los sistemas basados en UNIX se puede usar el *top* o el *free* para retornar los valores libres de la caché. En Windows se puede ver el tamaño de la caché del sistema en el Administrador de Tareas en la pestaña de rendimiento.

- **Sort memory:** Este parámetro establece la máxima cantidad de memoria que una conexión a la base de datos puede usar para realizar procesos de ordenamiento. Si una consulta tiene sentencias **ORDER BY** o **GROUP BY** y requiere ordenar una gran cantidad de datos, incrementar este parámetro ayuda a mejorar el rendimiento, pero hay que tener cuidado porque este parámetro es para cada ordenamiento de cada conexión, por lo que no se debe aumentar mucho su valor, sobre todo en bases de datos con muchos usuarios.

- **Fsync y WAL files:** Este parámetro establece si se va a escribir o no en disco tan pronto como se haga la transacción, lo cual es hecho a través del Write Ahead Logging (WAL). Si se tiene suficiente confianza en el hardware, en el abastecimiento

de energía eléctrica y en la batería de respaldo, entonces se puede establecer este parámetro en NO y se logra aumentar la velocidad de escritura en disco, pero esto significa que si sucede cualquier imprevisto y se apaga el servidor, es muy probable que se tenga que restaurar la base de datos con el último respaldo que se le haya hecho, pues se perdería un gran volumen de datos. Si no se puede correr ese riesgo de todas formas se puede tener la protección de WAL y un buen rendimiento moviendo el directorio pg_xlog para un disco diferente que los archivos principales de la base de datos. En bases de datos con mucha actividad de escritura WAL se debe ubicar en su propio disco para asegurar una alta velocidad de acceso.

- **Ramdon page cost.**

- **Vacumm mem:** Este parámetro establece la cantidad de memoria que puede utilizar el proceso de VACUUM. Normalmente VACUUM es un proceso intensivo de lectura/escritura de disco, pero aumentando este parámetro aumentará la velocidad del proceso mencionado porque le permite a PostgreSQL copiar mayores bloques a la memoria. Aunque no se debe poner muy grande tampoco, porque es memoria que se estaría restando a los procesos normales de la base de datos y por tanto afectaría el rendimiento de la misma. Para la mayoría de los sistemas un valor entre 16 MB y 32 MB debe ser suficiente.

- **Max fsm pages:** PostgreSQL almacena cuales son los espacios libres de cada una de sus páginas de datos, información de gran utilidad para el proceso de VACUUM, pues le dice qué páginas buscar para liberar espacio. Si se tiene una base de datos donde se borra y se actualiza con frecuencia se van a generar muchas tuplas muertas debido al sistema MVCC de PostgreSQL. El espacio ocupado por estas tuplas muertas puede ser liberado con VACUUM, a no ser que exista más espacio sin usar que no esté guardado en el Free Space Map (FSM), en cuyo caso se debe ejecutar el muy costoso FULL VACUUM. Extendiendo la memoria del FSM esto se puede remediar ya que así se logrará cubrir todas las tuplas muertas.

  La mejor forma de establecer este valor es interactivamente. Lo primero es establecer el intervalo de tiempo que va a haber entre un VACUUM y otro, mientras más actividad de borrado y actualización haya en la base de datos menor debe ser,

luego se realiza un VACUUM y se deja al sistema correr bajo condiciones normales y se espera el tiempo que se estableció entre un VACUUM y otro, se realiza un VACUUM VERBOSE ANALYZE, y se salva la salida para un archivo, por último se calcula el total máximo de páginas necesarias entre dos VACUUM basados en la salida del archivo y se establece este número como valor del parámetro max_fsm_pages en el archivo postgresql.conf. Se debe tener en cuenta que cada página de FSM utiliza 6 bytes de RAM de encabezado por lo que incrementar demasiado este valor en sistemas con baja RAM puede ser contraproducente.

- **Max fsm relations:** Este parámetro indica la máxima cantidad de relaciones (tablas) que serán vigiladas por el FSM, algo que es tan sencillo como conocer la cantidad de tablas con las que cuenta el servidor.

- **Logging:** Hay muchas cosas que pueden ser registradas por los logs de PostgreSQL pero para que se guarde toda esta información es necesario configurarlos. Algunos de los parámetros relativos a que se va o no a registrar en los logs y donde son los siguientes:

    - **Log_destination & log_directory (log_filename):** Lo que se escribe en estas opciones no es tan importante como las pistas que puede sobre donde el servidor está registrando los logs.

    - **Log_min_error_statement:** Este parámetro es recomendable mantenerlo en ON, para así poder ver las consultas SQL que pueden causar errores.

    - **Log_min_duration_statement:** No es recomendable tenerlo activado todo el tiempo ya que puede causar bajo rendimiento en el sistema si hay muchas peticiones al mismo tiempo, pero es recomendable activarlo sistemáticamente para revisar consultas lentas que puedan ser optimizadas.

    - **Log_statement:** La activación de este log en servidores en explotación provoca que se degrade el rendimiento del mismo, pero es muy útil a la hora de detectar cambios en la base de datos que puedan ser realizados sin la debida autorización.

2.1.7 Optimizar la estructura física

Cada tabla de una base de datos tiene un archivo primario de encabezado donde la mayoría de los datos son guardados. Si la tabla tiene alguna columna donde en algún momento determinado se almacenen valores muy grandes, también se crea un fichero TOAST asociado a dicha tabla con el objetivo de guardar estos valores que no caben adecuadamente en el archivo de la tabla principal, en dichas tablas también existirá un índice. Cada tabla o índice de una base de datos en PostgreSQL será almacenada en un fichero por separado, incluso en ocasiones en más de uno, pues si la tabla o el índice exceden de 1 Gigabyte otro fichero será creado por el gestor donde se almacenarán los nuevos valores de la tabla o del índice.

Para monitorear el espacio utilizado en disco por los archivos de PostgreSQL se pueden usar varias formas, una de ellas es utilizando las funciones presentadas en la siguiente tabla:

Nombre	Tipo que devuelve	Descripción
pg_column_size(any)	int	Número de bytes usados para almacenar un valor en particular (posiblemente comprimido).
pg_database_size(oid)	bigint	Espacio en disco utilizado por la base de datos con el OID especificado.
pg_databse_size(name)	bigint	Espacio en disco utilizado por la base de datos con el nombre especificado.
pg_relation_size(oid)	bigint	Espacio en disco utilizado por la tabla o índice con el OID especificado
pg_relation_size(name)	bigint	Espacio en disco utilizado por la tabla o índice con el nombre especificado. El nombre de la tabla puede ser especificado junto con

		el nombre del esquema al que pertenece.
pg_size_pretty(bigint)	text	Convierte un tamaño dado en bytes a una forma más legible, dándole formato que incluye unidades tales como Gigabyte o kilobyte.
pg_tablespace_size(oid)	bigint	Espacio en disco utilizado por el tablespace con el OID especificado.
pg_tablespace_size(name)	bigint	Espacio en disco utilizado por el tablespace con el nombre especificado.
pg_total_relation_size(oid)	bigint	Espacio total en disco utilizado por la tabla con el OID especificado, incluyendo índices y tablas TOAST.
pg_total_relation_size(name)	bigint	Espacio total en disco utilizado por la tabla con el nombre especificado, incluyendo índices y tablas TOAST.

Tabla # 3 Funciones para conocer espacio utilizado

Una de las principales tareas de un administrador de base de datos es evitar que se llenen los discos duros que utiliza su servidor de base de datos, pues si esto sucede los datos no se corrompen pero sí puede ocurrir que no se pueda realizar ninguna actividad que incluya el uso de este disco por el servidor. Si el disco que almacena los ficheros WAL se llena el servidor puede entrar en pánico y por tanto este puede caerse.

Durante la actividad de una base de datos los cabezales de los discos se mueven considerablemente y si se realizan muchas peticiones de lectura/escritura el disco se puede saturar causando un bajo rendimiento. Una solución para los discos saturados puede ser mover ficheros de datos para otros discos, no para otros directorios dentro del mismo disco, ni siquiera para otras particiones ya que éstas usan los mismos cabezales. Algunas soluciones para esto pueden ser:

- Mover bases de datos completas.
- Mover tablas.
- Mover índices.
- Mover joins.
- Mover ficheros de logs.

Uso de Tablespaces:

Los tablespaces en PostgreSQL le permiten a los administradores de base de datos definir en qué lugar del sistema de archivos del servidor serán almacenados los diferentes objetos de la misma, ya díganse tablas, índices o bases de datos. Los tablespaces son muy útiles a la hora de administrar servidores PostgreSQL ya que en caso de que se llene el disco duro o la partición donde fue creado el clúster y no pueda ser extendido, se puede crear un tablespace en un disco duro o partición diferente y usarlo para almacenar las nuevas entradas hasta que el sistema pueda ser debidamente reconfigurado, o usarse como solución permanente.

Los tablespaces también pueden ser utilizados para mejorar el rendimiento de una base de datos, por ejemplo: si un índice grande que es utilizado con frecuencia, se sitúa en un disco extremadamente rápido y al que se acceda poco para realizar otras solicitudes, esto aumentaría considerablemente el tiempo de respuesta de la misma. Siguiendo este mismo principio, aquellos datos que son raramente consultados o actualizados y que solamente se almacenan con propósitos históricos, pueden ser ubicados en discos más lentos y por tanto menos costosos, ahorrando así presupuesto que puede ser invertido en mejorar otros aspectos del servidor.

Para crear un tablespace se utiliza el comando *CREATE TABLESPACE*, como se muestra a continuación:

CREATE TABLESPACE example **LOCATION** '/dev/sda5/postgresql/';

A la hora de especificar el directorio donde se va a crear el nuevo tablespace se debe tener en cuenta que este debe ser un directorio vacío, el cual pertenezca al usuario del

sistema PostgreSQL. Todos los objetos creados en ese tablespace serán almacenados en ficheros que se crearán en este directorio. La creación de un nuevo tablespace debe ser realizada por un superusuario del sistema, pero después que es creado se les puede dar permisos a usuarios comunes de la base de datos para que hagan uso de estos, proporcionándoles el privilegio CREATE. Tablas, índices y bases de datos enteras pueden ser asignadas a un tablespace en particular. Para hacer esto un usuario con el privilegio CREATE en el tablespace debe pasar el nombre del tablespace a la hora de crear el objeto en cuestión, por ejemplo:

```
CREATE TABLE table (id int) TABLESPACE example;
```

Alternativamente se puede establecer un DEFAULT TABLESPACE para que todos los nuevos objetos creados en el clúster y a los cuales no se les especifique un tablespace a la hora de ser creados sean almacenados en éste. Para definir el DEFAULT TABLESPACE se usa la siguiente instrucción.

```
SET default_tablespace = example;
```

También existen los tablespaces temporales que son utilizados para crear objetos temporales (tablas temporales e índices en tablas temporales) cuando una instrucción CREATE no especifica explícitamente un tablespace. Los archivos temporales para ordenar data sets muy grandes también son creados en estos tablespaces. Estos tablespaces temporales son almacenados en una lista y cada vez que se crea un objeto temporal, un miembro de esta lista es escogido de forma aleatoria para ser utilizado.

