Sistemas de Soporte para la Toma de Decisiones

Trabajo Práctico 3

"*Diseño Lógico e Implementación de Data Warehouse*"

FACULTAD DE INGENIERÍA.

Profesor: **Mg. Claudia López de Munaín.**
Aux. Graduado: **Lic. Celia Cintas.**

Alumnos:

Etchart, Walter
Huincalef Rodrigo A.
Luque Leandro
Martinovic, Andres

Año 2016

1) A continuación enumeramos algunas empresas a las que se les debe dise~nar un DW, ya sea bajo un esquema estrella o copo de nieve, en ambos casos detalle dimensiones, medidas y hechos utilizados:

- **Empresa de Vinos: Venta Online de Vinos, esta empresa necesita que se diseñe un DW para registrar la cantidad y ventas de sus vinos a sus clientes. Parte de la base de datos actual está compuesta por las siguientes tablas:**

  - **CUSTOMER (Code, Name, Address, Phone, BDay, Gender)**
  - **WINE (Code, Name, Type, Vintage, BottlePrice, CasePrice, Class)**
  - **CLASS (Code, Name, Region)**
  - **TIME (TimeStamp, Date, Year)**
  - **ORDER (Customer, Wine, Time, nrBottles, nrCases)**

  Estas tablas representan las entidades principales de un ER, por lo que es necesario derivar las relaciones entre ellas para poder dise~nar correctamente el DW. Luego de realizado el dise~no del DW realice las siguientes consultas SQL:

  - **Muestre porcentajes de tipos de vinos más vendidos en X año.**
  - **Cual es la temporada con mayor cantidad de ventas de X vino?**
  - **Que clientes han realizado más compras a lo largo de 4 a~nos?**

- **Inmobiliaria: En este caso deseamos darle al Gerente Principal una vista general del negocio, en términos de cuáles son las propiedades que la inmobiliaria maneja y el seguimiento del trabajo de cada Agente dentro de la empresa. Nuestra Inmobiliaria cuenta con las siguientes tablas en su base de datos:**

  - **OWNER (IDOwner, Name, Surname, Address, City, Phone)**
  - **ESTATE (IDEstate, IDOwner, Category, Area, City, Province, Rooms, Bedrooms,**
  - **Garage, Meters)**
  - **CUSTOMER (IDCust, Name, Surname, Budget, Address, City, Phone)**
  - **AGENT (IDAgent, Name, Surname, Oce, Address, City, Phone)**
  - **AGENDA (IDAgent, Data, Hour, IDEstate, ClientName)**
  - **VISIT (IDEstate,IDAgent, IDCust, Date, Duration)**
  - **SALE (IDEstate,IDAgent, IDCust, Date, AgreedPrice, Status)**
  - **RENT (IDEstate,IDAgent, IDCust, Date, Price, Status, Time)**

  Luego de realizado el dise~no del DW realice las siguientes consultas SQL:

  - **Que tipo de propiedad se vendió por el precio más alto con respecto a cada ciudad y meses?**
  - **Quien ha comprado un piso con el precio más alto con respecto a cada mes?**

- **Cual es la duración media de visitas en las propiedades de cada categoría?**

1) Las respuesta y códigos a este inciso se encuentran anexados a este informe.

2) Configurar PosgreSQL para funcionar como DW:

A. **Configurar Write-ahead logging en un disco distinto.**
B. **Aumentar el límite de conexiones.**
C. **Aumentar límites de memoria. (shared_buffers, work_mem, temp_buffers, etc.)**
D. **Que es AUTOVACUUM ? Es recomendable tener esta característica en nuestro DW ? Como se configura en PostgreSQL?**
E. **Que es un tablespace? Cuales son los beneficios de utilizarlos? Crear dos dbs y colocarlas en discos diferentes.**
F. **Que es una Windowing function? Cómo se implementan en SQL? Aplicar un ejemplo en la db creada anteriormente.**
G. **Es necesaria una política de caducidad de los datos en un DW? Justique.**

