**Titulación: Grado en Ingeniería Informática y Sistemas de Información**

**Curso: 2019-2020. Convocatoria Ordinaria de Junio**

**Asignatura: Bases de Datos Avanzadas – Laboratorio**

**Practica 3: Seguridad, Usuarios y Transacciones.**

**ALUMNO 1:**

**Nombre y Apellidos: JAVIER GARCÍA JIMÉNEZ**

**DNI: 09099503J**

**ALUMNO 2:**

**Nombre y Apellidos: ISABEL MARTÍNEZ GÓMEZ**

**DNI: 06027983M**

**Fecha: 08/06/2020**

**Profesor Responsable: José Carlos Holgado**

Mediante la entrega de este fichero los alumnos aseguran que cumplen con la normativa de autoría de trabajos de la Universidad de Alcalá, y declaran éste como un trabajo original y propio.

En caso de ser detectada copia, se puntuará **TODA** la asignatura como Suspenso – Cero.

**Plazos**

Tarea online: Semana 13 de Abril, Semana 20 de Abril y semana 27 de Abril.

Entrega de práctica: Día 8 de Junio. Aula Virtual

Documento a entregar: Este mismo fichero con las respuestas a las cuestiones planteadas, con el código SQL utilizado en cada uno de los aparatos. Si se entrega en formato electrónico se entregará en un ZIP comprimido: **DNI'sdelosAlumnos\_PECL3.zip**

**AMBOS ALUMNOS DEBEN ENTREGAR EL FICHERO EN LA PLATAFORMA.**

**Introducción**

El contenido de esta práctica versa sobre el manejo de las transacciones en sistemas de bases de datos, así como el control de la concurrencia y la recuperación de la base de datos frente a una caída del sistema. Las transacciones se definen como una unidad lógica de procesamiento compuesta por una serie de operaciones simples que se ejecutan como una sola operación. Entre las etiquetas BEGIN y COMMIT del lenguaje SQL se insertan las operaciones simples a realizar en una transacción. La sentencia ROLLBACK sirve para deshacer todos los cambios involucrados en una transacción y devolver a la base de datos al estado consistente en el que estaba antes de procesar la transacción. También se verá el registro diario o registro histórico del sistema de la base de datos (en PostgreSQL se denomina WAL: Write Ahead Loggin) donde se reflejan todas las operaciones sobre la base de datos y que sirve para recuperar ésta a un estado consistente si se produjera un error lógico o de hardware. La versión de postgres a utilizar deberá ser la versión 12.

**Actividades y Cuestiones**

En esta parte la base de datos **TIENDA** deberá de ser nueva y no contener datos. Además, consta de 5 actividades:

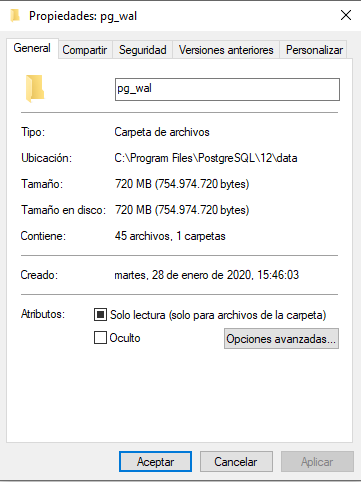
* Conceptos generales.
* Manejo de transacciones.
* Concurrencia.
* Registro histórico.
* Backup y Recuperación

Cuestión 1: Arrancar el servidor Postgres si no está y determinar si se encuentra activo el diario del sistema. Si no está activo, activarlo. Determinar cuál es el directorio y el archivo/s donde se guarda el diario. ¿Cuál es su tamaño? Al abrir el archivo con un editor de textos, ¿se puede deducir algo de lo que guarda el archivo?

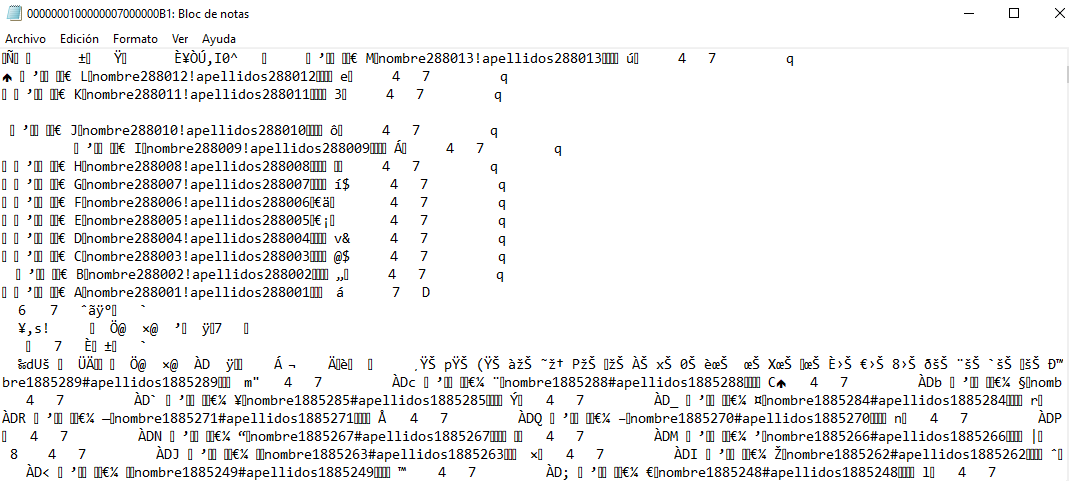
El diario del sistema o WAL se habilita automáticamente, no se requiere ninguna acción del administrador excepto que se asegure de que se cumplen los requisitos de espacio en disco para los registros del WAL.

Los registros del WAL se almacenan en el directorio pg\_wal dentro del data\_directory, que como vimos en prácticas anteriores el directorio de datos se encontraba en “C:\Program Files\PostgreSQL\12\data”. Dentro de la carpeta pg\_wal dentro del data\_directory observamos un conjunto de archivos de segmento, normalmente de 16 MB de tamaño. Cada segmento está dividido en bloques, normalmente de 8 KB. El contenido del registro depende del tipo de evento que se registra. Los archivos de segmento reciben como nombre, números que van en orden ascendente comenzando en 000000010000000000000000.

Si observamos las propiedades de la carpeta pg\_wal vemos que su tamaño es de 720 MB.



Al abrir un archivo con el editor de textos, vemos algunas palabras que podemos entender como nombre y apellidos, sin embargo, muchas otras cosas salen con símbolos y números y no podemos entenderlo. Aun así, deducimos que se trata de una tabla de la base de datos que guarda información de los nombres, apellidos y algún otro campo.

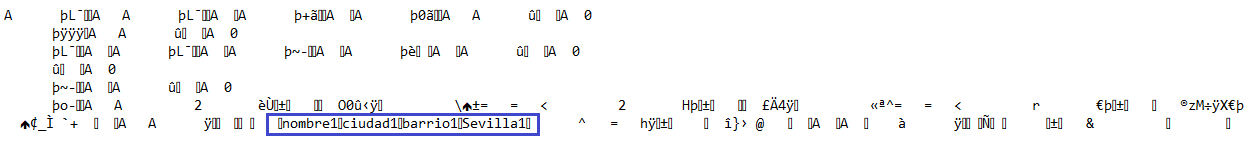


Cuestión 2: Realizar una operación de inserción de una tienda sobre la base de datos **TIENDA**. Abrir el archivo de diario ¿Se encuentra reflejada la operación en el archivo del sistema? ¿En caso afirmativo, por qué lo hará?



Tras hacer la inserción en la tabla Tienda, podemos ver que el archivo de segmento se ha modificado, por lo que usando el editor de texto hemos buscado por el final “nombre1” y hemos dado con la tupla insertada. Como podemos ver, los campos son legibles y se distinguen bien de los símbolos y caracteres que aparecen a su alrededor, aun así, el identificador de tienda no lo podemos ver, posiblemente esté codificado.

Postgres hace esto para garantizar la integridad de los datos. Al insertar una tupla en una tabla se está produciendo una alteración en la base de datos, estos cambios en los archivos de datos deben escribirse solo después de que esos cambios se han registrado, es decir, no necesitamos vaciar las páginas de datos al disco en cada transacción COMMIT, porque sabemos que en caso de un bloqueo podremos recuperar la base de datos mediante REDO.