Un tablespace puede ser utilizado por varias bases de datos, tablas e índices y para ser borrado tiene que estar completamente vacío. Para borrar un tablespace se utiliza la siguiente instrucción:

```
DROP TABLESPACE example;
```

Si se desea conocer todos los tablespaces creados, se puede utilizar la siguiente consulta SQL:

```
SELECT spcname FROM pg_tablespace;
```

Cuando se inicializa el servidor PostgreSQL por primera vez, dos tablespaces son creados por defecto:

1. **pg_global:** Es donde se almacenan las tablas del catálogo del sistema que son compartidas por todas las bases de datos.

2. **pg_default**: Es donde se almacenan la bases de datos template1 y template0 y que por tanto será el tablespace por defecto para todas las demás bases de datos creadas en el clúster, a no ser que se especifique lo contrario.

En el directorio $PGDATA/pg_tblspc se encuentran especificados vínculos a cada uno de los tablespaces que no han sido creados por el sistema y aunque no se debe realizar esta operación a menudo, pues si no se hace de forma correcta puede ocasionar que el servidor tenga que ser totalmente reconfigurado y por tanto la perdida todos los datos a partir del último backup realizado, se pueden modificar estos links para mover los tablespaces a otros discos o particiones. Esta tarea debe ser realizada con el servidor detenido y después de reiniciado se debe actualizar la dirección del vínculo en la tabla *pg_tablespace* para que el catálogo esté actualizado correctamente.

Otras opciones pueden ser el uso de discos RAID para separar un mismo sistema de ficheros en varios discos. Usar discos duros de espejo puede ralentizar los procesos de escritura en la base de datos, pero acelera las lecturas porque los datos pueden ser leídos de cualquiera de los discos.

2.1.8 Optimizar el sistema operativo

A pesar de todas las mejoras que se puedan introducir en el diseño de las reglas del negocio, de los datos, de la aplicación, en la configuración del gestor de bases de datos o el sistema de archivos, si el sistema operativo sobre el que está corriendo nuestro servidor no está apropiadamente configurado será casi imposible que la base de datos que se desea optimizar presente un buen rendimiento.

Algunos de los aspectos que se deben tener en cuenta a la hora de afinar un Sistema Operativo son:

- Sistema de archivos usado.

- Memoria RAM compartida.
- Versión.
- Problemas de seguridad y parches.
- Otras aplicaciones: En este punto se debe tratar la posibilidad de parar servicios del sistema operativo que no sean necesarios en un momento determinado. Esto debe mejorar el rendimiento del sistema, pues deja libre una porción de memoria que puede ser usada por otros procesos y aplicaciones.

  Para cubrir el mayor número de situaciones posibles Ubuntu GNU/Linux inicia toda una serie de servicios que en algunos casos no son necesarios. Si son deshabilitados los que no se necesitan no estarán durmiendo y consumiendo memoria innecesariamente. Existen otros programas y formas de evitar iniciar servicios, por ejemplo: el *update-rc.d* o el programa *boot-Up Manager*. A continuación se recomienda una forma de solucionar este problema que es simple, efectiva y también ha sido la recomendada por http://www.ubuntuguide.org:

  - Abrir una terminal e ir al directorio /etc/init.d: “cd etc/init.d/”.
  - Ver qué servicios puede ejecutar el ordenador: “ls” (son los que aparecen en verde).
  - Denegar el permiso de ejecución para los que no se quiere que arranquen: “sudo chmod -x nombre_del_servicio”. Por ejemplo, si no se usa fetchmail, se teclea “sudo chmod -x fetchmail”.
  - En caso de alguna equivocación al desactivar algún programa que no era el deseado, siempre se puede habilitar nuevamente el servicio haciendo: “sudo chmod +x nombre_del_servicio”.
  - Sugerencias de servicios que normalmente no se usan en un ordenador de escritorio:
    - **ntpdate:** Actualiza el reloj del sistema sincronizándolo cada vez que se reinicia.

- **ntpdate:** Actualiza el reloj del sistema sincronizándolo cada vez que se reinicia.
- **pcmcia:** Sólo se usa con portátiles que tengan tarjetas PCMCIA.
- **ppp:** Protocolo punto a punto. Sólo se utiliza si es usado un módem para la conexión a Internet.
- **powernowd:** En teoría lo usan los procesadores AMD para gestionar el uso de la energía.
- **rsync:** Utilidad para transferir archivos para hacer copias o mantener un espejo sincronizado.
- **fetchmail:** Recoge y reenvía correo y actúa como pasarela hacia el servicio smtp.
- **postfix:** Agente de transferencia de correo, parecido a sendmail.

Además, al ser cada caso distinto se recomienda que antes de evitar que arranque un servicio se sepa para que sirve: “man nombre_del_servicio” o bien buscar mediante otras fuentes más información sobre el mismo.
Debe tenerse cuidado si se desactiva algún servicio que usen otras aplicaciones. Por ejemplo: Gnome usa cupsys, demonio de impresión. Si se hace desde Gnome una llamada al mismo y no está ejecutándose, el sistema se vuelve inestable. Para evitar esto, hay que modificar, desde el menú Sistema, Preferencias, Sesiones, los demonios del escritorio que se inician en el arranque (en este caso se quitaría del arranque el proceso de Gnome relacionado con cupsys).

Conclusiones

En el capítulo se describieron los pasos propuestos para realizar la optimización en servidores de bases de datos en PostgreSQL. Se detallaron cada uno de estos, así como se presentaron un grupo de consultas de gran utilidad para conocer la actividad de la base de datos y a partir de esto tomar decisiones en cuanto al rendimiento del servidor. Se evidenció la importancia de aplicar esta guía, puesto que un buen rendimiento en el servidor es de vital importancia para el buen desempeño de las aplicaciones, por tanto se concluye en que:

- En los servidores de bases de datos de PostgreSQL existe gran cantidad de aspectos que son posibles de optimizar.
- Se ha llegado a fondo en cada una de las actividades de la solución propuesta en comparación con otras propuestas realizadas anteriormente por otros autores.
- Es posible informar al administrador de la base de datos de la actividad de esta, razón por la cual se han incluido consultas SQL en la solución que faciliten las tareas de optimización.
- Es complejo realizar la optimización de todos los parámetros de una base de datos, por lo que no siempre se podrá llevar a cabo en su totalidad.

Capítulo

3

Validación de los resultados

En este capítulo se muestran los resultados de la aplicación y la validación de la guía en el Centro de Tecnologías de Almacenamiento y Análisis de Datos (CENTALAD) como parte de los servicios que se ofrecen en este centro.

Se presentan los resultados obtenidos tras aplicar los pasos de la guía en varios de los aspectos que propone esta. Se aplicó en algunas de las bases de datos que se encuentran habilitadas en este centro, no siendo posible probar en su totalidad la solución, puesto que no se encuentran creadas todas las condiciones necesarias para ellos, problema que viene dado principalmente en la limitación de recursos.

3.1 Optimización del diseño de los datos y la aplicación

En el proyecto PATDSI del Centro de Tecnologías y Almacenamiento y Análisis de Datos (CENTALAD), específicamente en el módulo Generador de Reportes se cuenta con consultas embebidas en el código PHP las cuales fueron analizadas en búsqueda de posibles mejoras en su diseño. En este caso específico se trabajó respecto a algunas de las particularidades que tiene PostgreSQL y las ventajas que se podían obtener de estas. En algunas de las consultas que se emplean en los servidores de prueba, así como en muchos de los existentes, se hace referencia al uso del NOT LIKE, que por defecto este tipo de operaciones son realizadas con símbolos que pueden ralentizar el tiempo de ejecución de la consulta. Sin embargo, estos símbolos son sustituidos por NOT LIKE, así como la inserción del símbolo % detrás del *pg_temp* y el *pg_toast* y se obtiene un resultado con mejor rendimiento respecto al anterior, evidenciando así un menor costo. Esto se puede evidenciar a través de las siguientes imágenes tomadas en la ejecución de esta técnica mientras se aplicaba.