2.A) WAL o Write Ahead Logging, es un método estándar de Postgres para asegurar la integridad de los datos, que se basa en mantener un log respecto de los cambios que sufren los datos (donde los índices y tablas residen), y almacenar el valor real de los datos,luego de haber registrado estos cambios en el log. Esta característica se encuentra habilitada por defecto en Postgres. Los registros de WAL en Postgres se almacenan en el directorio **pg_xlog** en el directorio de datos, y se divide en una serie archivos (archivos de segmento), cada uno de un tamaño de 16 MB. Cada archivo de segmento se divide en páginas de un tamaño de 8Kb cada una, dependiendo el tipo de contenido del registro del tipo de evento que fue registrado. Los nombres de los archivos se componen de 24 dígitos hexadecimales, divididos en 3 partes:

- Los primeros 8 dígitos representan el time-line, o secuencia de los archivos de log escritos en disco.
- Los siguientes 8 dígitos representan la división por páginas o bloques.
- Los últimos 8 dígitos son el desplazamiento dentro del log del wal, de cada transacción.

Para configurar Write Ahead Logging, se deben editar el archivo de configuración de Postgres **postgresql.conf,** ubicado en Ubuntu 14.04 en /etc/postgresql/9.3/main/, y se deben descomentar las siguientes propiedades:

- **wal_level(hot_standby)**. Este parámetro establece el nivel de detalle de la información de las transacciones que se registrará en los archivos de segmentos. El valor por defecto es *minimal*, que solo produce la información necesaria para recuperarse de una colisión o apagado repentino. La opción *archive* añade información útil para la generación de archivos de segmentos, y *hot_standby* además introduce información necesaria para reconstruir el estado de las transacciones desde WAL.
- **fsync(on).** Cuando este parámetro se encuentra habilitado, el servidor de Postgresql se asegurará que las actualizaciones sean físicamente escritas en disco, emitiendo llamadas de sistema fsync () o métodos equivalentes. Ésto asegura que el cluster de base de datos se pueda recuperar a un estado consistente luego de un fallo del sistema operativo.
- **archive_mode(on).** Cuando este parámetro está habilitado, los archivos de segmento son enviados a un almacenamiento especificado en el parámetro archive_command. Este parámetro se debe establecer al inicio del servidor.
- **archive_command('mkdir -p /tmp/WAL-Logs && test ! -f /tmp/WAL-Logs/%f && cp %p /tmp/WAL-Logs/%f').** Este es el comando de shell ejecutado cuando se completa un segmento de archivo WAL. La cadena de conexión emplea dos placeholders: %p qué es la ruta en el servidor donde se encuentra el archivo (incluyendo el nombre de éste), y %f que es el nombre del archivo.
- **archive_timeout(120).** El comando archive_command es solo invocado para segmentos WAL completados. Así, si el servidor genera poco tráfico de WAL, puede transcurrir un largo tiempo desde la finalización de la transacción y su almacenamiento seguro en un almacenamiento de archivos. Para limitar qué tan antiguos los datos almacenados pueden ser, se puede establecer este parámetro para forzar al servidor a cambiar a un archivo de segmento WAL periódicamente. Cuando este valor sea mayor a cero, el servidor cambiara a un nuevo archivo de segmento, cuando la cantidad de segundos especificada haya transcurrido desde el último cambio de archivo de segmento.

2.B) Para aumentar el límite de conexiones, se debe el archivo postgresql.conf, tentativamente ubicado en Ubuntu 14.04 en /etc/postgresql/9.3/main/. Dentro de este archivo encontraremos el parámetro **max_connections**, éste parámetro aplica la cantidad máxima de conexiones de clientes. Esto es muy importante para algunos de los parámetros siguientes (particularmente work_mem), porque hay recursos de memoria que pueden ser ubicados por cliente, entonces el número máximo de clientes puede sugerir el máximo de memoria utilizada posible. Generalmente, postgresql en un buen hardware puede soportar unos pocos cientos de conexiones. Si se desea tener miles de conexiones, se debe considerar utilizar connection pooling software, para reducir la sobrecarga de conexión.

2.C) Para cargar grandes cantidades de datos a una base de datos de PostgreSQL, se debe aumentar los valores de shared_buffers, work_men y max_locks_per_transaction en el archivo postgresql.conf.