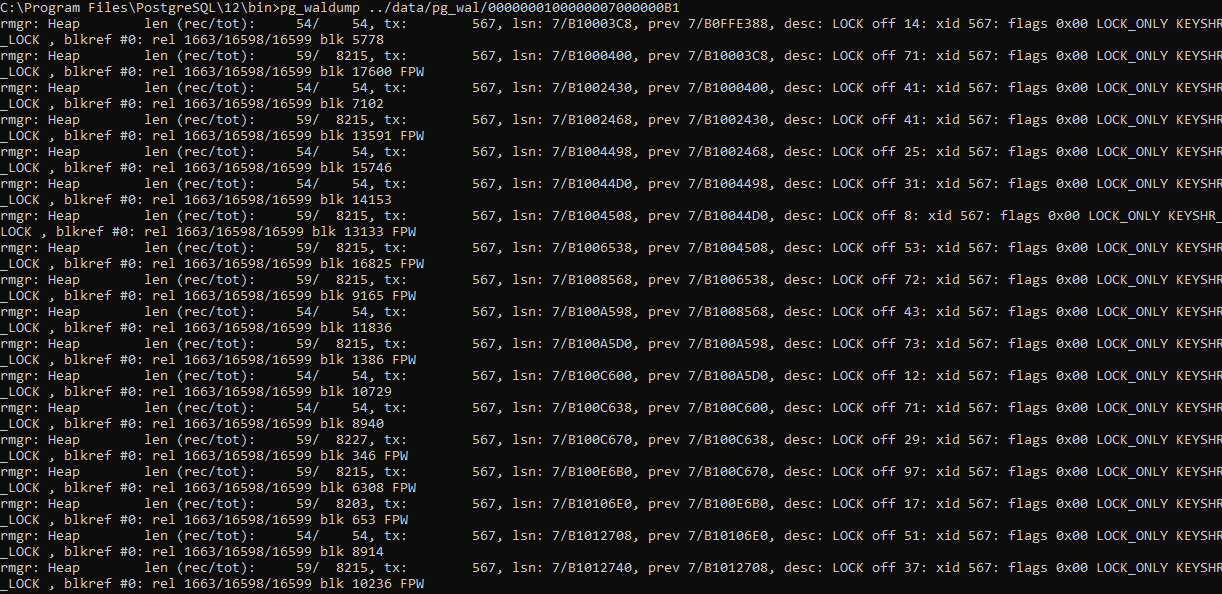
Cuestión 3: ¿Para qué sirve el comando pg\_waldump.exe? Aplicarlo al último fichero de WAL que se haya generado. Obtener las estadísticas de ese fichero y comentar qué se está viendo.

Es una herramienta de PostgreSQL que muestra una representación legible por humanos del WAL de una base de datos.

El comando pg\_waldump es principalmente útil para depuración o propósitos educativos. Este comando sólo puede ejecutarlo el usuario que instaló el servidor ya que requiere acceso de sólo lectura al directorio de datos.

Además, pg\_waldump tiene implementado una serie de opciones que podemos usar en función de qué queremos ver u obtener del WAL.

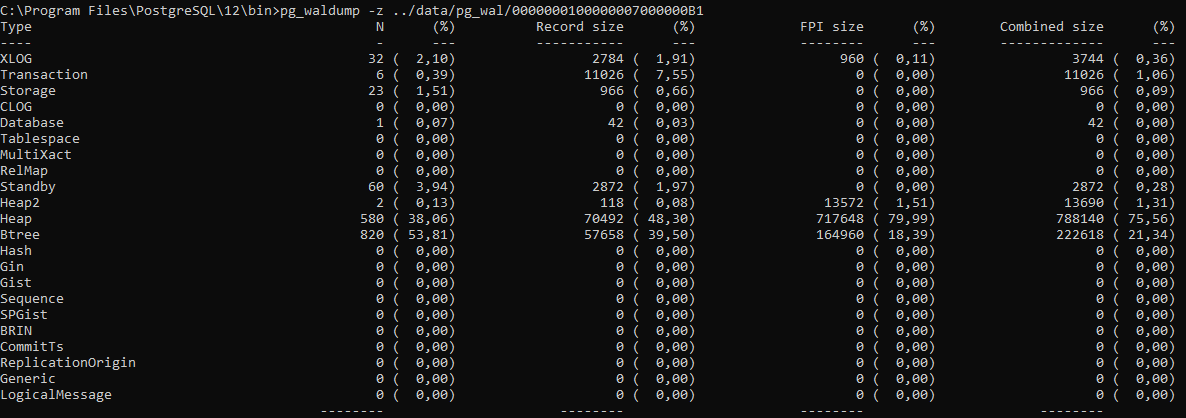
Al aplicar el comando pg\_waldump.exe al último fichero de WAL generado nos muestra lo siguiente:



Al aplicar el comando pg\_waldump, se muestra una representación legible del último archivo modificado del WAL. Podemos observar que nos indica información referente a los registros generados por el administrador de recursos (rmgr) como: el tipo de registro, el tamaño del registro, el bloque que ocupa, el LSN (log sequence number), la transacción previa y descendente y la referencia de bloque.

En los registros de tipo Standby y Storage podemos ver además el estado del registro (LOCK o CREATE). Por otra parte, en los registros de tipo Transaction podemos ver información referente al uso de la memoria caché.

Para obtener las estadísticas del fichero, usamos el parámetro -z. Obtenemos lo siguiente:



Como podemos observar, se muestran varias columnas con las estadísticas del archivo:

* Type: muestra el tipo de registro
* N: muestra el número de registros que hay del tipo de registro
* (%): muestra el porcentaje de transacciones en el archivo
* Record size: muestra el tamaño de los registros almacenados en el archivo
* (%): muestra el porcentaje del tamaño de los registros del tipo de registro con respecto al total
* FPI (Full page image) size: muestra el tamaño de las imágenes de los bloques completos
* (%): muestra porcentaje de FPI
* Combined size: muestra el tamaño de la combinación del número de registros y las imágenes de los bloques
* (%): muestra el porcentaje del combined size

Por tanto, vemos que hay un alto número de registros del tipo btree y heap que ocupan el 53,81 y 38,06% del archivo respectivamente. Por otra parte, aunque hay muchos menos, hay registros del tipo Standby, XLOG, Storage, Transaction y Heap2.

Por otra parte, vemos que, aunque hay más registros de tipo btree en el archivo, los registros de tipo heap ocupan más con un 48,3% del tamaño del total del archivo ocupando los registros del tipo btree un 39,5%.

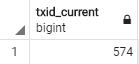
En cuanto al FPI, los registros de tipo heap ocupan un 79,99% con muchísima diferencia con respecto a los registros de tipo btree que ocupan un 18,39%, Heap2 1,51% y XLOG 0,11%.

Por último, como combined size, como es lógico vemos que el tipo de registros que más tamaño ocupa en el archivo son los registros de tipo heap con un 75,56% mientras que los de tipo btree que son los predominantes ocupan un 21,34%.

Cuestión 4: Determinar el identificador de la transacción que realizó la operación anterior. Aplicar el comando anterior al último fichero de WAL que se ha generado y mostrar los registros que se han creado para esa transacción. ¿Qué se puede ver? Interpretar los resultados obtenidos.

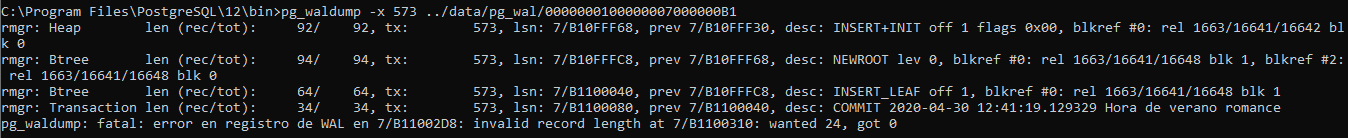
Para determinar el identificador de la transacción que realizó la operación anterior, ejecutaremos el comando txid\_current que nos devolverá el identificador de la transacción actual:





Como no nos interesa la transacción actual que no tiene nada, sino la anterior, simplemente tenemos que saber que su identificador es el 573 ya que no ha habido ningún error ni ningún fallo y la transacción con identificador 573 se ha ejecutado de forma correcta.

Para saber los registros que se han creado en la transacción con identificador 573 aplicaremos el comando pg\_waldump con el parámetro -x para que solo nos muestre los registros marcados con la XID pasada por parámetro, que será 573.



Una vez que ejecutamos el comando nos muestra la información de los registros marcados por la transacción. Podemos ver acerca de ellos:

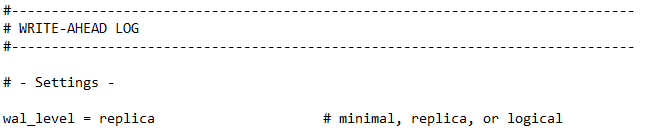
* rmgr: vemos que los tipos que hay asociados a la transacción son Btree y Heap
* len: vemos que hay 34 registros en la transacción
* tx: es el identificador de la transacción, que como vemos es 573
* lsn (log sequence number) se define por dos números en hexadecimal separados por una ‘/’
* prev: define el anterior lsn al actual
* desc: define el posterior lsn al actual.
* off: offset

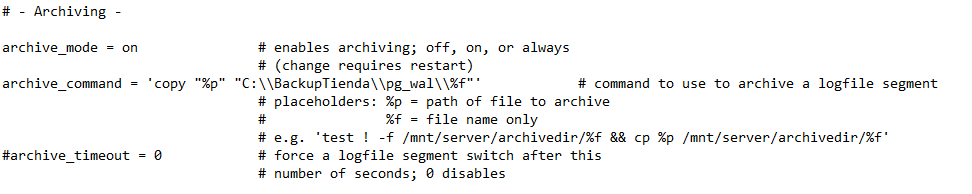
Por tanto, vemos que, en esta transacción, el Heap hace una operación INSERT + INIT y el Btree hace una operación INSERT\_LEAF, cuando finaliza la transacción se realiza un COMMIT “COMMIT 2020-04-30 12:41:19.129329 Hora de verano romance” por lo que finalmente se almacenan los cambios realizados en la base de datos junto con la fecha y hora a la que se han realizado.

Cuestión 5: Se va a crear un backup de la base de datos **TIENDA**. Este backup será utilizado más adelante para recuperar el sistema frente a una caída del sistema. Realizar solamente el backup mediante el procedimiento descrito en el apartado 25.3 del manual (versión 12 es *"Continous Archiving and point-in-time recovery (PITR)"*.

Para crear el backup de la base de datos Tienda seguimos el manual de Postgres de la sección 25.3.

