Introducción a las transacciones

Tiempo estimado: 30min

Una vez sabemos ejecutar comandos de lectura y escritura contra una base de datos, es indispensable conocer cómo aglutinar varios comandos en una única unidad lógica de trabajo o procesamiento, la transacción. Esta lección es agnóstica al motor de bases de datos. La siguiente lección es la específica del motor de bases de datos específica del curso. La idea es que primero hay que conocer los conceptos relacionados con las transacciones para, a continuación, abordar más detenidamente qué proporciona nuestro sistema de bases de datos en estudio. Esto ayudará al estudiante a elegir el mejor producto para cada proyecto.

La lección comienza con una introducción al concepto de transacción y varias clasificaciones atendiendo a distintas maneras de verlas. A continuación, se presenta la demarcación de las transacciones, cómo delimitar las operaciones que forman una transacción. Le sigue los programas y los niveles de aislamiento. Mediante los programas se permite el intercalado de las operaciones de las transacciones con objeto de permitir ejecutar más o menos transacciones concurrentes. Mientras que los niveles de aislamiento determinan qué datos modificados puede ver una transacción. Finalmente, se presenta el concepto de registro de transacciones.

Al finalizar la lección, el estudiante sabrá:

- Qué es una transacción y qué tipos de transacciones hay.
- Cómo se delimita las operaciones que forman las transacciones.
- Qué es un programa y su importancia dentro del nivel de concurrencia de la base de datos.
- Cuáles son los niveles de aislamiento más frecuentes.
- Qué es el registro de transacciones y qué información contiene.

Introducción

Una transacción (*transaction*) es una unidad lógica de trabajo. Una unidad atómica formada por una secuencia indivisible de uno o más comandos que llevan a cabo una determinada tarea en conjunto. El concepto de transacción es fundamental e indispensable para controlar la ejecución concurrente de comandos de distintos o el mismo usuario y para la recuperación ante fallos.

La mayoría de los motores de bases de datos SQL y NoSQL implementan las transacciones haciéndolas cumplir con lo que se conoce formalmente como propiedades ACID (*ACID properties*). Un conjunto de aspectos que el motor garantiza cumplirá toda transacción. Debe su nombre a las iniciales de las cuatro propiedades en inglés:

- *Atomicity* (atomicidad). Las operaciones o comandos que forman la transacción deben ejecutarse completamente. No se permite que al finalizar se haya aplicado los cambios realizados por unas operaciones y otras no. O todos los cambios o ninguno.

  Así pues, si la transacción finaliza con éxito, el motor de bases de datos garantiza que todos sus cambios se habrán aplicado. En cambio, si finaliza en fallo, el motor de bases de datos garantiza que cualquier cambio que se haya realizado durante la transacción será deshecho dejando los datos modificados como estaban antes de comenzar la transacción. Es más, si la instancia cae, el motor garantiza que deshará cualquier cambio realizado por cualquier transacción que estuviera sin finalizar antes de la caída, devolviendo así la base de datos a un estado consistente.

  En un motor de bases de datos, del cumplimiento de la atomicidad se encarga el subsistema de recuperación de transacciones. Se consigue básicamente mediante dos conceptos: la demarcación de la transacción y el registro de transacciones. Con la demarcación de la transacción, se delimita el inicio y fin de las transacciones. Mientras que el registro de transacciones se utiliza para registrar cada operación realizada por las transacciones, de tal

manera que cuando una transacción finaliza en fallo, utilizará el registro para deshacer sus cambios, esto es, devolver los datos a sus valores iniciales antes del comienzo de la transacción.

- *Consistency* (consistencia) o *consistency preservation* (conservación de la consistencia). Establece que si la transacción finaliza con éxito, los cambios realizados por la transacción serán consistentes y cumplirán las restricciones de integridad definidas en la base de datos.

  Durante la ejecución de una transacción, se puede relajar el cumplimiento de las restricciones pero, una vez finalizada, todos los cambios cumplirán las restricciones de integridad sin excepciones, dejando así la base de datos en un estado consistente.