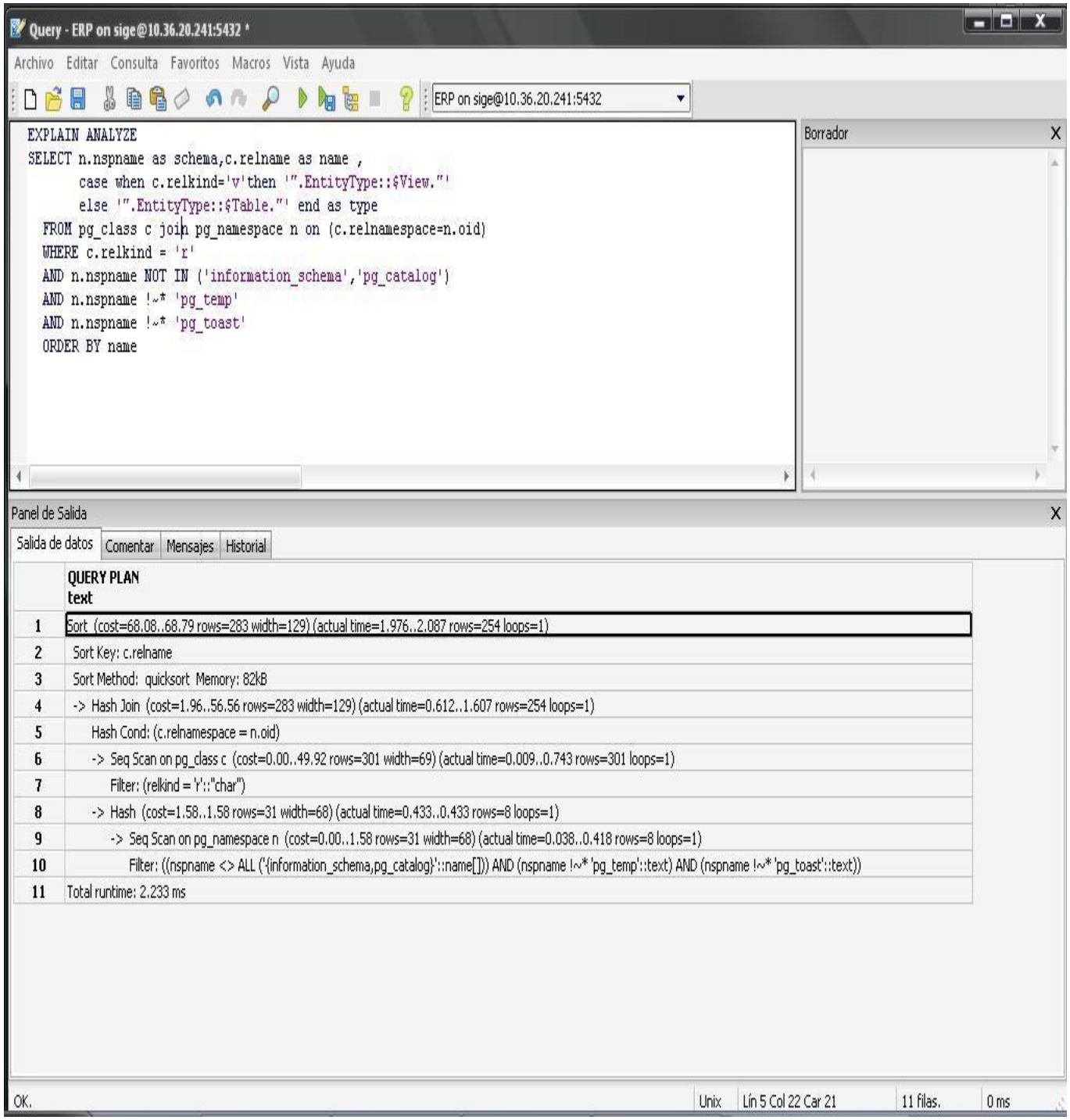


Figura # 10 Sin usar NOT LIKE

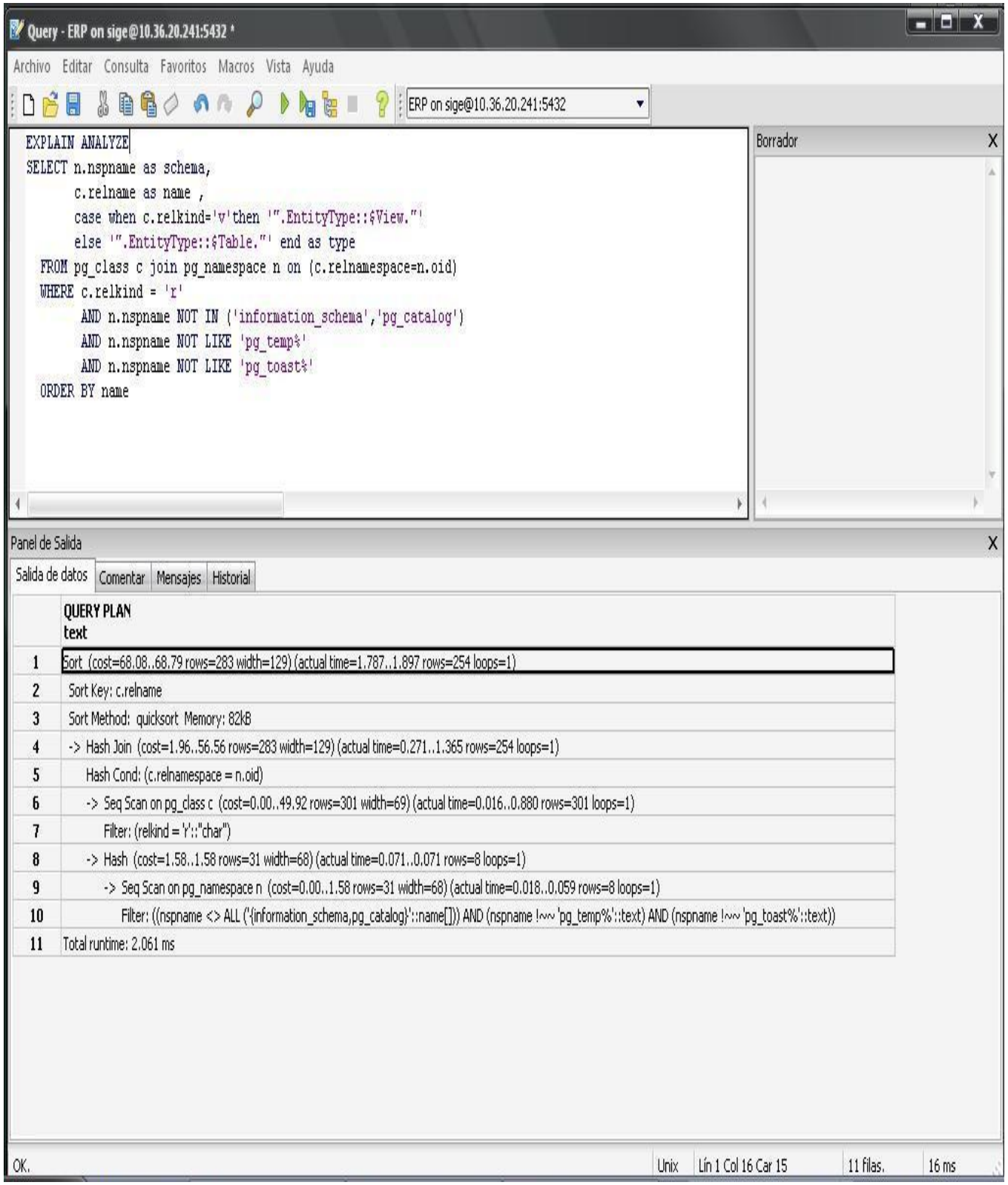

Query - ERP on sige@10.36.20.241:5432 *
Archivo Editar Consulta Favoritos Macros Vista Ayuda
ERP on sige@10.36.20.241:5432
EXPLAIN ANALYZE
SELECT n.nspname as schema,
c.relname as name ,
case when c.relkind='v'then '".EntityType::$View."'
else '".EntityType::$Table."' end as type
FROM pg_class c join pg_namespace n on (c.relnamespace=n.oid)
WHERE c.relkind = 'r'
AND n.nspname NOT IN ('information_schema','pg_catalog')
AND n.nspname NOT LIKE 'pg_temp%'
AND n.nspname NOT LIKE 'pg_toast%'
ORDER BY name
Borrador
Panel de Salida
Salida de datos
Comentar
Mensajes
Historial
QUERY PLAN
text
1 Sort (cost=68.08..68.79 rows=283 width=129) (actual time=1.787..1.897 rows=254 loops=1)
2 Sort Key: c.relname
3 Sort Method: quicksort Memory: 82kB
4 -> Hash Join (cost=1.96..56.56 rows=283 width=129) (actual time=0.271..1.365 rows=254 loops=1)
5 Hash Cond: (c.relnamespace = n.oid)
6 -> Seq Scan on pg_class c (cost=0.00..49.92 rows=301 width=69) (actual time=0.016..0.880 rows=301 loops=1)
7 Filter: (relkind = 'r'::"char")
8 -> Hash (cost=1.58..1.58 rows=31 width=68) (actual time=0.071..0.071 rows=8 loops=1)
9 -> Seq Scan on pg_namespace n (cost=0.00..1.58 rows=31 width=68) (actual time=0.018..0.059 rows=8 loops=1)
10 Filter: ((nspname <> ALL ('{information_schema,pg_catalog}'::name[])) AND (nspname !~~ 'pg_temp%'::text) AND (nspname !~~ 'pg_toast%'::text))
11 Total runtime: 2.061 ms
OK.
Unix
Lín 1 Col 16 Car 15
11 filas.
16 ms

Figura # 11 Usando NOT LIKE

Como se puede observar en las dos figuras el tiempo de ejecución de la consulta antes de ser modificada es 160 ms menor que el de la misma después de realizados los cambios anteriormente planteados. Por un momento se puede pensar que 160 ms es poco tiempo, pero cuando existen 50 o 100 usuarios realizando la misma operación esta pequeña diferencia repercute grandemente en el rendimiento de la aplicación.

3.2 Optimización de consultas SQL

Luego de ser analizado el diseño de las consultas del módulo Generador de Reportes se procedió a analizarlas desde el punto de vista de cómo se ejecutan las mismas. Otra de las pruebas realizadas con la guía en su puesta en práctica, fueron respecto al analizador de consultas de PostgreSQL. Lo primero en este caso, cuando se quiere optimizar la ejecución de una consulta, sería observar cuál es el plan de ejecución que tiene PostgreSQL por defecto para esta consulta. Se debe analizar este árbol y a partir de aquí identificar mejoras o modificaciones que se le puede hacer para mejorar el tiempo de ejecución de la consulta.

Un ejemplo de ello es verificar qué parámetros o sentencias pueden ser desactivados y analizar cómo influye su valor en la ejecución de la consulta. Algunos de estos parámetros, que son por defecto ejecutados, son los *sort* y los *nestedloop*. El primero quiere decir que el plan de consultas realizará muchos ordenamientos a la hora de ejecutar la consulta, lo que no es siempre necesario, por lo que se puede deshabilitar o poner en falso este valor y de este modo solo realizaría ordenamientos en el caso de que no exista otra solución alterna posible. El segundo se refiera a los ciclos anidados, que se comporta similar al caso anterior. Esto ocurre porque estas sentencias determinan un camino de la consulta, existiendo otros tantos que pueden ser usados para su ejecución. Por lo que se recomienda con esta práctica es no hacer uso en algunos casos de estas sentencias, siempre y cuando exista otro camino que sea menos costoso. A continuación se muestran dos imágenes donde se muestran los planes de ejecución para una misma consulta. En la primera está activado el *nestedloop* (como ocurre por defecto) y en la segunda está desactivado. Se evidencia la diferencia en el tiempo de ejecución de estas, así como el plan de ejecución de cada una de ellas.