Primero tendremos que irnos a la configuración del archivo postgresql.conf y modificar los parámetros *archive\_mode*, *archive\_command* y *wal\_level*:

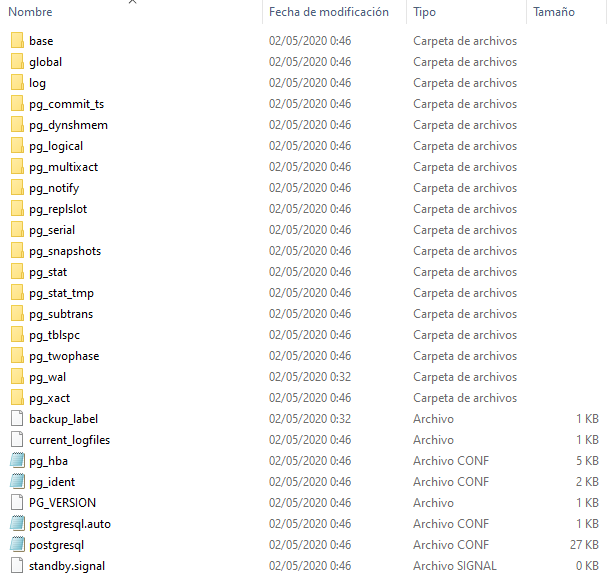




Después de modificar el archivo postgresql.conf, procederemos a crear el backup de la base de datos realizando la siguiente query donde pg\_basebackup es el módulo para hacer la copia de seguridad y con el parámetro h especificamos la dirección del usuario, con el parámetro p especificamos el número de puerto de la base de datos, con los parámetros -R y -D especificamos lectura y escritura y finalmente “C\BackupTienda” es el directorio donde queremos que se guarde la copia de seguridad.



Se crea una copia de seguridad en el directorio “C:\BackupTienda” de nuestro ordenador:



Cuestión 6: Qué herramientas disponibles tiene PotsgreSQL para controlar la actividad de la base de datos en cuanto a la concurrencia y transacciones? ¿Qué información es capaz de mostrar? ¿Dónde se guarda dicha información? ¿Cómo se puede mostrar?

PostgreSQL proporciona un amplio conjunto de herramientas para que los desarrolladores administren el acceso concurrente a los datos.

Internamente y por defecto, la consistencia de los datos se mantiene mediante el uso de un modelo multiversión (MVCC, Multiversion Concurrency Control). El MVCC funciona mediante instancias de la base de datos en un momento específico en el transcurso de una transacción.

Esto evita que las declaraciones vean datos inconsistentes producidos por transacciones concurrentes que realizan actualizaciones en las mismas filas de datos, proporcionando un mecanismo de control de concurrencia.

MVCC evita las metodologías de bloqueo de los sistemas de bases de datos tradicionales minimizando la contención de bloqueo para permitir un rendimiento razonable en entornos multiusuario.

La principal ventaja de usar el modelo MVCC de control de concurrencia en lugar de realizar bloqueos es que en MVCC, los bloqueos adquiridos para consultar datos (leer) no entran en conflicto con los bloqueos adquiridos para escribir datos, por lo que la lectura nunca bloquea a la escritura y la escritura nunca bloquea a la lectura. El uso adecuado de MVCC generalmente proporcionará un mejor rendimiento de los locks.

Aparte de MVCC Postgres implementa otras herramientas para controlar la actividad de la base de datos en cuanto a la concurrencia y transacciones. Postgres implementa otra herramienta que es el aislamiento de transacciones.

El estándar SQL define cuatro niveles de aislamiento de transacciones: Read uncommitted, Read committed, Repeatable read y Serializable. Estos niveles están ordenados de menos estrictos a más estrictos siendo el nivel más estricto Serializable, que es definido por el estándar y dice que cualquier ejecución concurrente de un conjunto Serializable garantiza que las transacciones producirán el mismo efecto que ejecutarlas de una en una en algún orden (serial).

Por otra parte, PostgreSQL proporciona la herramienta de bloqueos explícitos que proporciona varios modos de bloqueo para controlar el acceso concurrente a los datos en las tablas. Estos modos se pueden usar para el bloqueo controlado por la aplicación en situaciones en las que MVCC no proporciona el comportamiento deseado. La mayoría de los comandos de PostgreSQL adquieren automáticamente bloqueos de modos apropiados para asegúrese de que las tablas referenciadas no se eliminen o modifiquen de manera incompatible mientras el comando se ejecuta. Para examinar una lista de los bloqueos actualmente pendientes en un servidor de base de datos habría que usar la vista del sistema pg\_locks.

Los modos de bloqueo son:

**Bloqueo a nivel de tabla**

Todos los modos de bloqueo son bloqueos a nivel de tabla. Dentro de este nivel se tienen distintos modos de bloqueo. La única diferencia real entre un modo de bloqueo y otro es el conjunto de modos de bloqueo con los que cada conflicto entra en conflicto. Dos transacciones no pueden contener modos bloqueo en conflicto en la misma tabla al mismo tiempo. Sin embargo, una transacción nunca entra en conflicto con sí mismo. Por ejemplo, podría adquirir el ACCESS EXCLUSIVE y luego adquirir ACCESS SHARE. Los modos de bloqueo no conflictivos pueden ser mantenidos simultáneamente por muchas transacciones.

Algunos modos de bloqueo son autoconflictivos, por ejemplo, un bloqueo ACCESS EXCLUSIVE no puede ser retenido por más de una transacción a la vez mientras que otros no son autoconflictivos por ejemplo, un bloqueo de ACCESS SHARE puede ser mantenido por múltiples transacciones.

En total hay 8 modos de bloqueo, en orden de menos a más conflictivo: ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE y ACCESS EXCLUSIVE.

**Bloqueo a nivel de fila**

Además de los bloqueos a nivel de tabla, hay bloqueos a nivel de fila. Una transacción puede contener bloqueos en conflicto en la misma fila. Además, dos transacciones nunca pueden mantener bloqueos en conflicto en la misma fila.

En total hay 4 modos de bloqueo, en orden de menos a más conflictivo: FOR KEY SHARE, FOR SHARE, FOR NO KEY UPDATE y FOR UPDATE.

**Bloqueo a nivel de bloque**

Además de los bloqueos a nivel de tabla y fila, los bloqueos compartidos/exclusivos a nivel de bloque se utilizan para controlar el acceso de lectura/escritura a los bloques de la tabla en memoria compartida. Estos bloqueos se liberan inmediatamente después de una fila actualizada.

**Deadlocks**

El uso del bloqueo explícito puede aumentar la probabilidad de deadlocks, donde dos o más transacciones mantienen los locks que el otro quiere. PostgreSQL detecta automáticamente situaciones de deadlocks y las resuelve al cancelar una de las transacciones involucrada, permitiendo que el otro u otros se completen.

**Bloqueos de asesoramiento**

Los bloqueos de asesoramiento pueden ser útiles para estrategias de bloqueo que son incómodas para el modelo MVCC.

Por otra parte, la información que PostgreSQL puede mostrar son todos los bloqueos activos y los procesos que están esperando a que estos bloqueos se liberen. Esta información se guarda en la vista de sistema pg\_locks donde se muestra toda la información referente a los bloqueos que hay en el sistema. La información de la vista pg\_locks se puede ampliar combinándolo con la vista pg\_stat\_activity.



Cuestión 7: Crear dos usuarios en la base de datos que puedan acceder a la base de datos **TIENDA** identificados como usuario1 y usuario2 que tengan permisos de lectura/escritura a la base de datos tienda, pero que no puedan modificar su estructura. Describir el proceso seguido.

En esta cuestión nos encargaremos de crear dos usuarios nuevos que puedan acceder a la base de datos Tienda con permisos de lectura y escritura.

Para ello usaremos primero la consulta SQL *create user* para crear ambos usuarios:

* *create user usuario1*
* *create user usuario2*

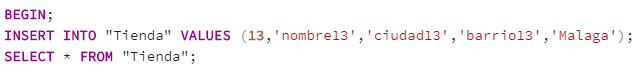
Una vez tenemos ambos usuarios creados, ahora pasamos a darles los permisos que necesitamos. Para ello utilizaremos la consulta SQL *grant*, y para asignar dichos permisos, completaremos con *select, insert, update y delete*. Por tanto, removemos los permisos que tiene el esquema público y procedemos a asignar los permisos a los usuarios creados:

* *revoke all privileges on all tables in schema public from usuario1*
* *revoke all privileges on all tables in schema public from usuario2*
* *grant select, insert, update, delete on all tables in schema public to usuario1*
* *grant select, insert, update, delete on all tables in schema public to usuario2*

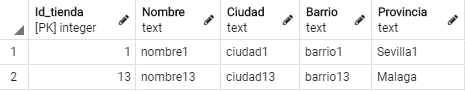
Con esto, ya tendríamos dos usuarios con permisos de lectura/escritura en nuestra base de datos Tienda.

Cuestión 8: Abrir una transacción que inserte una nueva tienda en la base de datos (NO cierre la transacción). Realizar una consulta SQL para mostrar todas las tiendas de la base de datos dentro de esa transacción. Consultar la información sobre lo que se encuentra actualmente activo en el sistema. ¿Qué conclusiones se pueden extraer?

Para abrir una transacción que inserte una nueva tienda, pero sin cerrarla, tenemos que usar BEGIN y no cerrarla con un COMMIT o haciendo un ROLLBACK.