  El cumplimiento de esta propiedad se delega en los desarrolladores o en el subsistema de restricciones de integridad. No todos los motores de bases de datos proporcionan un subsistema de restricciones de integridad completo. Algunos ni siquiera lo proporcionan. Cuando un motor no proporciona soporte integrado, se delega en los desarrolladores la responsabilidad de mantenerlas y hacerlas cumplir. Por lo general, los sistemas SQL proporcionan un muy buen soporte de restricciones de integridad, mientras que los NoSQL no lo hacen tan bien.

- *Isolation* (aislamiento). Dicta que la ejecución de varias transacciones concurrentemente tendrá el mismo resultado que si sus operaciones se aplicaran secuencialmente, esto es, primero todas las de una transacción y después las de la otra y así sucesivamente.

  Está relacionado con los datos que la transacción puede ver y lo que otras pueden ver acerca de los cambios realizados por las demás.

  Del aislamiento se encarga el subsistema de control de concurrencia. Aquel que permite la ejecución simultánea de varias transacciones. Para ello, se sirve de bloqueos y del nivel de aislamiento. Para garantizar el aislamiento de la transacción con respecto a las demás, el motor bloquea los datos accedidos por la transacción; cuantos más bloqueos aplique, menor será el nivel de concurrencia en la base de datos.

  Por otro lado, permite configurar el nivel de aislamiento bajo el cual una transacción debe ejecutarse. Cada uno de los cuales indica qué cambios realizados, por las demás transacciones concurrentes, puede ver una determinada transacción.

- *Durability* (durabilidad). Asegura que una vez una transacción ha finalizado y confirmado sus cambios, en caso de fallo de la instancia, no habrá pérdida de datos bajo ningún concepto. Es decir, el motor de bases de datos garantiza que los cambios realizados por una transacción finalizada en éxito no se perderán pase lo que pase.

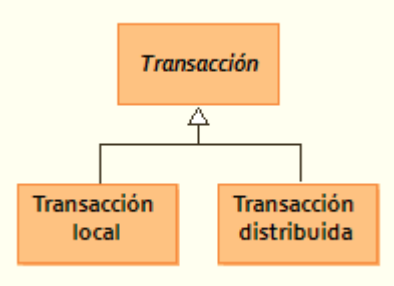
  Del cumplimiento de la durabilidad se encarga el subsistema de recuperación, el cual utiliza principalmente el registro de transacciones.

Visto lo visto, el objetivo de una transacción es realizar una secuencia de operaciones de lectura y/o escritura de manera completamente independiente y cuyos cambios se garantiza se realizarán todos ellos o ninguno, dejando en cualquier caso la base de datos en un estado consistente. Si alguna de las operaciones de la transacción falla, el motor deshará automáticamente todos los cambios realizados hasta ese momento, dejando los datos modificados como estaban antes de iniciarse la transacción. Si la transacción confirma sus cambios, el motor garantizará que todos ellos se aplican.

Del cumplimiento de las propiedades se encarga principalmente el motor de bases de datos. Aunque atendiendo al fabricante, algunas propiedades recaen en manos de los desarrolladores completa o parcialmente. Cuanto menos debe realizar un desarrollador, mejor, menos probabilidad de equivocación habrá y mayor será su productividad.

Transacciones locales y distribuidas

Atendiendo a las bases de datos sobre las que trabaja una misma transacción, distinguimos entre transacciones locales y distribuidas.



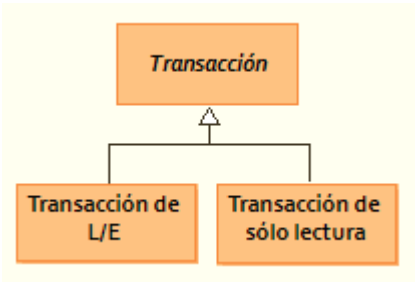
Una transacción local (*local transaction*) afecta o actualiza un único recurso, en nuestro caso, una única base de datos. En cambio, una transacción distribuida (*distributed transaction*) afecta o actualiza varias bases de datos.