El parámetro **shared_buffers** designa la cantidad de memoria que se utiliza para los búferes de memoria compartida. La documentación de PostgreSQL indica que, por razones de rendimiento, es probable que se deba utilizar una configuración mayor al valor mínimo de 128 KB o 16 KB multiplicado por el número que se configuró como el valor de max_connections. Se recomienda configurar shared_buffers para utilizar varias decenas de megabytes en las instalaciones de producción.

Cuando carga grandes cantidades de datos, es muy probable que necesite una mayor configuración de shared_buffers que el valor predeterminado. Después de cambiar este parámetro en el archivo postgresql.conf, se deberá reiniciar el cluster de base de datos.

work_men: Si se hacen muchas ordenaciones complejas, y se tiene bastante memoria, incrementando esta variable le permitirá a PostgreSQL a realizar ordenamientos más distendidos en memoria, obviamente incrementando la performance en comparación a las basadas en disco.

Este tamaño está aplicado a cada uno de los ordenamientos para cada usuario, y consultas complejas pueden utilizar múltiples buffers de memoria dedicados a estos. Establecerlo en 50MB y teniendo 30 usuarios ejecutando consultas, estaría utilizando 1.5GB de memoria real. Más allá, si una consulta implica hacer ordenamientos con juntas de 8 tablas, requeriría 8 veces work_mem. Debería considerar lo que tiene establecido en max_connections para establecer el work_mem apropiadamente. Éste es un valor donde los almacenes de datos, donde los usuarios ejecutan consultas extensas, podrían llegar a utilizar gigas en memoria.

temp_buffers: se utiliza para establecer el número máximo de buffers temporales para cada sesión en la base de datos. Estos son buffer de sesiones locales usados sólo para el acceso a tablas temporales.

max_locks_per_transaction: El valor de max_locks_per_transaction indica la cantidad de objetos de base de datos que se pueden bloquear de manera simultánea. En la mayoría de los casos, el valor predeterminado de 64 es suficiente. Sin embargo, cuando se carga una gran cantidad de datasets (por ejemplo, varios miles) a la vez, la cantidad de bloqueos de objeto concurrentes para la transacción puede exceder 64.

2.D) VACUUM es el proceso que realiza la limpieza y optimización del uso de una base de datos en PostgreSQL. El demonio de VACUUM realiza esto por medio de la eliminación de tuplas marcadas para borrar, reorganizando datos a nivel físico y manteniendo estadísticas respecto de las tablas (cantidad tuplas, cantidad de páginas de memoria que ocupan), para ser utilizadas por el planeador de consultas en la elección de la mejor estrategia .

El VACUUM se realiza periódicamente se para:

- **Recuperar espacio en disco perdido por datos borrados o actualizados**. En operaciones normales de PostgreSQL, las tuplas que se eliminan o quedan obsoletas por una actualización no se eliminan físicamente de su tabla, sino que permanecen hasta se ejecuta un VACUUM. Por lo tanto, es necesario hacer VACUUM periódicamente, especialmente en las tablas frecuentemente actualizadas.
- Actualizar las estadísticas de datos utilizados por el planificador de consultas SQL.
- Protegerse ante la pérdida de datos por reutilización de identificadores de transacción.

VACUUM puede recibir ciertos parámetros para realizar los diferentes tipos de vaciamiento:

- **FREEZE:** La opción FREEZE está depreciada y será removida en liberaciones futuras.
- **FULL:** Selecciona el vaciamiento “completo”, el cual recupera más espacio, pero toma mucho más tiempo y bloquea exclusivamente la tabla.
- **VERBOSE:** Imprime un reporte detallado de la actividad de vaciamiento para cada tabla.
- **ANALYZE:** Actualiza las estadísticas usadas por el planeador para determinar la forma más eficiente de ejecutar una consulta.
- **tabla**: El nombre de una tabla específica a vaciar. Por defecto, se toman todas las tablas de la base de datos actual.
- **columna**: El nombre de una columna específica a analizar. Por defecto se toman todas las columnas. Este comando es muy útil para automatizar vaciamientos a través de cualquier sincronizador de tareas.