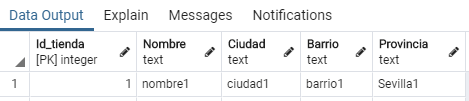
Tras realizar la transacción, obtenemos la siguiente salida:



Como podemos observar, vemos que hay dos tuplas en total en la tabla Tienda, la que hemos insertado ahora en la transacción y la que insertamos en la cuestión 2.

Sin embargo, si ahora mostramos todas las tiendas de la base de datos fuera de la transacción, nos encontramos con lo siguiente:





Vemos que al no haber realizado un COMMIT, la tupla con la nueva tienda no se ha insertado aún, sino que se está esperando a que la transacción, que sigue abierta, haga un COMMIT que implique que los datos se han comprometido y finalmente se inserte la tupla o, por el contrario, realice un ROLLBACK y aborte la transacción.

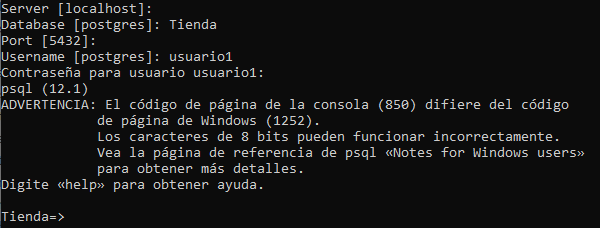
Cuestión 9: Cierre la transacción anterior. Utilizando pgAdmin o psql, abrir una transacción T1 en el usuario1 que realice las siguientes operaciones sobre la base de datos **TIENDA**. NO termine la transacción. Simplemente:

* Inserte una nueva tienda con ID\_TIENDA 1000.
* Inserte un trabajador de la tienda anterior.
* Inserte un nuevo ticket del trabajador anterior con número 54321.

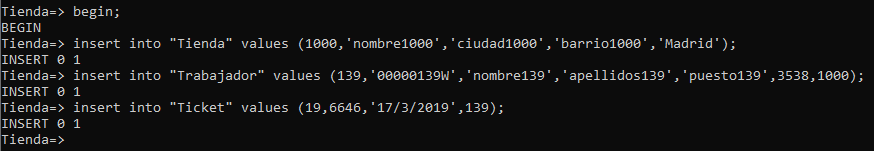
Para realizar esta cuestión utilizaremos la consola de postgres llamada “psql”.

Para poder entrar con el usuario1, psql nos pide la contraseña del usuario por lo que, en el pgadmin le asignaremos una contraseña y ya podremos entrar indicando la base de datos, el usuario y la contraseña:





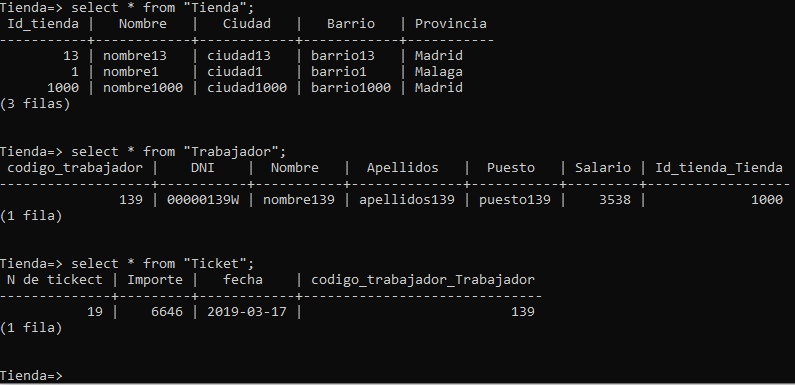
Con esto ya habríamos iniciado una conexión a la base de datos con nuestro usuario1. Una vez hemos hecho esto, realizaremos la transacción sin cerrarla:



Como se puede ver en la imagen anterior, en la transacción hemos insertado una Tienda con id 1000, un trabajador de esa tienda y un ticket generado por ese trabajador.

Cuestión 10: Realizar cualquier consulta SQL que muestre los datos anteriores insertados para ver que todo está correcto.

Sin salirnos de la conexión creada anteriormente, seleccionamos los datos que hay en las tablas Tienda, Trabajador y Ticket y podemos ver que todo está correcto, aparecen todos los datos insertados anteriormente.

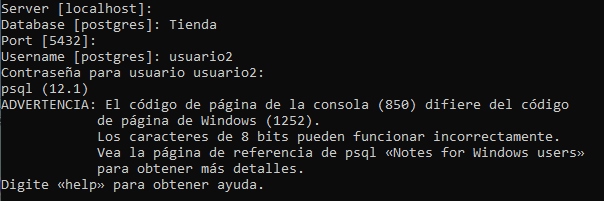


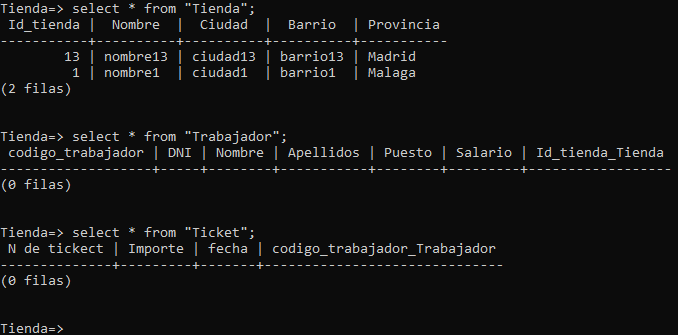
Cuestión 11: Establecer una **nueva conexión** con pgAdmin o psql a la base de datos con el usuario2 (abrir otra sesión diferente a la abierta actualmente que pertenezca al usuario2) y realizar la misma consulta. ¿Se nota algún cambio? En caso afirmativo, ¿a qué puede ser debido el diferente funcionamiento en la base de datos para ambas consultas? ¿Qué información de actividad hay registrada en la base de datos en este momento?

Para poder acceder al psql con el usuario2, necesitamos asignarle una contraseña:



Una vez hecho esto, podremos iniciar una nueva conexión con psql a la base de datos Tienda con el usuario2 y procedemos a realizar las mismas consultas que en la cuestión anterior:



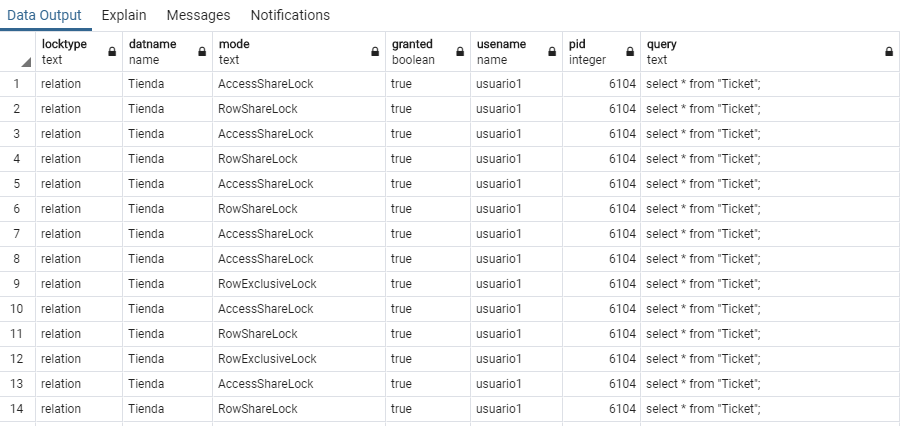


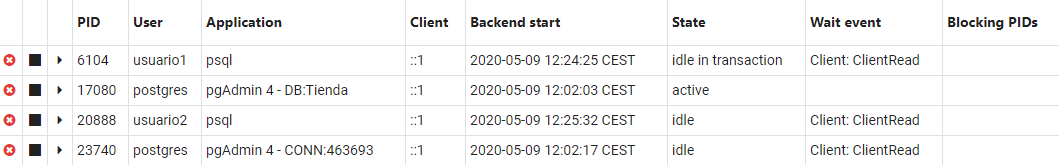
Como podemos observar, al obtener los datos de las tablas Tienda, Trabajador y Ticket desde el usuario2, los datos insertados por el usuario1 en la cuestión anterior no se encuentran introducidos.

Esto es lógicamente porque la transacción iniciada desde el usuario1 no ha sido cerrada, por tanto, los datos aún no se encuentran insertados físicamente en la base de datos.

Dichos datos por tanto se insertarán cuando se cierre la transacción con un commit, es decir, hasta que el usuario1 no realice un commit y cierre la transacción, el usuario2 no podrá ver los datos introducidos en la base de datos ya que estos no se han comprometido.

Ahora pasamos a ver la actividad de la base de datos:





Como podemos observar, el usuario1 ha adquirido bloqueos en la tabla Tienda, esto es ya que la transacción que ha abierto el usuario1 sigue abierta al no haberse realizado un commit o rollback y se crean bloqueos para que no se produzcan inconsistencias en la base de datos.

Cuestión 12: ¿Se encuentran los nuevos datos físicamente en las tablas de la base de datos? Entonces, ¿de dónde se obtienen los datos de la cuestión 2.10 y/o de la 2.11?

No, los datos aún no se encuentran físicamente en la base de datos.