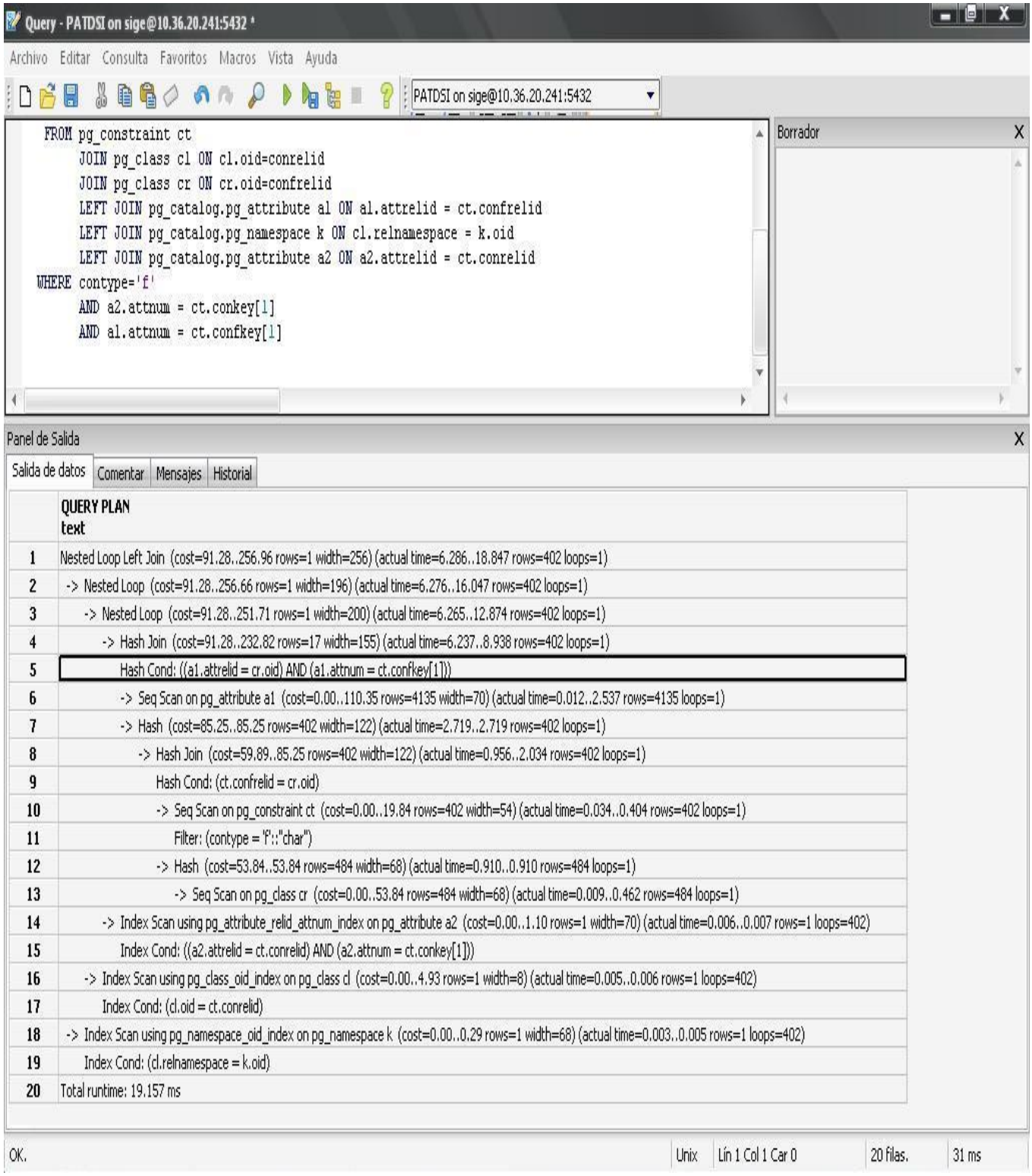


Figura # 12 Con *nestedloop* activado

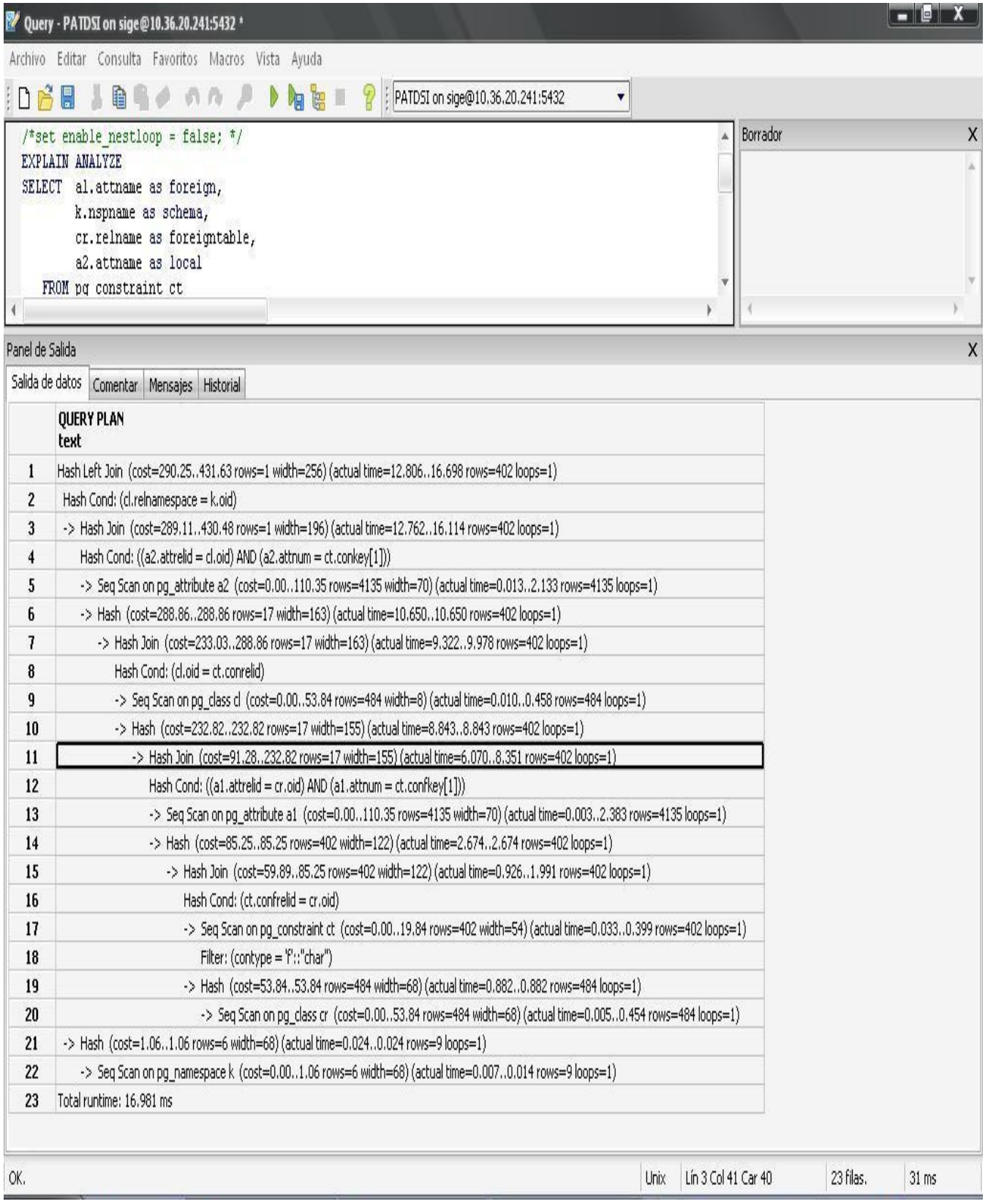

Query - PATDSI on sige@10.36.20.241:5432 *
Archivo Editar Consulta Favoritos Macros Vista Ayuda
PATDSI on sige@10.36.20.241:5432
/*set enable_nestloop = false; */
EXPLAIN ANALYZE
SELECT al.attname as foreign,
k.nspname as schema,
cr.relname as foreigntable,
a2.attname as local
FROM pg_constraint ct
Borrador
Panel de Salida
Salida de datos
Comentar
Mensajes
Historial
QUERY PLAN
text
1 Hash Left Join (cost=290.25..431.63 rows=1 width=256) (actual time=12.806..16.698 rows=402 loops=1)
2 Hash Cond: (cl.relnamespace = k.oid)
3 -> Hash Join (cost=289.11..430.48 rows=1 width=196) (actual time=12.762..16.114 rows=402 loops=1)
4 Hash Cond: ((a2.attrelid = cl.oid) AND (a2.attnum = ct.conkey[1]))
5 -> Seq Scan on pg_attribute a2 (cost=0.00..110.35 rows=4135 width=70) (actual time=0.013..2.133 rows=4135 loops=1)
6 -> Hash (cost=288.86..288.86 rows=17 width=163) (actual time=10.650..10.650 rows=402 loops=1)
7 -> Hash Join (cost=233.03..288.86 rows=17 width=163) (actual time=9.322..9.978 rows=402 loops=1)
8 Hash Cond: (cl.oid = ct.conrelid)
9 -> Seq Scan on pg_class cl (cost=0.00..53.84 rows=484 width=8) (actual time=0.010..0.458 rows=484 loops=1)
10 -> Hash (cost=232.82..232.82 rows=17 width=155) (actual time=8.843..8.843 rows=402 loops=1)
11 -> Hash Join (cost=91.28..232.82 rows=17 width=155) (actual time=6.070..8.351 rows=402 loops=1)
12 Hash Cond: ((a1.attrelid = cr.oid) AND (a1.attnum = ct.confkey[1]))
13 -> Seq Scan on pg_attribute a1 (cost=0.00..110.35 rows=4135 width=70) (actual time=0.003..2.383 rows=4135 loops=1)
14 -> Hash (cost=85.25..85.25 rows=402 width=122) (actual time=2.674..2.674 rows=402 loops=1)
15 -> Hash Join (cost=59.89..85.25 rows=402 width=122) (actual time=0.926..1.991 rows=402 loops=1)
16 Hash Cond: (ct.confrelid = cr.oid)
17 -> Seq Scan on pg_constraint ct (cost=0.00..19.84 rows=402 width=54) (actual time=0.033..0.399 rows=402 loops=1)
18 Filter: (contype = 'f'::"char")
19 -> Hash (cost=53.84..53.84 rows=484 width=68) (actual time=0.882..0.882 rows=484 loops=1)
20 -> Seq Scan on pg_class cr (cost=0.00..53.84 rows=484 width=68) (actual time=0.005..0.454 rows=484 loops=1)
21 -> Hash (cost=1.06..1.06 rows=6 width=68) (actual time=0.024..0.024 rows=9 loops=1)
22 -> Seq Scan on pg_namespace k (cost=0.00..1.06 rows=6 width=68) (actual time=0.007..0.014 rows=9 loops=1)
23 Total runtime: 16.981 ms
OK.
Unix
Lín 3 Col 41 Car 40
23 filas.
31 ms

Figura # 13 Con *nestedloop* desactivado

Como se puede observar en las figuras la diferencia del tiempo de ejecución de las dos consultas fue casi de 2 segundos, lo cual significa una gran diferencia en rendimiento para la aplicación.

A continuación se muestran dos figuras, donde se muestran los planes de ejecución para otra consulta en la cual fue manipulado el parámetro que habilita y deshabilita los ordenamientos en el optimizador de consultas de PostgreSQL. En la primera está activado el *sort*, (como ocurre por defecto) y en la segunda se encuentra desactivado. Se evidencia la diferencia en el tiempo de ejecución de estas, así como el plan de ejecución de cada una de ellas.