Es más fácil controlar las propiedades ACID de una transacción local que de una distribuida. Esto lleva implícito que presenta mejor rendimiento una local que una distribuida. Casi todos los motores soportan el concepto de transacción local. Y pocos el de transacción distribuida.

Groso modo, las aplicaciones interactúan con el administrador de transacciones, siendo éste el que interactúa finalmente con los distintos administradores de recursos de bases de datos involucrados en la transacción. Así pues, se puede definir formalmente el administrador de transacciones (*transaction manager*) como el componente que se encarga de coordinar la transacción entre los distintos recursos (bases de datos) y, por lo tanto, actualizar los datos dentro de la misma transacción. Debe poder comunicarse con todos los servidores y recursos participantes, los cuales se pueden encontrar dentro del mismo dominio o fuera.

Transacciones de lectura/escritura y de sólo lectura

Atendiendo a si la transacción lee o escribe datos, se distingue entre transacciones de lectura/escritura (L/E) y de sólo lectura.



Una transacción de lectura/escritura (*read-write transaction*) es aquella que realiza operaciones tanto de lectura como de escritura de datos. En cambio, una transacción de sólo lectura (*read-only transaction*) es aquella que sólo realiza operaciones de lectura de datos.

Para un motor de bases de datos, es más fácil gestionar una transacción de sólo lectura que una de L/E. Algunos permiten que los usuarios indiquen ante qué tipo de transacción nos encontramos para así ayudar a reducir el uso de recursos, mejorando así el rendimiento. Siempre que sepamos que estamos ante una transacción de sólo lectura, si el motor permite indicárselo, se recomienda encarecidamente hacerlo.

Demarcaciones de las transacciones

Como una transacción es una secuencia indivisible de comandos u operaciones, los motores proporcionan un medio a través del cual indicarlas, o sea, una manera de indicar cuáles son las operaciones que forman la transacción. Básicamente, se distingue entre delimitación explícita e implícita. Un mismo motor puede soportar uno o ambos tipos de demarcación.

Delimitación explícita

Mediante la delimitación explícita (*explicit demarcation*), los usuarios disponen de comandos específicos para marcar expresamente el inicio y el final de las transacciones. Se basa en el concepto

de demarcación de transacción (*transaction demarcation*), fijar sus límites. Un límite (*boundary*) no es más que la marca de inicio o fin de la transacción.

El inicio de transacción (*transaction begin*) indica su comienzo. Todas las operaciones que se lleven a cabo desde ese momento formarán la unidad indivisible. Por lo general, se marca mediante el comando BEGIN TRANSACTION.

El fin de transacción (*transaction end*), por su parte, marca el término de la transacción de usuario en curso. Se distingue entre confirmación y aborto. La confirmación (*commit*) indica que deseamos que los cambios realizados se hagan efectivos, asegurando así que todos los cambios se vuelcan a la base de datos. Si, por alguna razón, el motor no pudiera aplicar todos los cambios, el motor deberá cancelarlos y devolver un error. No puede aplicar algunos cambios y otros no hacerlo. O todos o ninguno. Nada de medias tintas. Recordemos la propiedad de atomicidad.

En cambio, mediante el aborto (*rollback*) lo que hacemos es indicar que deseamos cancelar la transacción y, por ende, todos sus cambios. En este caso, el motor deshará cualquier cambio realizado a la base de datos, asegurando así dejarla como estaba antes del inicio de la transacción.

Por lo general, las confirmaciones se marcan con el comando COMMIT y el aborto con ROLLBACK.

La idea es muy sencilla. Mediante la delimitación explícita lo que se hace es solicitar a los usuarios que marquen el inicio y el fin de las transacciones. El usuario puede finalizar la transacción solicitando que los cambios se apliquen, se confirmen, o bien se descarten, se aborten. Cuando el motor recibe el fin, actuará de una manera u otra. Si se ha solicitado la confirmación de datos, garantizará que todos los datos se almacenan en la base de datos. Si no puede, por la razón que sea, la confirmación acabará en error y, por ende la transacción, y se desharán todos los cambios. Si, por el contrario, se solicita directamente cancelar los cambios, el motor deshará los cambios.