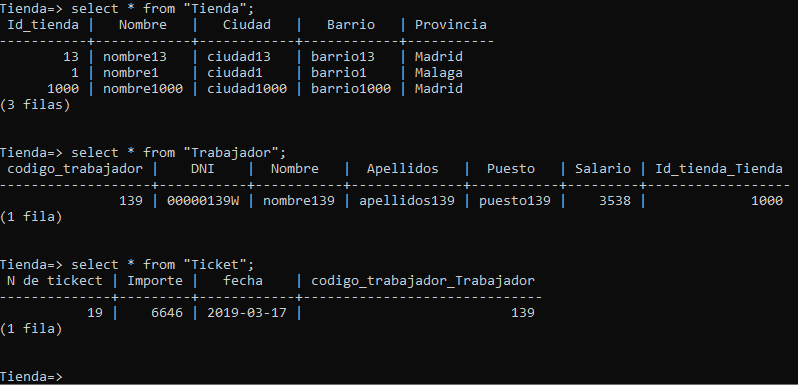
Esto se debe a que la transacción aún no ha sido cerrada, por tanto, si hacemos un select dentro de la propia transacción (como hemos hecho en la cuestión 2.10), podemos ver que los datos se han introducido correctamente, pero esto se debe a que estamos dentro de la transacción que ha insertado los datos.

Sin embargo, en la sesión iniciada por el usuario2, no se ven los datos insertados por el usuario1 y esto se debe a que al ser una nueva conexión y no pertenecer a la transacción que ha insertado los datos y que sigue abierta, físicamente los datos no han sido introducidos aún en la base de datos porque la transacción no ha sido cerrada con un COMMIT, lo que quiere decir que los datos no se han comprometido.

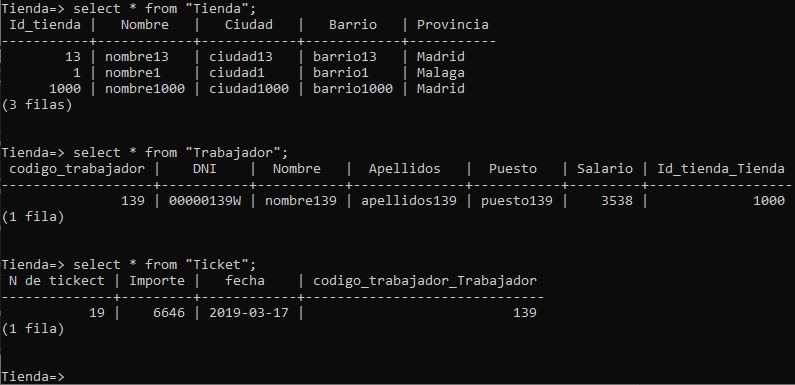
Cuestión 13: Finalizar con éxito la transacción T1 y realizar la consulta de la cuestión 2.10 y 2.11 sobre ambos usuarios conectados. ¿Qué es lo que se obtiene ahora? ¿Por qué?

Realizamos el COMMIT para cerrar la transacción que ha abierto el usuario1. Veamos qué aparece ahora si realizamos una lectura de las tablas con el usuario1 y con el usuario2.

**Usuario1:**



**Usuario2:**



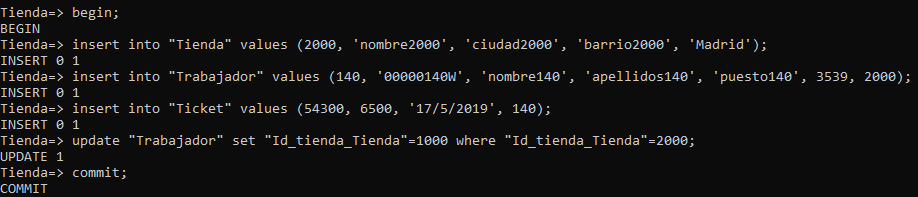
Ahora que la transacción ha sido cerrada mediante un commit desde la conexión abierta por el usuario1, podemos observar que desde el usuario2 ya se pueden leer los datos que introdujo el usuario1. Esto es obviamente porque al haber cerrado la transacción, los datos se han introducido correctamente dentro de la base de datos, y están disponibles para todos los usuarios.

Cuestión 14: Sin ninguna transacción en curso, abrir una transacción en un usuario cualquiera y realizar las siguientes operaciones:

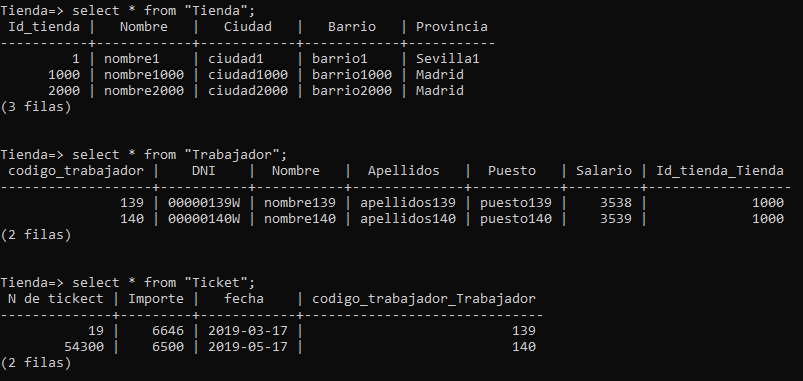
* Insertar una tienda nueva con ID\_TIENDA a 2000.
* Insertar un trabajador de la tienda 2000.
* Insertar un ticket del trabajador anterior con número 54300.
* Hacer una modificación del trabajador para cambiar el número de tienda de 2000 a 1000.
* Cerrar la transacción.

¿Cuál es el estado final de la base de datos? ¿Por qué?

Nos conectamos con el usuario1, abrimos la transacción y realizamos las operaciones indicadas:

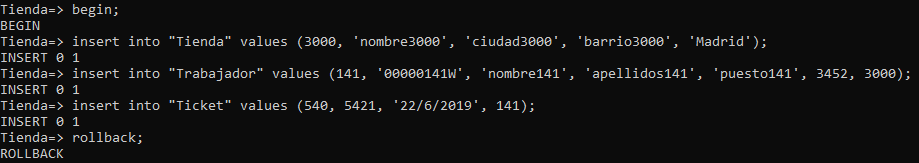


Realizamos una selección que nos muestre cómo están los datos en la base de datos:

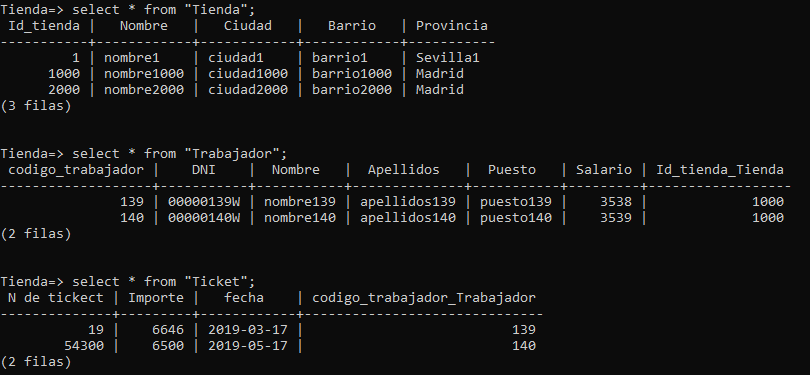


Como podemos ver, se han insertado y actualizado todos los datos dentro de la transacción. Además, el trabajador insertado, que pertenecía inicialmente a la tienda con identificador 2000, tras hacer el update, vemos que ahora pertenece a la tienda con identificador 1000.

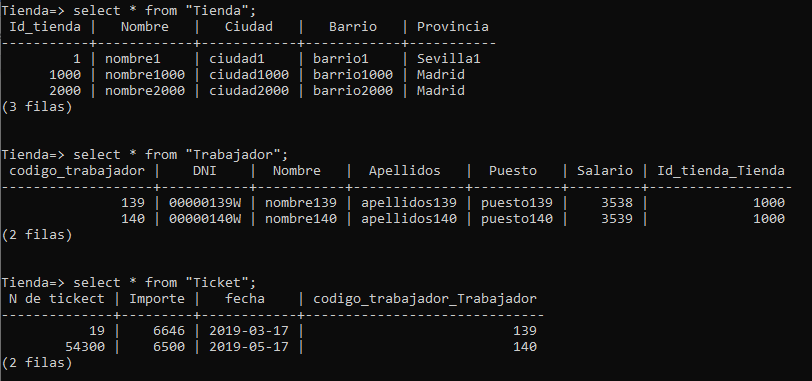
Cuestión 15: Repetir la cuestión 9 con otra tienda, trabajador y ticket. Realizar la misma consulta de la cuestión 10, pero ahora terminar la transacción con un ROLLBACK y repetir la consulta con los mismos dos usuarios. ¿Cuál es el resultado? ¿Por qué?



**Usuario1**

****

**Usuario2**

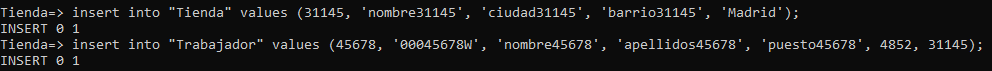
****

Como podemos ver, al hacer un ROLLBACK, no se ha modificado nada en la base de datos, es decir, ninguna de las tuplas que hemos insertado se pueden ver ni desde la conexión del usuario1 ni desde la conexión del usuario2.

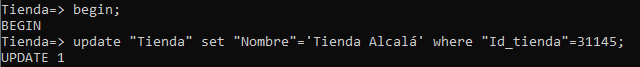
Esto se debe a que el ROLLBACK es un comando que sirve para deshacer los cambios que se han realizado en una transacción, por lo que todas las inserciones realizadas en la transacción se han deshecho, no insertándose nada en la base de datos.

Cuestión 16: Cerrar todas las sesiones anteriores. Abrir una sesión con el usuario1 de la base de datos **TIENDA**. Insertar la siguiente información en la base de datos:

* Insertar una tienda con id\_tienda de 31145.
* Insertar un trabajador que pertenezca a la tienda anterior y tenga un código de 45678.