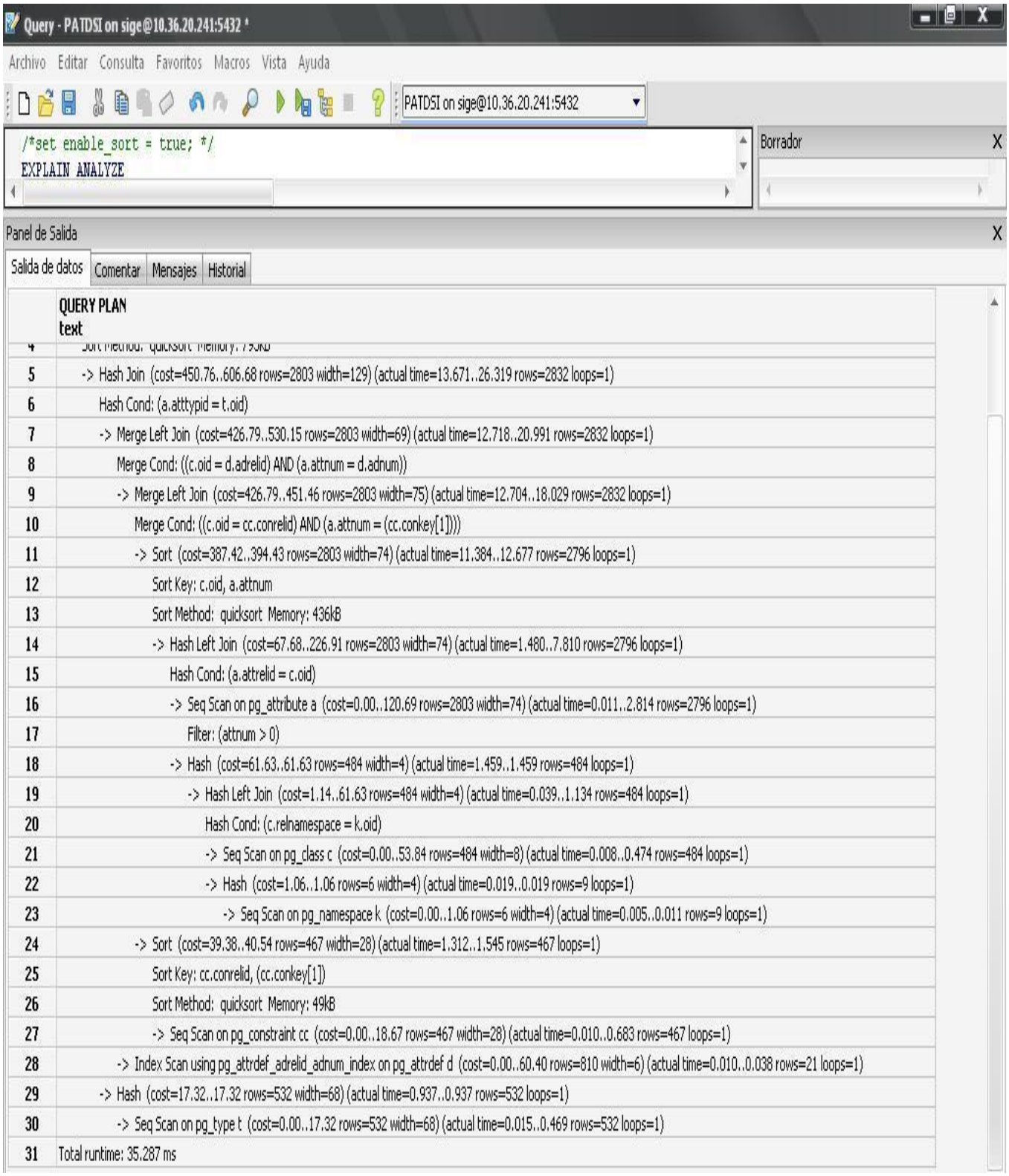


Figura # 14 *Sort* activado

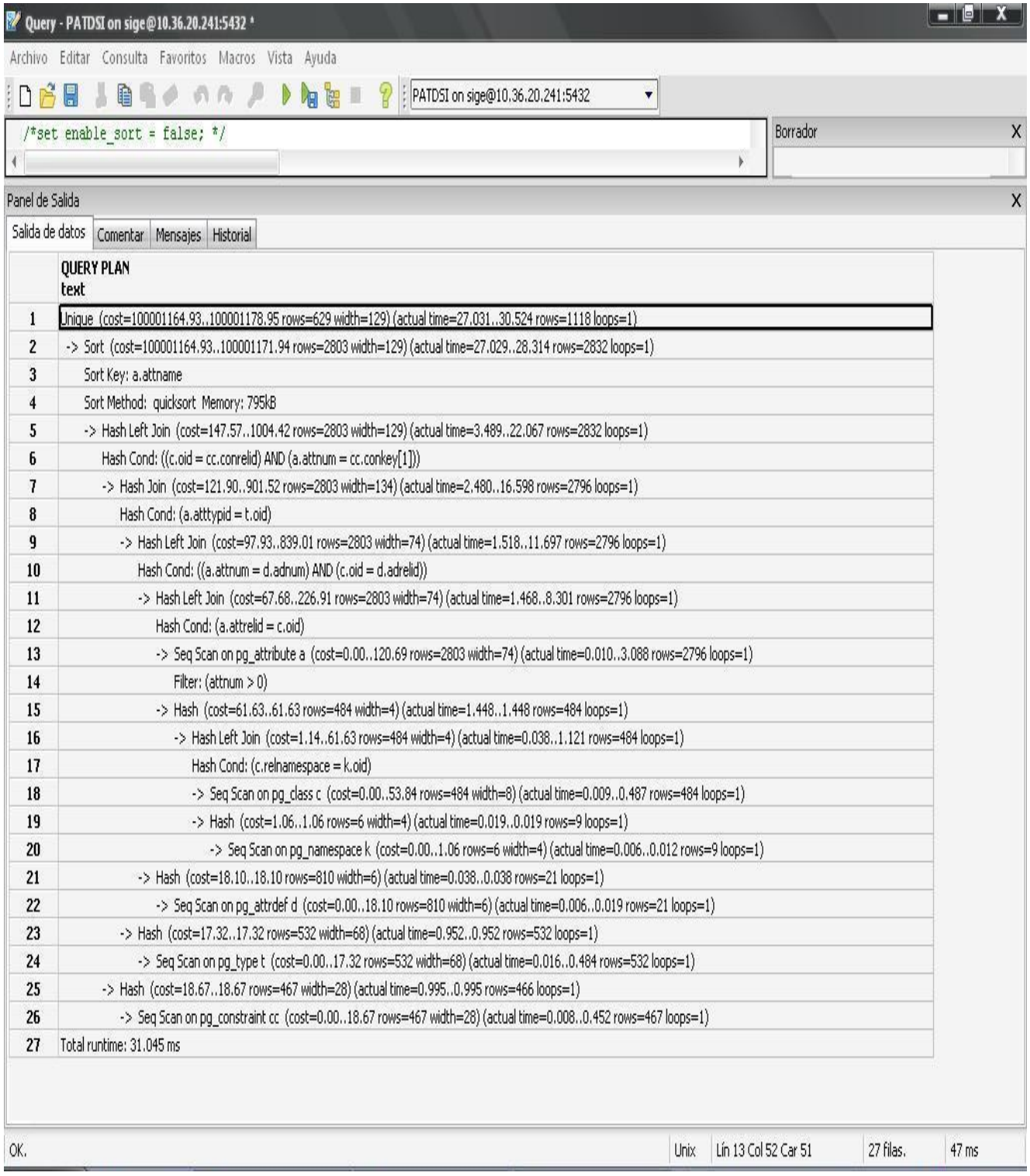


Figura # 15 *Sort* desactivado

3.3 Optimización de los índices y caminos de acceso

En el paso de la optimización de los índices se propone una consulta para conocer el comportamiento actual de los índices en la base de datos y así conocer cuáles son los que están poco utilizados. Al ejecutar esta consulta la base de datos del módulo Generador de Reportes del proyecto PATDSI se obtuvo el siguiente resultado:

table_name	index_name	times_used	table_size	index_size	num_writes
tbreportdatasource	pk_tbreportdatasource	73	8192 bytes	16 kB	1
nsubscriptiontype	pk_nsubscriptiontype	56	8192 bytes	16 kB	4
nformat	pk_nformat	55	8192 bytes	16 kB	4
tbsubscription	pk_tbsubscription	55	8192 bytes	24 kB	105
nrol	pk_nrol	20	8192 bytes	24 kB	3
tbmodel	pk_tbmodel	16	8192 bytes	40 kB	1
tbroltask	pk_tbroltask	12	8192 bytes	16 kB	5
nscheduletype	pk_nscheduletype	8	8192 bytes	16 kB	5
noptionrw	pk_noptionrw	7	8192 bytes	24 kB	4
ntask	pk_ntask	5	8192 bytes	24 kB	10
tbcategory	pk_tbcategory	4	8192 bytes	24 kB	2
tbregistro	PK15	3	8192 bytes	16 kB	3
tbdatasource	dsname	2	8192 bytes	32 kB	1
tbmodel	modelname	1	8192 bytes	40 kB	1
tbreport	title	1	8192 bytes	40 kB	2
tbschedule	pk_tbschedule	1	8192 bytes	16 kB	14
tbtemplate	pk_tbtemplate	1	8192 bytes	40 kB	2
ncss	pk_ncss	0	0 bytes	16 kB	0

Figura # 16 Tabla de Índices

En esta figura se puede apreciar cuales son los índices menos usados en el proyecto y el espacio en discos que dichos índices están ocupando, por lo que se pueden tomar medidas para saber cómo eliminar algunos de estos, con el propósito de mejorar el rendimiento del servidor a la hora de escribir en las tablas a las que pertenecen los índices, ya que cuando se escriben datos en una tabla con índice PostgreSQL también tiene que escribir en el archivo que almacena dicho índice, perdiendo un valioso tiempo

en acceso a disco. En este caso no se sugiere eliminar ningún índice ya que el tamaño que ocupan los índices listados por la consulta y la cantidad de escrituras en sus tablas son relativamente pequeños y no influyen apreciablemente en el rendimiento de la aplicación.

3.4 Optimización de hardware

Otra de las pruebas realizadas se describe a continuación y trata sobre la velocidad de lectura y escritura en disco el duro, siguiendo el procedimiento planteado en el epígrafe 2.1.5, ejecutada en una estación de trabajo con las siguientes características:

- Procesador: Intel Pentium 4 CPU 3.00GHz
- Memoria RAM: 1.0 GB, DDR2, 533 MHz
- Disco Duro: Barracuda 7200.7 SATA, 160 GB, 7200 RPM, buffer 8MB, Velocidad de transferencia de buffer 150 MB/s.
- Sistema Operativo: Ubuntu 8.10

La primera prueba donde se mide la velocidad de escritura secuencial en el disco duro arrojó el siguiente resultado:

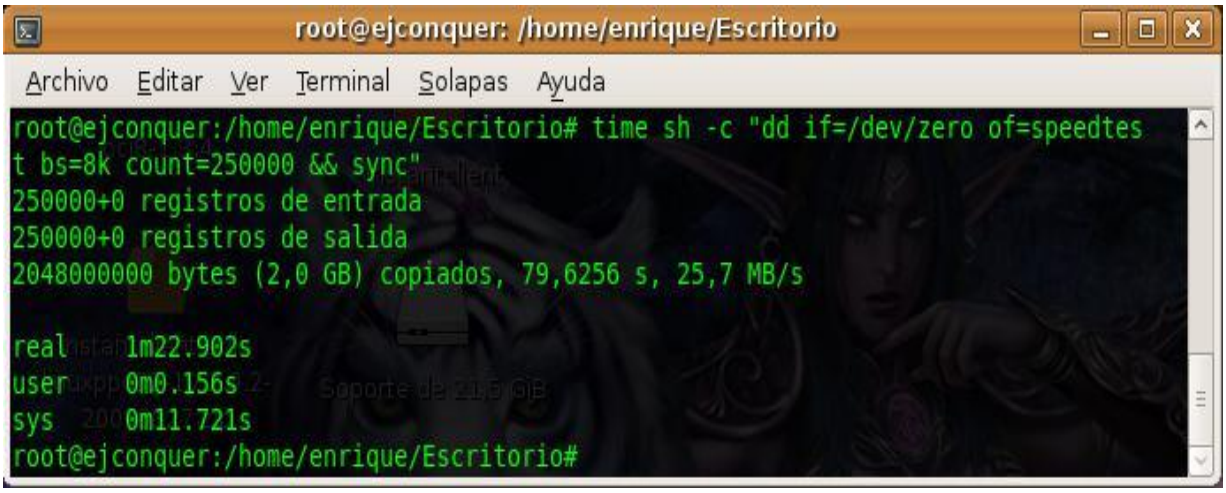


Figura # 17 Velocidad de escritura

Como se puede observar, la velocidad efectiva de copia en disco duro para este caso es de 25,7 MB/s, lo que quiere decir que PostgreSQL podrá guardar alrededor de 8MB de datos cada segundo, ya que la velocidad con la que escribe en el disco duro es de alrededor el 30% de la del Sistema Operativo.