Para un buen funcionamiento, la transacción debe finalizar en éxito, el cual no se mide en si finalizó en confirmación o aborto, sino con la garantía de que se aplicaron todos los cambios o ninguno. Si una confirmación acaba en error, es porque un fallo importante se ha producido al volcarse los datos. Si un aborto acaba en error, es porque un fallo importante se ha producido al deshacer los datos.

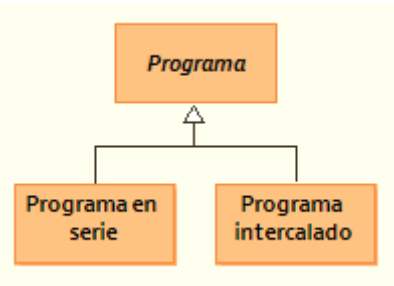
Confirmación implícita o modo autoconfirmación

El modo autoconfirmación (*autocommit mode*) indica que cada comando ejecutado lleva implícito una confirmación de la transacción al finalizarse el comando. Se trata de transacciones con una única operación. Desde este modo, se puede abrir, en cualquier momento, una transacción delimitada explícitamente por BEGIN y COMMIT o ABORT; al finalizarse la transacción delimitada, la sesión vuelve al modo autoconfirmación.

Por ejemplo, en PostgreSQL, las sesiones se abren en modo autoconfirmación de manera predeterminada.

Programas

Un programa (*schedule*) es una manera de ejecutar las operaciones de varias transacciones. Atendiendo a si se puede intercalar las operaciones de las transacciones concurrentes, se distingue entre programas en serie o intercalados.



Un programa en serie (*serial schedule*) es aquel que sólo permite ejecutar una única transacción cada vez. Hasta que no termina la transacción en curso, no se pasa a la siguiente. Por su parte, un programa intercalado (*nonserial schedule*) mezcla las operaciones de distintas transacciones.

Los programas en serie garantizan de manera muy, pero que muy sencilla, las propiedades ACID. Pero

tiene como inconveniente que reduce considerablemente el rendimiento, pues las transacciones no ceden los recursos hasta que han finalizado. Por ejemplo, cuando la transacción ejecuta una operación de E/S a disco, el sistema se bloquea hasta que la E/S ha finalizado. Con los programas intercalados, las operaciones de las distintas transacciones se reparten los recursos del sistema de tal manera que en un momento dado puede estar ejecutando varias transacciones simultáneamente, mejorando así el rendimiento final.

Por desgracia, la ejecución simultánea no es sencilla y el motor debe atender posibles problemas que aparecen debido a esta concurrencia de transacciones. No es fácil de implementar y consume recursos.

Para intentar solventar los problemas que puede aparecer, los motores de bases de datos buscan programas serializables (*serializable schedules*), programas intercalados que simulan programas en serie. Esto significa que entre la enorme cantidad de posibles combinaciones que puede haber para ejecutar varias transacciones, el motor prefiere aquellos cuyo resultado final sea equivalente a ejecutar las transacciones como si se ejecutasen una detrás de otra, sin intercalar operaciones, pero haciéndolo.

Por lo general, los motores SQL soportan programas intercalados, permitiendo la ejecución concurrente de transacciones. Otros motores, generalmente NoSQL como por ejemplo Redis, sólo soportan los programas en serie. En el caso de Redis, esto es posible porque es un sistema de bases de datos en memoria y las operaciones de E/S a disco son contadas. En vez de complicar su implementación, proporcionando el servicio de concurrencia de transacciones, lo que hace es anteponer el rendimiento. Al ser más fácil y no acceder a disco, puede permitirse el lujo de permitir únicamente los planes en serie, obteniendo un rendimiento similar al de un sistema con soporte para planes intercalados.