Existen dos variantes de VACUUM: VACUUM y VACUUM FULL; VACUUM puede ejecutarse en paralelo con las operaciones de base de datos tales como INSERT, UPDATE, DELETE, SELECT y permite que la base de datos continúe sus ejecuciones normalmente, aunque no se permite la modificación de la definición de la tabla con comandos como ALTER TABLE, mientras el demonio de VACUUM está en ejecución. Por el contrario, VACUUM FULL, requiere el bloqueo exclusivo sobre la tabla durante la ejecución, y por lo tanto, no puede ser hecho en paralelo con el uso de la tabla. VACUUM FULL puede reclamar más espacio en disco pero se ejecuta de manera más lenta.

Este comando es muy útil para automatizar vaciamientos a través de cualquier sincronizador de tareas (ya sea el cron de *nix u otro de Windows).

Asimismo, existe la opción del Autovacuum, cuya funcionalidad es ir realizando de manera paulatina la mantención de la base, automatizando la ejecución de los comandos VACUUM y ANALYZE. Previamente y antes de activar esta funcionalidad, es recomendable leer sobre las consideraciones a tener en cuenta, para no degradar la performance de nuestro servidor.

Contar con VACUUM en un data warehouse es recomendable según el esquema que se emplee para la implementación, ya que si se emplea un esquema copo de nieve, donde puede ser necesario realizar múltiples joins entre tablas de dimensión normalizadas y, la característica de ANALYZE puede ser útil para mantener un rendimiento óptimo entre las operaciones de join. Por el contrario, la característica de VACUUM estándar o FULL, no sería de uso frecuente en un data warehouse, debido a que en este almacén solamente se realiza la inserción o actualización de datos, pero nunca se efectúan operaciones de modificación o borrado, que puedan requerir organizar el almacenamiento físico de los datos en disco.

Para configurar autovacuum en Postgres 9.3 se debe acceder al archivo de configuración postgresql.conf, instalado en la versión Ubuntu 14.04 en el path: **/etc/postgresql/9.3/main/postgresql.conf** y habilitar/establecer el valor del parámetro autovacuum = on. Y asegurarse que la característica track_counts, para el mantenimiento de estadística de las tablas de la base de datos se encuentra habilitada.
A continuación, se describen algunos de los parámetros de configuración para personalizar el comportamiento del demonio autovacuum en el servidor:

- l**og_autovacuum_min_duration (Entero)**: Este parámetro causa que cada acción ejecutada por el demonio sea registrada si se ejecuta por al menos, el número de milisegundos especificado. Si se establece este valor a 0, se loguean todas las acciones de autovacuum, y con el valor -1 se deshabilitan las acciones de autovacuum. Este parámetro puede ser útil para rastrear la actividad de autovacuum.

- **autovacuum_max_workers (Entero):** Especifica el número máximo de procesos hijos de autovacuum (además del demonio lanzador principal) que pueden estar en ejecución al mismo tiempo. Por defecto son 3. Este parámetro sólo puede ser establecido al inicio del servidor.

- **autovacuum_naptime (Entero):** Especifica el retraso mínimo entre la ejecución de autovacuums en cualquier base de datos. En cada ronda, el demonio examina la base de datos y emite comandos VACUUM y ANALYZE, según se necesiten en cada tabla. El retraso es medido en segundos, y por defecto se encuentra establecido a un minuto.

- **autovacuum_vacuum_threshold (Entero):** Este parámetro especifica el número mínimo de tuplas actualizadas o borradas, necesitadas para disparar un VACUUM en una tabla. Por defecto se encuentra establecido a 50 filas. Este parámetro se puede sobreescribir en cada tabla, cambiando la configuración de almacenamiento.

- **autovacuum_analyze_threshold (Entero):** Especifica el número mínimo de tuplas insertadas, actualizadas o borradas necesitadas para disparar un ANALYZE en cualquier tabla. Por defecto este valor es de 50 filas. Este parámetro se puede sobreescribir en cada tabla, cambiando la configuración de almacenamiento.