La prueba de lectura secuencial también fue realizada utilizando el mismo procedimiento y se obtuvieron los siguientes resultados:

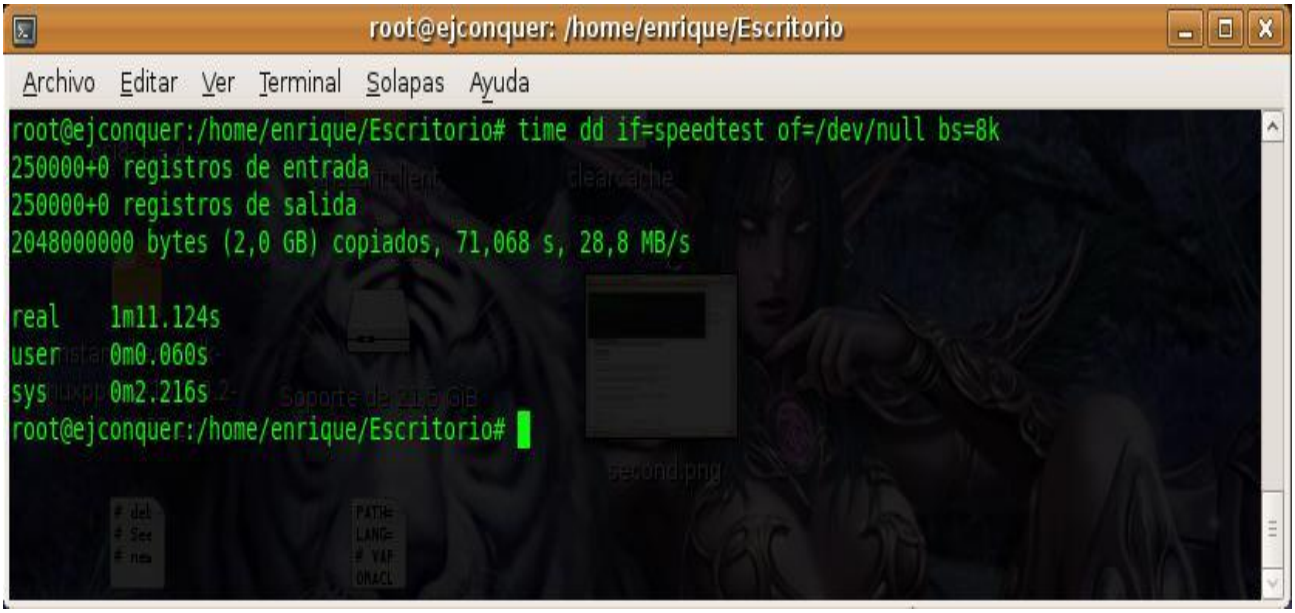


Figura # 18 Velocidad de lectura

Como se puede apreciar la velocidad de lectura es ligeramente mayor que la velocidad de escritura, el valor resultante de 28.8 MB/s indica que PostgreSQL podrá leer alrededor de 9MB de datos cada segundo para el caso de la configuración de hardware probada.

3.5 Optimización del fichero postgresql.conf

En este punto se realizaron pruebas con una herramienta llamada “jmeter” con el objetivo de observar como influían los cambios en los parámetros *shared_buffers* y *effective_cache_size* para un servidor de PostgreSQL funcionando sobre una estación de trabajo con las siguientes características:

- Procesador: Intel Pentium 4 CPU 3.00GHz

- Memoria RAM: 1.0 GB, DDR2, 533 MHz
- Disco Duro: Barracuda 7200.7 SATA, 160 GB, 7200 RPM, buffer 8MB, Velocidad de transferencia de buffer 150 MB/s.
- SO: Ubuntu 8.10

Esta herramienta simula múltiples conexiones al servidor realizando peticiones cada una cantidad de segundos especificados en la configuración de la prueba, creando carga en el servidor y por tanto permitiendo ver cómo reacciona este según su configuración cuando se encuentra bajo estrés.

Para la primera prueba realizada los parámetros mencionados tenían el valor que le pone PostgreSQL por defecto (32MB) para el caso de *shared_buffer* y 128MB para *effective_cache_size*. Se simularon primeramente 10 conexiones simultáneas que realizaban una petición por segundo de la siguiente consulta a la BD del proyecto PATDSI:

```
SELECT * FROM mod_recuperaciones.tbdatasource ds JOIN mod_recuperaciones.tbmodel md ON ds.iddatasource = md.iddatasource;
```

Después de un minuto de ejecución de este proceso se estableció una oncena conexión que realizó 500 peticiones en un intervalo de un minuto también, para medir el tiempo de respuesta del servidor para estas peticiones específicamente.

A continuación se muestran dos de las pantallas con resultados que brinda esta herramienta:

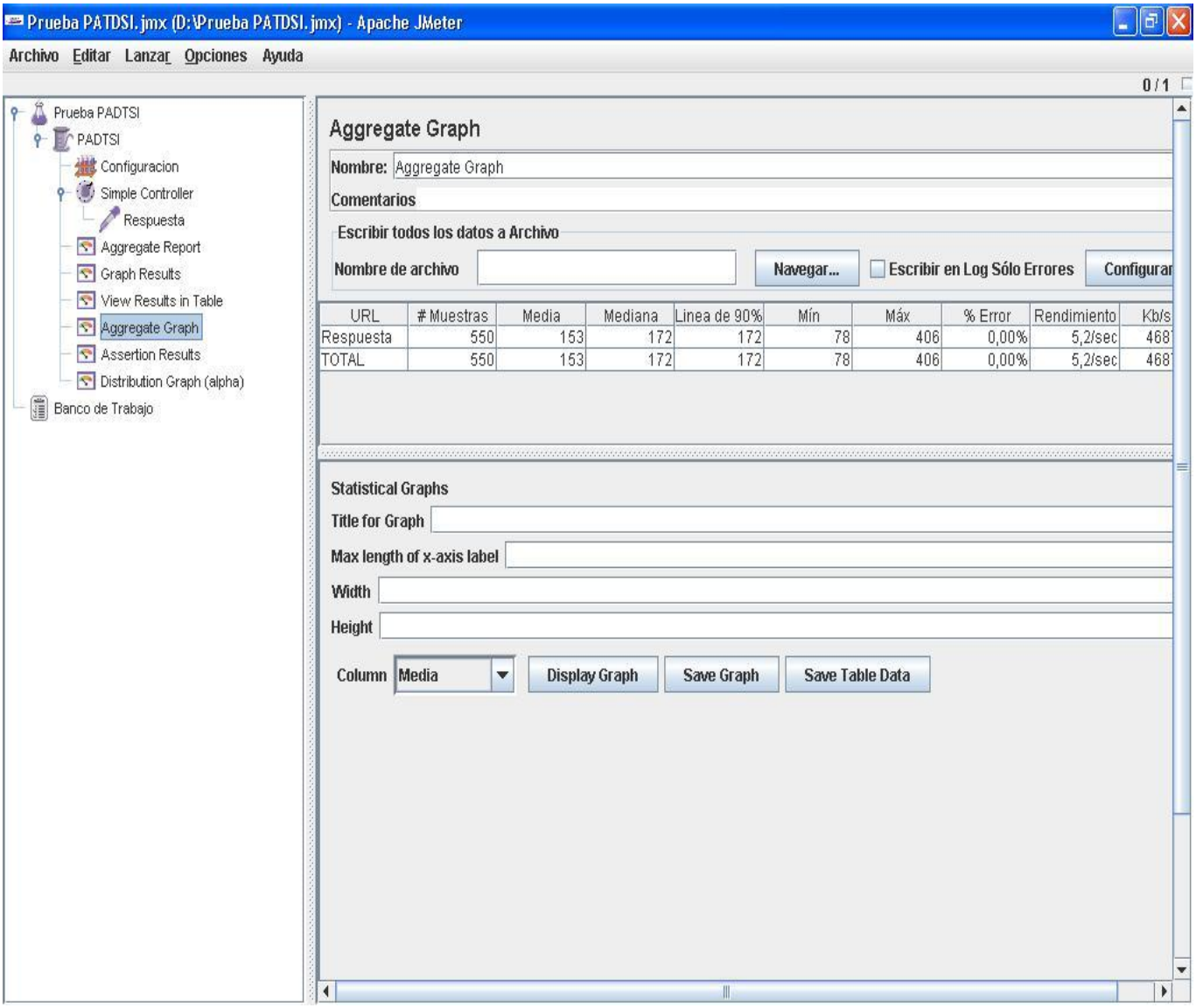


Figura # 19 Antes de modificar el fichero postgresql.conf

Como se puede apreciar en la imagen anterior el menor tiempo que tardó en ejecutarse la consulta fue de 78 ms y el máximo fue de 406, presentando un rendimiento promedio de 5.2 consultas realizadas por segundo. Esta imagen que se presenta a continuación es otra de las formas que tiene la herramienta de representar la información.

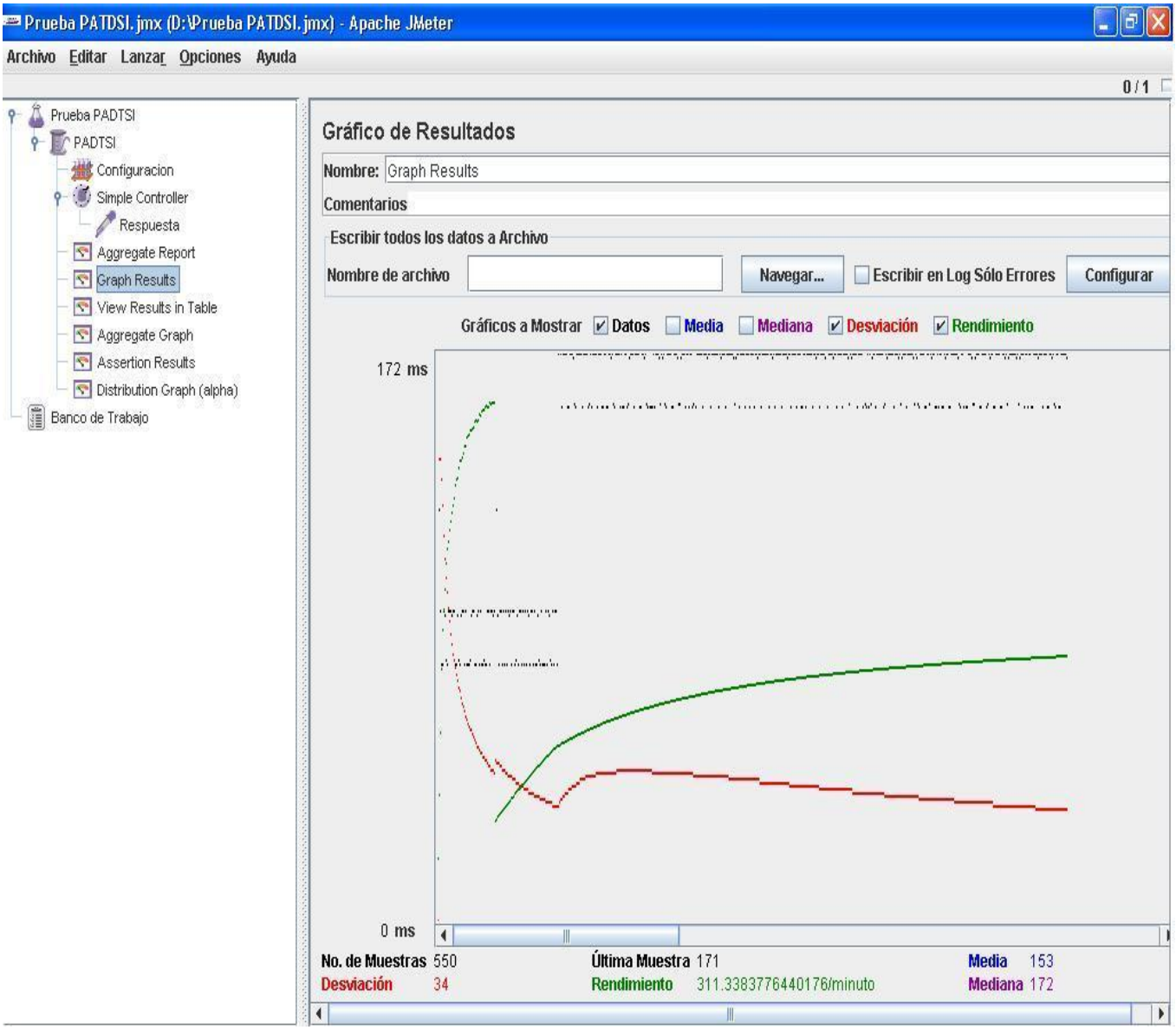


Figura # 20 Antes de modificar el fichero postgresql.conf. Gráfico

La segunda prueba se realizó luego de incrementado el valor de *shared_buffer* a 256MB, o sea, un cuarto de la memoria RAM del servidor y *effective_cache_size* a 768MB, la mitad de la memoria cache del mismo y se utilizaron los mismos parámetros que para la anterior prueba. Los resultados que esta arrojó fueron los siguientes:

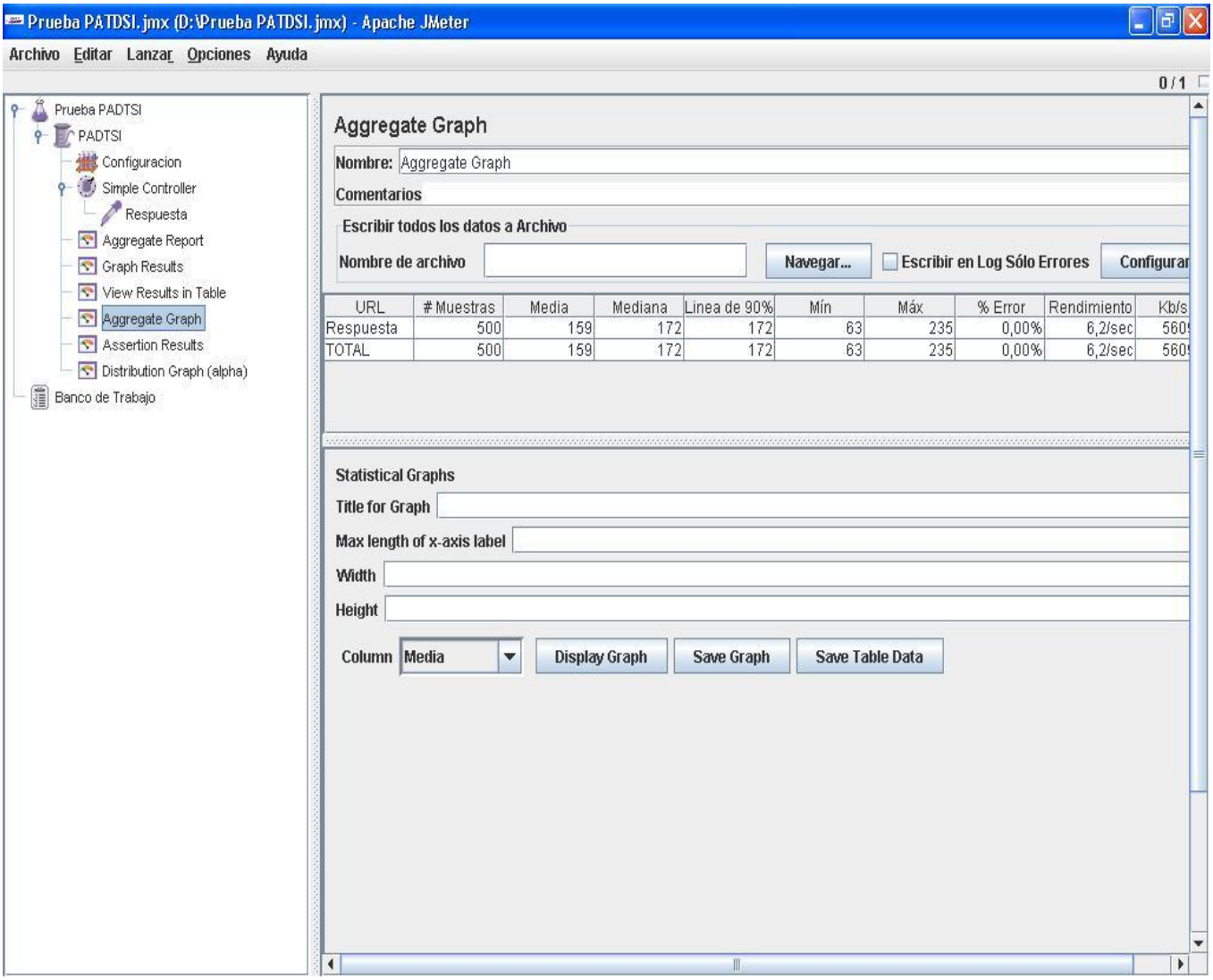


Figura # 21 Después de modificar el fichero postgresql.conf

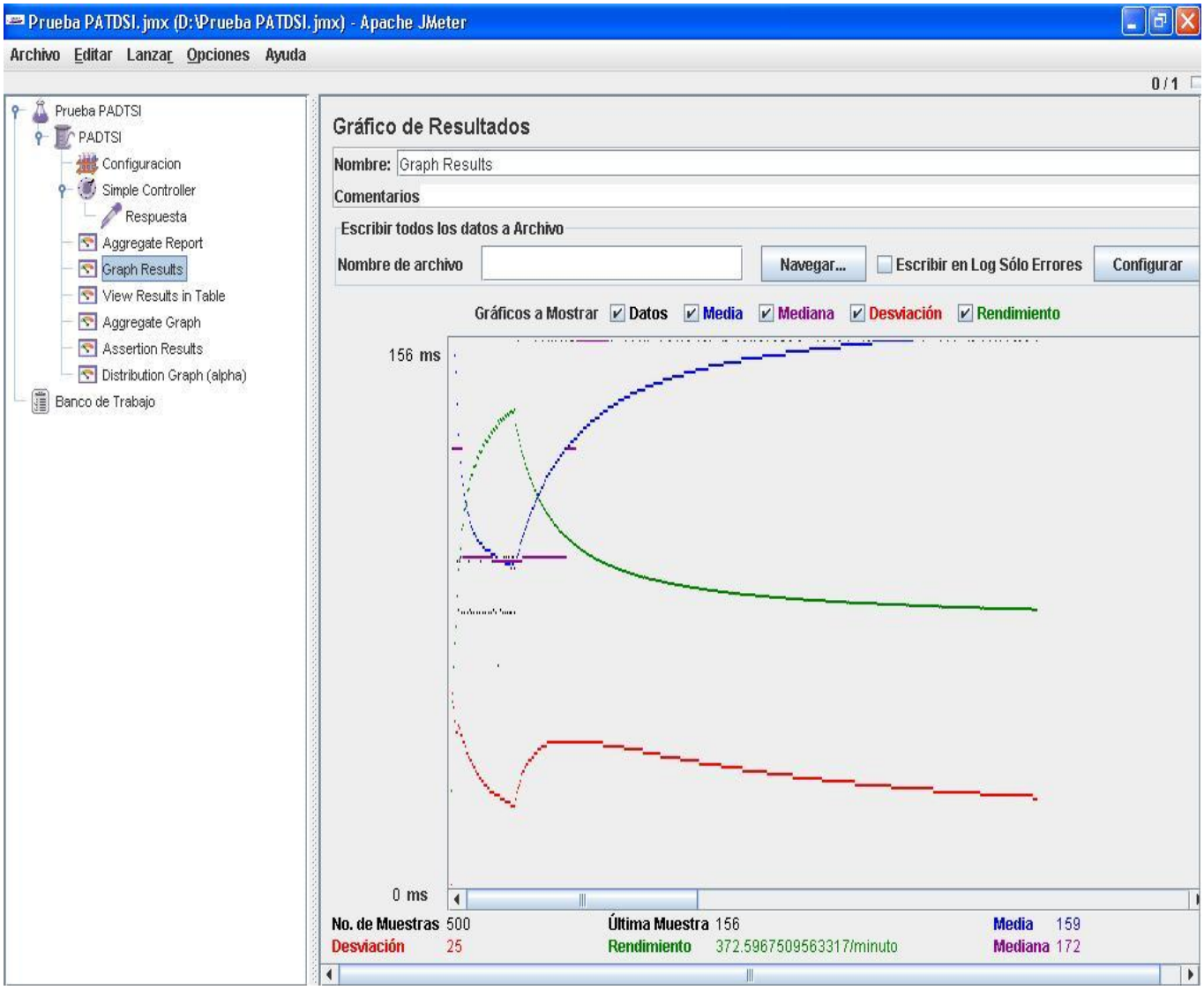


Figura # 22 Después de modificar el fichero postgresql.conf. Gráfico

Como se puede observar el menor tiempo de respuesta del servidor para la ejecución de la consulta disminuyó ligeramente pero el máximo si presentó una gran disminución, lo que permite afirmar que los cambios realizados en los dos parámetros analizados fueron gran ayuda a la hora de mejorar el rendimiento del servidor de PostgreSQL.

Conclusiones

A partir de los resultados obtenidos en la puesta en práctica de los pasos propuestos en la guía se puede llegar a la conclusión de que:

- No pudo ser probada la solución en su totalidad debido a que todas las condiciones del entorno no estaban creadas.
- Los resultados obtenidos fueron satisfactorios completamente.
- Es más cómodo saber mediante las consultas propuestas cómo se comporta la actividad de la base de datos, lo cual permite conocer con exactitud si funciona el método mediante el cual se ha optimizado.

Conclusiones Generales

Como conclusiones generales se puede concluir en que:

- Todos los objetivos de este trabajo fueron cumplidos satisfactoriamente y por tanto también las tareas generales de la investigación que se trazaron para dar cumplimiento a los objetivos.

- Quedó elaborada en su totalidad la guía de optimización propuesta como solución al problema y con los requisitos esperados, puesto que cada uno de los pasos explica detalladamente los parámetros o aspectos que pueden influir en el rendimiento de los servidores de bases de datos y en qué modo producen algún cambio en el rendimiento su modificación.

- Debe destacarse la visión práctica que ofrece la guía, pues esto le confiere al administrador de la base de datos una idea más exacta de lo que ocurre con la actividad del sistema, pudiendo así tomar mejores decisiones acerca de la optimización del servidor.

- Finalmente fueron evaluados los pasos que fueron posibles en el CENTALAD, observando los resultados satisfactorios que arrojaron las pruebas realizadas, lo que da una garantía de la validez del presente trabajo de diploma.

- Es importante mencionar una vez más la importancia que tiene la optimización de los servidores de bases de datos para lograr un buen rendimiento y desempeño de las aplicaciones y por tanto el beneficio que reporta este trabajo para lograr estos objetivos.