Nivel de aislamiento

Mediante el nivel de aislamiento (*isolation level*) fijamos a qué datos modificados puede acceder una transacción. Permite o impide que los datos modificados por otras transacciones concurrentes puedan ser vistos por la transacción. El motor aplica bloqueos a los datos accedidos para impedir así su acceso a otras transacciones. Cuanto mayor es el nivel de aislamiento, mayor número de objetos serán bloqueados, reduciéndose así el nivel de concurrencia de transacciones.

Cuanto más bajo es el nivel de aislamiento, mayor cantidad de datos se puede consultar y, por lo tanto, incrementa el nivel de concurrencia de las transacciones. Pero huelga decir que incrementa también la posibilidad de efectos colaterales como, por ejemplo, las lecturas sucias o las actualizaciones perdidas. Digamos que menos probabilidad hay de que dos o más transacciones se bloqueen entre sí.

En cambio, cuanto mayor es el nivel de aislamiento, mayor es el número de recursos invertidos por el motor para conseguir el aislamiento, reduciéndose además el nivel de concurrencia, porque las transacciones tendrán que esperar a que los datos que desean acceder se desbloqueen para así poder usarlos.

A continuación, se presenta los principales niveles de aislamiento proporcionados por los motores SQL, base para la mayoría de los motores de bases de datos. La presentación se hace de mayor nivel de aislamiento a menor y, por lo tanto, de menor concurrencia a mayor.

Nivel de aislamiento serializable

Bajo el nivel de aislamiento serializable (*serializable isolation level*), el más estricto de todos, todo dato accedido por la transacción queda bloqueado desde su primer acceso hasta el final de la transacción. Así, por ejemplo, todas las filas restringidas por las consultas mediante cláusulas WHERE serán bloqueadas hasta finalizar, aun si no son accedidas más a lo largo de la transacción. De esta manera, impide que otras transacciones puedan modificarlos y, así, la transacción siempre verá los datos que hay en la base de datos desde que comenzó hasta el final, obviamente con sus correspondientes modificaciones realizadas por la propia transacción.

Este tipo de aislamiento tiene como objeto simular un comportamiento de programa en serie, recordemos, como si cada transacción se ejecutara de inicio a fin sin otras transacciones concurrentes y sin interrupciones.

Por suerte, los motores de bases de datos cada día son más óptimos y pueden implementar el aislamiento aplicando el menor número de bloqueos, en algunos casos, incluso ninguno. En este caso, el motor supervisará todas las transacciones y podrá detectar, sin temor a equivocaciones, que dos o más

transacciones han realizado operaciones conflictivas entre sí. En este caso, el motor elegirá una y finalizará con error las demás, garantizando así el nivel de aislamiento requerido. Gracias a este funcionamiento, el nivel de concurrencia crece considerablemente.

Nivel de aislamiento con lecturas repetibles

Cuando ejecutamos una transacción con un nivel de aislamiento con lecturas repetibles (*repeatable read isolation level*), se sigue impidiendo que dos transacciones puedan modificar el mismo dato. Una vez un dato es modificado por una transacción, el resto no puede acceder a él, porque contiene un dato que no es el que ellos habrían visto si fueran los únicos que están ejecutándose en el sistema.

En este nivel de aislamiento, se puede producir lecturas fantasmas (*phantom reads*), esto es, una transacción puede llegar a ver una nueva fila insertada por otra transacción y utilizar su contenido, pero no puede ver modificaciones realizadas por UPDATEs de otras filas, ni le desaparecerá ninguna fila suprimida por un DELETE.

El problema de la lectura fantasma se comprende más fácilmente mediante un ejemplo. Supongamos dos transacciones concurrentes X1 y X2. La transacción X1 realiza una consulta contra la tabla T, la cual le devuelve, digamos 10 filas. A continuación, el motor ejecuta una inserción de la transacción X2 sobre la misma tabla T. Por circunstancias de la vida, la transacción X1 vuelve a ejecutar la misma consulta anterior contra la tabla T, pero ahora en vez de recibir 10 filas, recibe 11. Esta nueva fila se considera una fila fantasma, pues ha aparecido en la transacción sin venir a cuento.