- **autovacuum_vacuum_scale_factor (Punto Flotante):** especifica el tamaño de la fracción de la tabla a partir del cual agregar la función autovacuum_vacuum_threshold cuando se decide lanzar el VACUUM. Por defecto, el valor tomado es el 0.2 (20% del tamaño de la tabla). Este parámetro puede ser seteado únicamente en el archivo postgresql.conf o por línea de comando. Esta configuración puede ser realizada para tablas en forma individual.

- **autovacuum_analyze_scale_factor (Punto Flotante):** especifica el tamaño de la fracción de la tabla a partir del cual agregar la función autovacuum_vacuum_threshold cuando se decide lanzar el ANALYZE. Por defecto, el valor tomado es el 0.1 (10% del tamaño de la tabla). Este parámetro puede ser seteado únicamente en el archivo postgresql.conf o por línea de comando. Esta configuración puede ser realizada para tablas en forma individual.

- **autovacuum_freeze_max_age (Entero):** especifica la antigüedad máxima (en las transacciones) que debe tener el campo pg_class.relfrozenxid de una tabla para resguardar el ID, antes de que postgres realice una operación de VACUUM, que borre el ID de la transacción.

- **autovacuum_vacuum_cost_delay (Entero):** especifica el valor de costo de retardo que se usará en operaciones de autovacuum. Si se especifica -1 se usará el valor regular o por defecto. El valor por defecto es 20 milisegundos. Este parámetro puede ser seteado únicamente en el archivo postgresql.conf o por línea de comando. Esta configuración puede ser realizada para tablas en forma individual.

- **autovacuum_vacuum_cost_limit (Entero):** especifica el valor de costo limite que se usará en operaciones de autovacuum. Si se especifica -1 se usará el valor regular o por defecto. El valor es proporcionalmente distribuido por la cantidad de procesos de autovacuum ejecutándose. Este parámetro puede ser seteado únicamente en el archivo postgresql.conf o por línea de comando. Esta configuración puede ser realizada para tablas en forma individual.

2.E) Un **tablespace** es una ubicación de almacenamiento donde pueden ser guardados los datos correspondientes a los objetos de una base de datos. Este provee una capa de abstracción entre los datos físicos y lógicos permitiéndonos asignar espacio para todos los segmentos administrados del sistema de gestión de base de datos (DBMS). En síntesis es un puente entre el sistema de ficheros del sistema operativo y la base de datos.

Un segmento es un objeto de la base de datos el cual ocupa espacio físico, como por ejemplo, los datos de una tabla y los índices. Una vez creado, un tablespace puede ser referido por su nombre cuando se crean segmentos de la base de datos.

Un tablespace puede pertenecer sólo a una BD. Estos se utilizan para mantener juntos los datos de usuarios o de aplicaciones para facilitar su mantenimiento o mejorar las prestaciones del sistema. De esta manera, cuando se crea una tabla se debe indicar el espacio de tablas al que se destina. Por defecto, se depositan en el espacio de tablas **SYSTEM**. Este espacio de tablas es el que contiene el diccionario de datos, por lo que conviene reservarlo para el uso del servidor, y asignar las tablas de usuario a otro.

Se pueden ver los espacios de tablas definidos en nuestra BD con el comando SQL siguiente:

SELECT * FROM user_tablespaces;

Cada tabla o índice pertenece a un tablespace, es decir cuando se crea una tabla o índice se crea en un tablespace determinado. Son estructuras donde se almacenan los objetos del esquema de la base de datos, tales como tablas, índices, etc. con la particularidad de poderse repartir en varios ficheros.

Tipos de *tablespaces*

- *Tablespace* SYSTEM.
    - Se crea automáticamente al hacer la instalación de la base de datos.
    - Contiene el diccionario de datos.
- *Tablespaces* TEMPORALES.
    - Es aquél en el que solamente puede haber objetos temporales. No se pueden crear objetos permanentes como pueden ser los índices, las tablas o los segmentos de *rollback*.
    - Optimización operaciones de ordenación.
- De tipo deshacer cambios (9i).
    - Se utilizan para gestionar el poder deshacer las transacciones incompletas (*rollback).*
- Con tamaño de bloque variable (9i).
- De tipo *BigFile* (10g).