Cuestión 17: Abrir una sesión con el usuario2 a la base de datos **TIENDA**. Abrir una transacción T2 en este usuario2 y realizar una modificación de la tienda código 31145 para cambiar el nombre a “Tienda Alcalá”. ¿Qué actividad hay registrada en la base de datos? ¿Cuál es la información guardada en la base de datos? ¿Por qué?



Veamos qué actividad hay registrada en la base de datos:





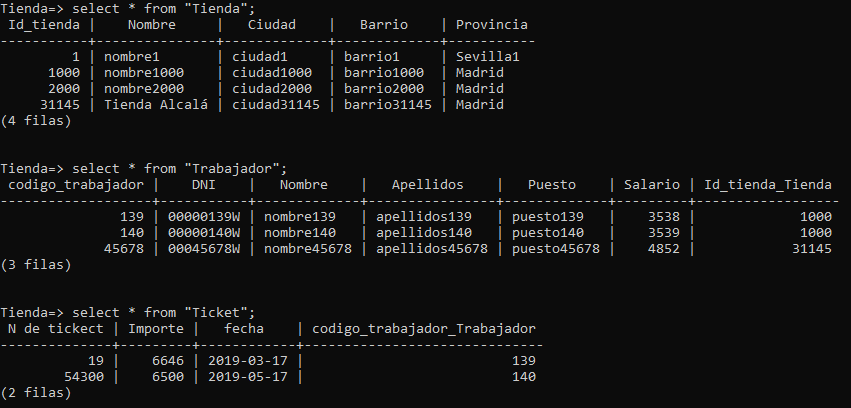
Podemos ver que la transacción del usuario2 está aún abierta y que ha realizado una modificación en Tienda para cambiar el nombre. Además, podemos ver que los dos usuarios se han conectado desde el psql y que el usuario1 se conectó para hacer una o varias inserciones en la base de datos, pero que no tiene ninguna transacción en curso.

Veamos qué información hay ahora guardada en la base de datos:

**Usuario1**



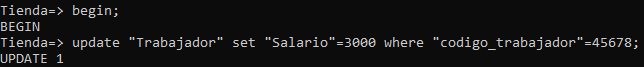
**Usuario2**



Como vemos, al no haberse cerrado la transacción T2 con un commit en el usuario2, los datos no se han comprometido y, por tanto, los cambios realizados dentro de la transacción no quedarán reflejados en el usuario1. Sin embargo, en la conexión del usuario2, al leer los datos que se encuentran en las tablas de la base de datos sí que aparece la modificación realizada sobre el nombre de la tienda.

También vemos que las dos operaciones de inserción realizadas en la cuestión anterior se encuentran físicamente en la base de datos y los dos usuarios pueden ver los datos.

Cuestión 18. Abra una transacción T1 en el usuario1. Haga una actualización del trabajador con número 45678 para cambiar el salario a 3000. ¿Qué actividad hay registrada en la base de datos? ¿Cuál es la información guardada en la base de datos? ¿Por qué?



Veamos qué actividad hay registrada en la base de datos:

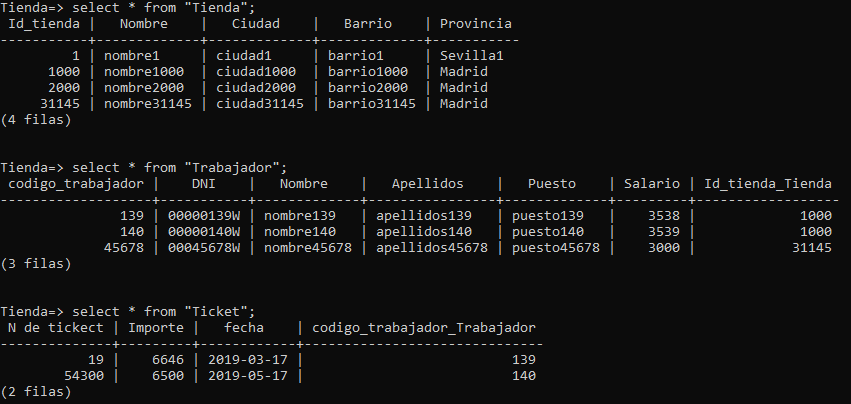




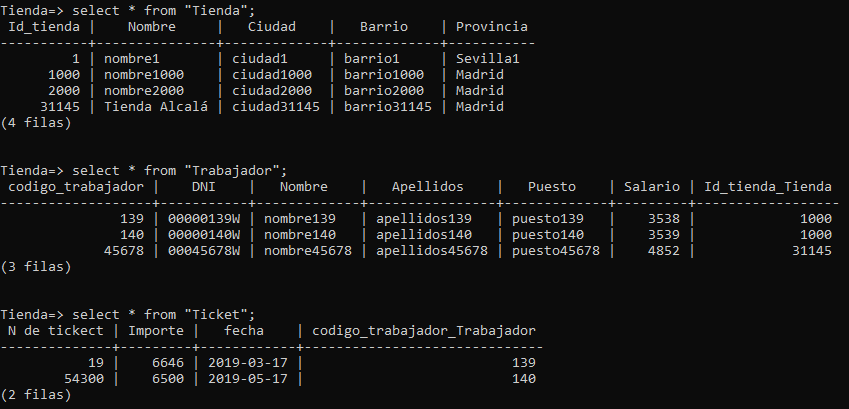
Vemos que ahora, tanto el usuario1 como el usuario2 se encuentran activos en psql *(iddle in transaction)* y realizando una transacción que aún no ha sido cerrada. Ahora, vemos que el usuario1 lo que está haciendo es una modificación en la tabla Trabajador, mientras que el usuario2 lo último que hizo fue leer sobre la tabla Ticket.

Como las dos transacciones siguen abiertas y no se han cerrado o no se ha producido ni un COMMIT ni un ROLLBACK, la información guardada en la base de datos será la misma que antes ya que aún no se ha insertado ni actualizado nada.

**Usuario1**

****

**Usuario2**



Como podemos observar, el usuario2 no puede ver que el usuario1 en la transacción que ha abierto ha modificado el salario del trabajador con código 45678 porque el usuario1 aún no ha realizado un commit, por lo que sigue teniendo la transacción abierta. Sin embargo, el usuario1, dentro de su transacción puede ver los cambios que hasta el momento ha realizado dentro de su transacción por lo que puede ver que ha modificado el salario del trabajador con código 45678 de 4.852 a 3.000.

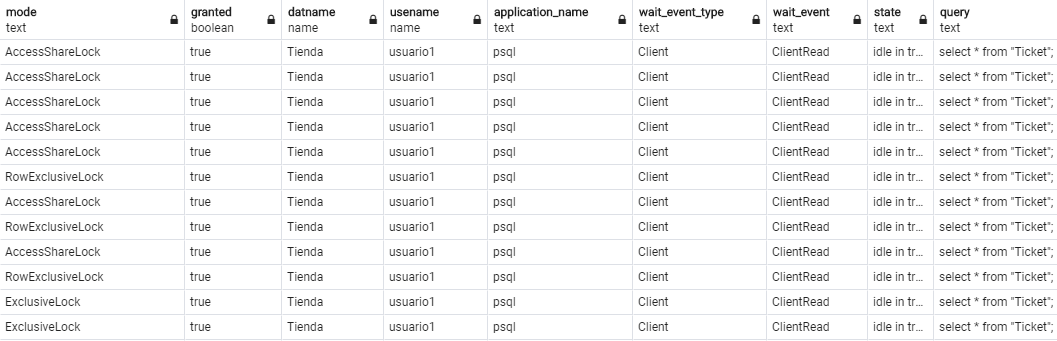
Cuestión 19: En la transacción T2, realice una modificación del trabajador con código 45678 para cambiar el puesto a “Capataz”. ¿Qué actividad hay registrada en la base de datos? ¿Cuál es la información guardada en la base de datos? ¿Por qué?



Como podemos ver, tras realizar una modificación del trabajador con el código 45678 para asignarla ahora el puesto de Capataz, vemos que se ha quedado bloqueado o en espera y no nos responde con un UPDATE 1 como en casos anteriores.

Si miramos la actividad que hay ahora registrada en la base de datos nos encontramos con lo siguiente:







Por una parte, vemos que en la columna *wait\_event\_type*, el usuario1 tiene el valor “Client”, además la columna *state* tiene el valor “idle in transaction”, esto quiere decir que la T1 está en curso.

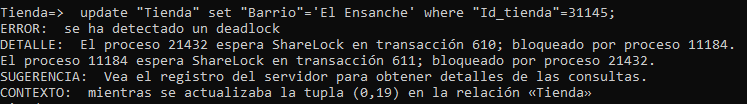
Por otra parte, vemos que en la columna *wait\_event\_type*, el usuario2 tiene el valor “Lock”, además la columna state tiene el valor “active”, esto quiere decir que el usuario2 tiene un bloqueo y no puede realizar la actualización hasta que la T1 del usuario1 termine.

Por tanto, la consulta realizada anteriormente en la T2 del usuario2 está en espera a que la T1 del usuario1 se ejecute y finalice.

La información guardada en la base de datos es la misma que anteriormente ya que el update de la T2 no se ha realizado, está a la espera de que finalice la T1 y la T1 no ha hecho ninguna modificación sobre ninguna tabla de la base de datos.