Recomendaciones

Al dar por concluida la realización de este trabajo de diploma los autores recomiendan que se le de una continuidad al mismo teniendo en cuenta los siguientes aspectos:

- Aplicar la guía a un proyecto desde sus inicios.
- Actualizar la guía cada vez que salga una versión nueva de PostgreSQL.
- Profundizar en cada uno de los pasos enumerados.
- Implementar una solución informática para el monitoreo de bases de datos en PostgreSQL que tenga en cuenta alguno de los aspectos propuestos en la guía como el monitoreo de índices, memoria, uso de los discos duros y rendimiento de los mismos, entre otros aspectos.

Bibliografía

1. **Andrés, M. M. (2001).** Universitat Jaume I. Obtenido de http://www3.uji.es/~mmarques/f47/apun/node1.html

2. **C. Batini, S. C. (1994).** Diseño Conceptual de Bases de Datos. Un enfoque de entidades-interrelaciones . Addison-Wesley / Díaz de Santos.

3. **Date, C. J. (2003).** Introducción a los Sistemas de Bases de Datos. La Habana: Féliz Varela.

4. **G.W. Hansen, J. H. (1997).** Diseño y Administración de Bases de Datos . Prentice Hall.

5. **Hernández, M. J. (1997).** Database Design for Mere Mortals . Addison-Wesley Developers Press.

6. **Kimball, R. (2002).** The Data Warehouse Toolkit (2ª edición). John-Wiley & Sons. .

7. **M.J. Folk, B. Z. (1992).** File Structures . Addison-Wesley.

8. **Mª. J. Ramos, A. R. (2006).** Sistemas gestores de bases de datos. España: McGraw-Hill .

9. **Manola, F. (1994).** An Evaluation of Object-Oriented DBMS Development. GTE Laboratories Incorporated.

10. **Medina, J. M. (2006).** Administración Postgres. Valencia.

11. **Moraga, C. R. (2002).** Generalidades sobre OODBMS. DIIC-Revista Informática , 1-10.

12. **R. Elmasri, S. N. (1999).** Sistemas de Bases de Datos. Conceptos fundamentales . Addison-Wesley.

13. **T. Connolly, C. B. (1996).** Database Systems. A Practical Approach to Design, Implementation and Management . Addison-Wesley.

14. **Tamayo, M., & Moreno, F. J. (2006).** Scielo. Obtenido de http://www.scielo.org.co/scielo.php?pid=S0120-56092006000300016&script=sci_arttext

15. **Villanueva, W. D. (2001-2002).** Web Universidad de Valencia. Recuperado el enero de 2009, de http://informatica.uv.es/iiguia/DBD/Practicas/boletin_1.pdf

16. **Yunta, L. R. (2001).** Bases de Datos documentales: estructura y uso. Madrid: CINDOC.

17. **Gascón, M. d. (2004).** Red Científica. Obtenido de http://www.redcientifica.com/doc/doc200203070001.html

18. **Kevin Loney, M. T. (2002).** ORACLE9I. MANUAL DEL ADMINISTRADOR

19. **Mannino.** DISEÑO APLICACIÓN y ADMINISTRACIÓN DE BASES DE DATOS España.

20. **OLGA PONS CAPOTE, N. M. (2005).** INTRODUCCIÓN A LAS BASES DE DATOS. EL MODELO RELACIONAL

21. **OLGA PONS CAPOTE, N. M.** INTRODUCCIÓN A LOS SISTEMAS DE BASES DE DATOS.

22. **Pablos. (2009).** kabytes. Obtenido de http://www.kabytes.com/programacion/mysqltuner-optimizar-mysql/

23. **SÁNCHEZ, G. C. (2001).** SISTEMAS GESTORES DE BASES DE DATOS.

24. **smarquez. (s.f.).** Velneo. Obtenido de http://blog.velneo.es/468/optimizacion-de-aplicaciones-en-clienteservidor-56-rendimiento-optimo/

25. **Stanek. (2006).** MS SQL SERVER 2005 MANUAL DEL ADMINISTRADOR.

Anexos

Anexo # 1 Tablas del catálogo de PostgreSQL 8.3.7

Tabla	Descripción
pg_agregate	Funciones agregadas.
pg_am	Métodos de acceso a índices.
pg_amop	Operadores asociados con las clases de los operadores de métodos de acceso a índices.
pg_amproc	Procedimientos soportados asociados con las clases de los operadores de métodos de acceso a índices.
pg_attrdef	Valores por defecto de las columnas.
pg_attribute	Información sobre las columnas de las tablas; una fila por cada columna de cada tabla.
pg_auth_members	Mantiene las relaciones de pertenencia entre los roles.
pg_authid	Almacena los roles de la base de datos.
pg_autovacuum	Almacena parámetros de configuración para el demonio del autovacuum.
pg_cast	Almacena métodos de conversión de tipos de datos, tanto los predefinidos como los definidos por el usuario con CREATE CAST.
pg_class	Almacena información sobre los objetos (tablas, índices, secuencias, vistas y otros objetos especiales), una fila por cada tabla, índice, secuencia o vista.
pg_constraint	Almacena la definición de las restricciones check, clave primaria, unicidad, y clave ajena de las tablas; las restricciones de valor no nulo se almacenan en pg_attribute.
pg_conversion	Información sobre las conversiones de

	codificación (juegos de caracteres).
pg_database	Bases de datos creadas en el clúster.
pg_depend	Guarda las relaciones de dependencia entre objetos de la base de datos (permite implementar el DROP... CASCADE o DROP... RESTRICT).
pg_description	Almacena una descripción o comentario opcional sobre los objetos.
pg_index	Información adicional sobre los índices.
pg_inherits	Jerarquía de la herencia de las tablas.
pg_language	Interfaces de llamadas o lenguajes en los que se pueden escribir los procedimientos almacenados.
pg_largeobject	Guarda la información que maquilla a los LOB; un LOB es identificado por un OID que se le asigna cuando se crea; los LOB se almacenan divididos en trozos pequeños.
pg_listener	Da soporte a las instrucciones LISTEN y NOTIFY; una listener crea una entrada para cada notificación que está esperando; un notificador rastrea pg_listener y actualiza cada entrada que le casa para informar que se ha producido una notificación.
pg_namespace	Esquemas y propietarios de esquemas
pg_opclass	Guarda las clases de operadores de métodos de acceso a índices
pg_operator	Información sobre los operadores definidos, ya sean predefinidos o definidos por el usuario
pg_pltemplate	Almacena plantillas de información para lenguajes procedurales.
pg_proc	Información sobre funciones y procedimientos.

pg_rewrite	Almacena las reglas de reescritura definidas.
pg_shdepend	Almacena las relaciones de dependencia entre objetos de la base de datos y objetos compartidos, como roles.
pg_shdescription	Almacena descripciones opcionales (comentarios) de los objetos compartidos de la base de datos.
pg_statistics	Información estadística; es mejor usa las vistas para visualizarla.
pg_tablespace	Información sobre los tablespaces del clúster.
pg_trigger	Información sobre la definición de los disparadores creados
pg_ts_config	Contiene entradas representando configuraciones de búsqueda de texto.
pg_ts_config_map	Contiene entradas que diccionarios de búsqueda de texto deben ser consultados.
pg_ts_dict	Contiene entradas que definen diccionarios de búsqueda de texto.
pg_ts_parser	Contiene entradas que definen parseadores de texto.
pg_ts_template	Contiene entradas que definen plantillas de búsqueda de texto.
pg_type	Almacena información sobre los tipos de datos ya sean escalares (CREATE TYPE), tipos compuestos (para cada tabla o fila de tabla).

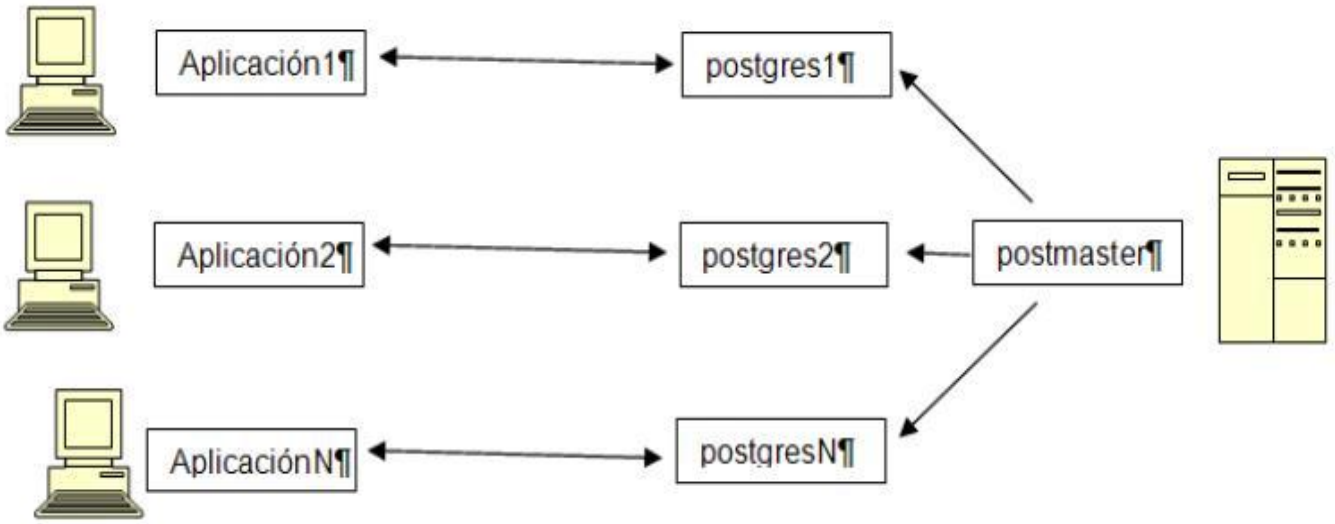
Anexo # 2 Vistas del catálogo de PostgreSQL 8.3.7

Vista	Descripción
pg_cursors	Lista los cursores disponibles en un momento dado.

pg_group	Compatibilidad hacia atrás; muestra los roles que no son de conexión.
pg_indexes	Información útil de los índices.
pg_locks	Muestra los bloqueos que se están llevando a cabo para la ejecución de las transacciones en curso.
pg_prepared_statements	Muestra todas las consultas preparadas que están disponibles para la presente sesión.
pg_prepared_xacts	Información sobre las transacciones preparadas para el protocolo de confirmación en dos fases.
pg_roles	Información útil sobre los roles (más legible que pg_authid).
pg_rules	Información sobre las reglas de reescritura creadas.
pg_settings	Información sobre los parámetros de puesta en marcha del servidor.
pg_shadow	Compatibilidad hacia atrás; muestra los roles de conexión.
pg_stats	Información legible de pg_statistics
pg_tables	Información legible sobre las tablas creadas
pg_timezone_abbrevs	Almacena las abreviaturas de los usos horarios.
pg_timezone_names	Almacena los nombres de los usos horarios.
pg_user	Información sobre los usuarios; forma alternativa de pg_shadow.
pg_views	Información sobre las vistas creadas en la base

de datos.

Anexo # 3 Arquitectura de PostgreSQL



Anexo # 4 Arquitectura de PostgreSQL detallado