Mediante el uso de los tablespace, un administrador puede controlar la distribución del disco de instalación del motor de base de datos (en nuestro caso PostgreSQL). Esto es útil en al menos dos formas. En primer lugar, si la partición o el volumen en el que se ha inicializado el cluster se queda sin espacio y no pueden ser extendidos, un espacio de tabla se puede crear en una partición diferente y se utiliza hasta que el sistema pueda ser reconfigurado.

En segundo lugar, los tablespace permiten utilizar el conocimiento del patrón de uso de los objetos de base de datos para optimizar el rendimiento. Por ejemplo, un índice que se utiliza muy frecuentemente se puede colocar en un disco muy rápido, tal como un dispositivo de estado sólido. Al mismo tiempo, una tabla que almacena los datos archivados que rara vez

se utiliza podría ser almacenada en un sistema de disco más lento y menos costoso que el caso anterior.

Para crear un tablespace se utiliza la siguiente sentencia SQL:

CREATE TABLESPACE tablespace_name [**OWNER** user_name] **LOCATION** 'directory'

2.F) Una ***window function*** es una función que realiza un cálculo a lo largo de un conjunto de filas de la tabla, que están de alguna manera relacionadas con la fila actual. Es comparable al tipo de cálculo que puede ser hecho con una *aggregate function*. Pero a diferencia de las *aggregate functions* tradicionales, el uso de una *window function* no causa que las filas sean agrupadas dentro de una única fila de salida (las filas mantienen sus identidades separadas). La window function es capaz de acceder a más que solo la fila actual del resultado de la consulta.

En SQL se implementa haciendo uso de la opción OVER, en conjunto con el uso de una función de agregación (AVG, SUM, COUNT, RANK) más PARTITION BY para indicar sobre que se aplica la función, y GROUP BY para indicarle cómo se deben ordenar los datos.

Ejemplo:

last_name	salary	department
Jones	45000	Accounting
Adams	50000	Sales
Johnson	40000	Marketing
Williams	37000	Accounting
Smith	55000	Sales

```
SELECT last_name, salary, department, rank() OVER (
    PARTITION BY department
    ORDER BY salary
    DESC
)
FROM employees;
```

Resultado:

last_name	salary	department	rank
Jones	45000	Accounting	1
Williams	37000	Accounting	2
Smith	55000	Sales	1
Adams	50000	Sales	2
Johnson	40000	Marketing	1

Window Function - Tabla Ventas - DB Vinos

```
SELECT "Id_wine", "Precio_orden", rank() OVER (
        PARTITION BY "Id_wine"
        ORDER BY "Precio_orden"
        DESC
) AS total_price
FROM ventas LIMIT 10;
```

Resultado:

	Id_wine bigint	Precio_orden bigint	total_price bigint
1	2	522	1
2	3	949	1
3	3	105	2
4	4	832	1
5	4	817	2
6	7	752	1
7	8	1066	1
8	8	641	2
9	9	503	1
10	10	584	1
11	11	1090	1

2.G) La política de caducidad es necesaria en un entorno de DW. Este período de tiempo dependerá del ámbito en el cuál se encuentre el DW, ya que en base a éste se determina cuál es el período de tiempo durante el que tiene sentido almacenar los datos.

Por ejemplo:

1. En organizaciones que registran mora de los clientes que manejan, tienen asignado por ley (Argentina) un período máximo de 5 años para conservar los datos. Pasado dicho período, los datos asociados a las personas deben borrarse de forma permanente.
2. En organizaciones dedicadas al estudio de la meteorología, es preciso conservar los datos durante un largo periodo de tiempo (15~20 años), dado que los mismos se emplean para el pronosticar el clima.

3) ETL (Extract, Transform, Load) en Python

- **A. Tomar datos de un archivo csv, de json y una db SQL. (Para este ejercicio utilizar las bibliotecas psycopg2 y pandas).**
  1. **De los tres DataFrames realizar las siguientes operaciones y mostrar sus resultados, join, merge, concat, append, con una clave a elección.**
  2. **Guardar en una db SQL, la intersección de los datos obtenidos en el punto anterior.**

3) Las respuesta y códigos a este inciso se encuentran anexados a este informe.