Cuestión 20: En la transacción T1, realice una modificación de la tienda con código 31145 para modificar el barrio y poner “El Ensanche”. ¿Qué actividad hay registrada en la base de datos? ¿Cuál es la información guardada en la base de datos? ¿Por qué?

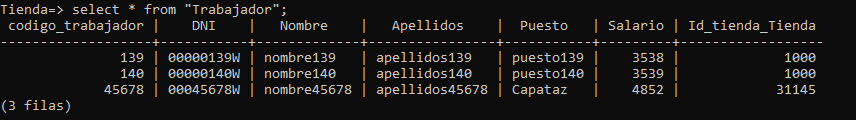
Al intentar realizar una modificación de la tienda 31145 para modificar el barrio, se produce un error ya que se ha detectado un deadlock. Este interbloqueo se debe a que se genera en el sistema un ciclo de transacciones en espera, es decir, la T2 estaba esperando a que la T1 terminase y al realizar esta consulta, la T1 espera a que la T2 termine de ejecutarse, por lo que se genera un interbloqueo que es detectado por PostgreSQL y para resolverlo, automáticamente deshace la transacción 1.



Como podemos observar, la transacción 2 sigue abierta y ejecutándose, por lo que una vez que postgres ha deshecho la transacción 1, esta ya es capaz de adquirir un bloqueo y puede modificar la tupla de Trabajador con código 45678.



En la base de datos seguirá habiendo lo mismo que antes, aunque ahora dentro de la T2 del usuario2, el usuario2 podrá ver como se ha modificado la tupla con el puesto cambiado a Capataz. Aún así, al no haberse realizado un COMMIT, este cambio no estará reflejado en la base de datos.



Cuestión 21: Comprometa ambas transacciones T1 y T2. ¿Cuál es el valor final de la información modificada en la base de datos **TIENDA**? ¿Por qué?

Vamos a comprometer ambas transacciones con el comando COMMIT:

**Usuario1**



**Usuario2**

****

Como podemos ver, en la T1 al hacer el COMMIT se ha producido un ROLLBACK, esto ha pasado debido a que como en la anterior cuestión se ha producido un interbloqueo, PostgreSQL ha decidido deshacer esta transacción por lo que, al deshacer la transacción, se debe de deshacer todos los cambios realizados dentro de la transacción.

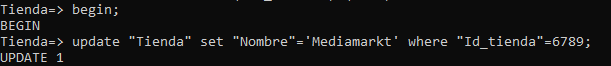
Por otra parte, en la T2 al hacer el COMMIT, la transacción se ha comprometido y todas las inserciones y modificaciones realizadas en la transacción se han volcado a la base de datos.

Si miramos ahora, cuál es el valor final de la información en la base de datos tenemos lo siguiente:

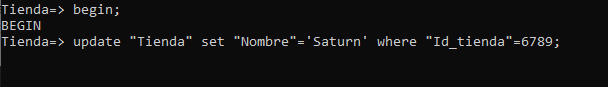


Se ha modificado el nombre de la tienda con identificador 31145 y se ha modificado el puesto del trabajador con código 45678. Sin embargo, no se ha modificado el salario del trabajador con código 45678, ni se ha modificado el barrio de la tienda con identificador 31145 al haber realizado estos cambios la T1 y haberse producido un ROLLBACK debido al interbloqueo producido entre T1 y T2.

Cuestión 22: Cerrar todas las sesiones anteriores. Abrir una sesión con el usuario1 de la base de datos **TIENDA**. Insertar en la tabla tienda una nueva tienda con código 6789. Abrir una transacción T1 en este usuario y realizar una modificación de la tienda con código 6789 y actualizar el nombre a “Mediamarkt”. No cierre la transacción.

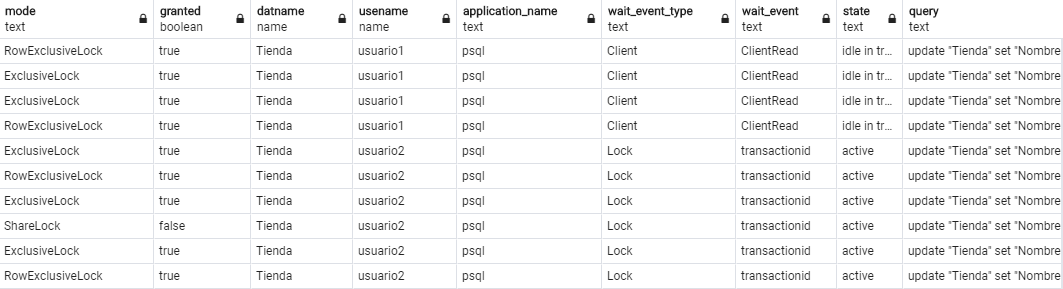


Cuestión 23: Abrir una sesión con el usuario2 de la base de datos **TIENDA**. Abrir una transacción T2 en este usuario y realizar una modificación de la tienda con código 6789 y cambiar el nombre a “Saturn”. No cierre la transacción. ¿Qué es lo que ocurre? ¿Por qué? ¿Qué información se puede obtener de la actividad de ambas transacciones en el sistema? ¿Es lógica esa información? ¿Por qué?



De nuevo, la transacción se queda en espera de que la transacción 1 termine. La T1 tiene concedido un bloqueo sobre la tabla Tienda y aún no ha cerrado la transacción por lo que la T2 está en espera a que la T1 termine para adquirir el bloqueo y poder insertar la tupla.

Si miramos la actividad de ambas transacciones en el sistema:



Vemos que la T1 está abierta y se está ejecutando mientras que la T2 tiene un Lock en estado activo y está esperando a que la T1 se cierre o aborte para que el sistema le conceda los bloqueos en la tabla Tienda para poder actualizar la tupla de la tienda con identificador 6789.

Si miramos la información que tiene la base de datos, vemos que no ha cambiado nada. Es lógico ya que la T1 ha actualizado la tupla de la tabla Tienda con identificador 6789 pero no ha comprometido la transacción con un commit por lo que los cambios aún no se han modificado en la base de datos y por otra parte, la T2 sigue aún activa pero en espera de que la T1 termine.



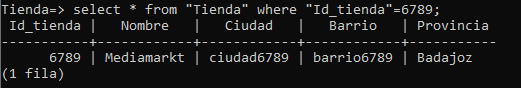
Cuestión 24: Comprometa la transacción T1, ¿Qué es lo que ocurre? ¿Por qué? ¿Cuál es el estado final de la información de la tienda con código 6789 para ambos usuarios? ¿Por qué?

Al comprometer la transacción T1 vemos que automáticamente se actualiza la tupla que estaba en espera de la T2. Esto pasa ya que la T1 termina de ejecutarse y por tanto se libera el bloqueo concedido a esta y pasa a ejecutarse T2.

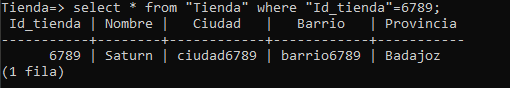




La información final de la tienda con código 6789 para el usuario1 será la siguiente:



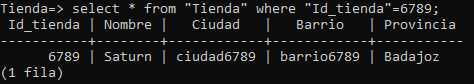
En este caso, la T1 se ha comprometido y por tanto ha cambiado el nombre de la tienda con código 6789 a Mediamarkt como podemos observar. Sin embargo, vemos cuál es el estado de esta misma tienda para el usuario2:

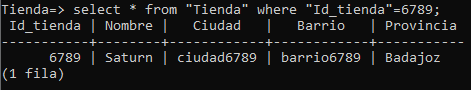


Tras el COMMIT de T1, se realiza la modificación que estaba en espera de la T2, por lo que una vez que leemos los datos dentro de la T2 se muestran los actualizados por la misma. Sin embargo, el usuario1 aún no puede ver que se han modificado los datos por la T2 al no haberse comprometido aún.

Cuestión 25: Comprometa la transacción T2, ¿Qué es lo que ocurre? ¿Por qué? ¿Cuál es el estado final de la información de la tienda con código 6789? ¿Por qué?





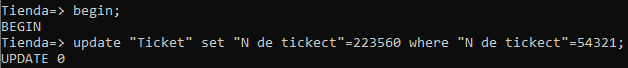


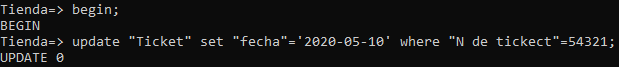
Como se puede observar, al cerrar la transacción 2, tanto el usuario1 como el usuario2 ven ya los datos que hay físicamente en la base de datos ya que no hay ninguna transacción abierta y las dos que habían se han comprometido.

Como T2 realizó un cambio y se comprometió después de hacerlo T1, T2 modifica el nombre “Mediamarkt” que lo modificó el usuario1 con la T1, por “Saturn”.

Finalmente, al comprometerse T2, los dos usuarios pueden ver la última actualización que es la que ha hecho el usuario2 y en la que ha cambiado el nombre que tiene la tienda con identificador 6789 por “Saturn”.

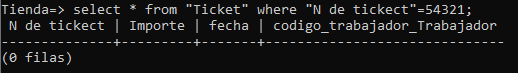
Cuestión 26: Cerrar todas las sesiones anteriores. Abrir una sesión con el usuario1 de la base de datos **TIENDA**. Abrir una transacción T1 en este usuario y realizar una modificación del ticket con número 54321 para cambiar su código a 223560. Abra otro usuario diferente del anterior y realice una transacción T2 que cambie la fecha del ticket con número 54321 a la fecha actual. No cierre la transacción.

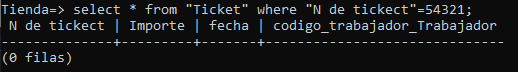




Cuestión 27: Comprometa la transacción T1, ¿Qué es lo que ocurre? ¿Por qué? ¿Cuál es el estado de la información del ticket con código 54321 para ambos usuarios? ¿Por qué?

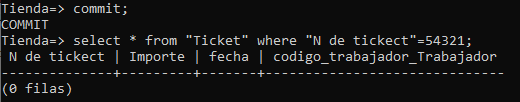


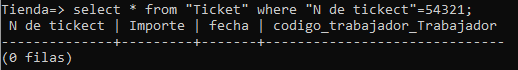




Como vemos, tanto en la sesión que ha abierto el usuario1 como en la sesión que ha abierto el usuario2, al leer la información que hay sobre el ticket con número 54321 en la tabla Ticket, vemos que no hay ninguna tupla. Esto es debido a que no existe ningún ticket en la base de datos con ese número y por tanto, si nos fijamos en la cuestión anterior, al hacer la modificación tanto en una transacción como en la otra, vemos que nos devuelve un “UPDATE 0”, esto quiere decir que se han actualizado 0 tuplas. Obviamente, no se ha actualizado ninguna tupla al hacer esa consulta porque no hay ningún registro en la base de datos, concretamente en la tabla Ticket, con número de ticket 54321.

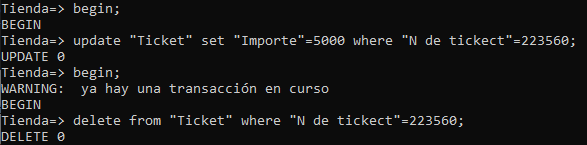
Cuestión 28: Comprometa la transacción T2, ¿Qué es lo que ocurre? ¿Por qué? ¿Cuál es el estado final de la información del ticket con número 54321 para ambos usuarios? ¿Por qué?





Al igual que en la cuestión anterior, vemos que no hay ninguna fila con N de tickect = 54321. Esto es debido, como hemos dicho antes, a que no existía en la tabla Ticket, ningún ticket con ese número, por tanto, al actualizar en la T2 la fecha de ese ticket, no se ha actualizado nada porque no se ha encontrado el ticket con el número especificado, por tanto, al comprometerse la transacción, no se ha registrado ningún cambio en la base de datos.

Cuestión 29: ¿Qué es lo que ocurre en el sistema gestor de base de datos si dentro de una transacción que cambia el importe del ticket con número 223560 se abre otra transacción que borre dicho ticket? ¿Por qué?



Como vemos, al intentar abrir una transacción dentro de otra, psql muestra un mensaje de WARNING en el que advierte que ya hay una transacción en curso.

Como podemos ver, no se nos ha indicado que la transacción ha sido abortada, por tanto, la transacción sigue en curso y podemos seguir haciendo consultas y operaciones dentro de ella. Esto es debido a que Postgres utiliza bloques de transacciones en vez de transacciones autónomas.

Sin embargo, PostgreSQL implementa un comando denominado SAVEPOINT, el SAVEPOINT establece un nuevo punto de guardado dentro de la transacción actual.

Un SAVEPOINT es una marca especial dentro de una transacción que permite restaurar el estado de la transacción a lo que era justo en el momento del SAVEPOINT.

Para volver al punto de guardado se utiliza ROLLBACK TO SAVEPOINT y para destruir un punto de guardado manteniendo los cambios que se han hecho después de establecerse, se usa RELEASE SAVEPOINT.

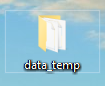
Los SAVEPOINT solo se pueden establecer dentro de un bloque de transacciones. Puede haber múltiples SAVEPOINT definidos dentro de una transacción.

Cuestión 30: Suponer que se produce una pérdida del cluster de datos y se procede a restaurar la instancia de la base de datos del punto 6. Realizar solamente la restauración (recovery) mediante el procedimiento descrito en el apartado 25.3 del manual (versión, 12) *"Continous Archiving and point-in-time recovery (PITR)*. ¿Cuál es el estado final de la base de datos? ¿Por qué?

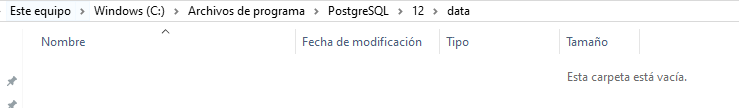
Lo primero que debemos de hacer es detener el servidor de PostgreSQL:



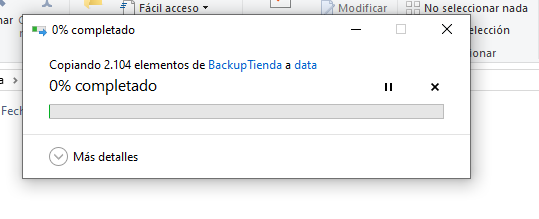
Como tenemos suficiente espacio en nuestro sistema, copiaremos todo lo que tenemos ahora en el directorio de datos, en una carpeta temporal (data\_temp), por si lo necesitamos más tarde.



Ahora, eliminamos todos los archivos y subdirectorios existentes en el directorio de datos del clúster.

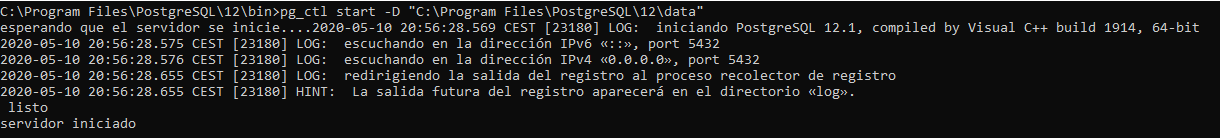


Una vez borrados todos los archivos y subdirectorios del directorio de datos, restauraremos los archivos de la copia de seguridad de la base de datos:

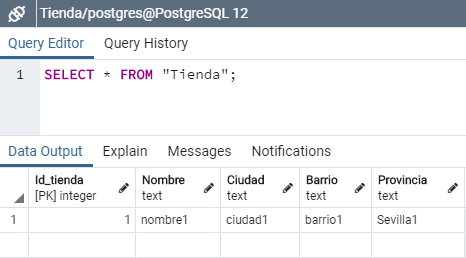


Creamos un archivo en el directorio de datos del clúster llamado *recovery.signal*.

Por último, arrancamos el servidor y comprobamos que todo está correcto en la base de datos y que hemos vuelto al estado en el que hicimos la copia de seguridad de la base de datos Tienda, que fue en la cuestión 5.



En el momento que hicimos la copia de seguridad de la base de datos, solo habíamos añadido una tupla en la tabla Tienda. Si comprobamos qué es lo que tiene la tabla Tienda, vemos que efectivamente se ha realizado bien la recuperación de la copia de seguridad:



Cuestión 31: A la vista de los resultados obtenidos en las cuestiones anteriores, ¿Qué tipo de sistema de recuperación tiene implementado postgreSQL? ¿Qué protocolo de gestión de la concurrencia tiene implementado? ¿Por qué? ¿Genera siempre planificaciones secuenciables? ¿Genera siempre planificaciones recuperables? ¿Tiene rollbacks en cascada? Justificar las respuestas.

PostgreSQL implementa un sistema de recuperación REDO. Postgres utiliza un registro de escritura anticipada (WAL) para garantizar la integridad de los datos. El concepto de WAL es que los cambios en los archivos de datos deben escribirse solo después de que esos cambios se hayan registrado, es decir, cuando los registros del log que describen los cambios se hayan vaciado al almacenamiento permanente. Si seguimos este procedimiento, no es necesario que vacíe las páginas de datos al disco en cada COMMIT porque sabemos que en caso de bloqueo podremos recuperar la base de datos utilizando el registro. Cualquier cambio que no haya sido aplicado a las páginas de datos puede rehacerse desde el registro de logs.

En cuanto al control de la concurrencia, PostgreSQL utiliza un mecanismo llamado Multiversion Concurrency Control (MVCC).

Este mecanismo hace que cada instrucción SQL vea una versión de la base de datos que no está actualizada, es decir, una versión de la base de datos antigua, sin tener en cuenta el estado actual de los datos. Esto consigue prevenir que las instrucciones SQL vean datos inconsistentes producidos por transacciones actuales que cambian, añaden o borran tuplas. Esto proporciona una transacción única por cada sesión de la base de datos.

Postgres implementa una herramienta que es el aislamiento de transacciones. El estándar SQL define cuatro niveles de aislamiento de transacciones: Read uncommitted, Read committed, Repeatable read y Serializable. Cada uno de los niveles tiene de menos a más secuenciabilidad, siendo el nivel Serializable el de mayor secuenciabilidad ejcutándose las transacciones de una en una, en algún orden.

Por otra parte, las planificaciones que genera PostgreSQL son recuperables ya que no se vacía al almacenamiento permanente los cambios hasta que se haya realizado un COMMIT. Es decir, es recuperable ya que las transacciones solo pueden leer un dato escrito de otra transacción cuando esta ya se ha comprometido.

PostgreSQL no implementa el ROLLBACK en cascada ya que en caso de que hubiese un fallo en una transacción esto no da lugar a que las demás transacciones se tienen que deshacer si no que se deshace únicamente la transacción que ha tenido el fallo.

**Bibliografía**

* Capítulo 13: Concurrency Control.
* Capítulo 25: Backup and Restore.
* Capítulo 27: Monitoring Database Activity.
* Capítulo 29: Reliability and the Write-Ahead log.