Nivel de aislamiento de lecturas confirmadas

Cuando una transacción se ejecuta bajo el nivel de aislamiento de lecturas confirmadas (*read committed isolation level*), la transacción podrá ver todos los datos confirmados, desde el momento de su confirmación. Analicemos las posibles instrucciones DML con este nivel de aislamiento. Para ello, supongamos, por un lado, dos transacciones X1 bajo cualquier nivel de aislamiento y X2 bajo nivel de aislamiento de lecturas confirmadas y, por otro lado, supondremos que X1 finalizará siempre antes que X2, ambas ejecutándose en paralelo:

- Si la transacción X1 realiza una inserción y finaliza, X2 podrá ver la nueva fila desde el momento de la confirmación de X1.
- Si la transacción X1 realiza una actualización y finaliza, X2 podrá ver el dato modificado desde el momento de la confirmación de X1.
- Si la transacción X1 realiza una supresión y finaliza, X2 dejará de ver los datos suprimidos desde el momento de la confirmación de X1.

Como la transacción con nivel de aislamiento READ COMMITTED puede ver datos confirmados por otras transacciones desde su confirmación, este tipo de transacción puede ejecutar la misma consulta y recibir dos resultados distintos. Por esta razón, este nivel de aislamiento *no* se debe utilizar si es necesario que una misma consulta, ejecutada varias veces durante la transacción, debe devolver siempre los mismos resultados, a menos que la diferencia de resultado se deba a modificaciones realizadas por la propia transacción.

Registros de transacciones

El registro de transacciones (*transaction log*), también conocido como registro de operaciones (*journaling*), es un diario ordenado en el que la instancia almacena las operaciones de las distintas transacciones que ha ejecutado. Se trata de un componente crítico del sistema, porque ayuda a recuperar la base de datos en caso de fallo o desastre.

Está formado por uno o más archivos y es de vital importancia que se encuentren en discos rápidos, pues posiblemente son los más accedidos del sistema de bases de datos. Durante un funcionamiento normal de la instancia, los archivos son de sólo añadir (*append-only*). A los archivos sólo se accede para añadir entradas. Sólo cuando algo va mal, se acceden en modo lectura.

Tiene como objeto básicamente:

- Poder restaurar el estado consistente de la base de datos en caso de fallo.

  Si las operaciones ejecutadas por la instancia se encuentran registradas, aquellas cuyos datos

no se volcaron a los archivos de datos, se podrán reejecutar y así conseguir el estado que tenía la base de datos en el momento del fallo.

- Deshacer las transacciones que finalmente se aborten.

  Si las operaciones ejecutadas por una transacción, que acaba siendo abortada por el usuario, se encuentran en el registro, el motor puede utilizarlo para deshacer sus cambios. Bastará con recorrer el registro, extraer las operaciones de la transacción y deshacer los cambios.

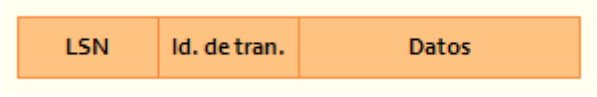
Al disponer de toda la información en el registro, la instancia puede rehacer los cambios de transacciones confirmadas y deshacer las operaciones de las transacciones no confirmadas.

Para ayudar a distinguir unas transacciones de otras, los motores de bases de datos les asignan un identificador de transacción (*transaction id*), un valor único utilizado para distinguirla de manera inequívoca de las demás. Lo fija automáticamente en el momento de comenzar su ejecución.

Entrada del registro de transacciones

El registro de transacciones está formado por registros, cada uno de los cuales se conoce formalmente como entrada del registro de transacciones (*transaction-log record*). Básicamente, cada una de estas entradas contiene:

- Un identificador único para distinguirla de las demás, conocido formalmente como número de secuencia del registro (*log sequence number*, LSN). Lo fija automáticamente el motor en el momento de su ejecución.
- El identificador de la transacción a la que pertenece. Con este valor, el motor puede consultar las operaciones de una transacción.
- La operación realizada y los cambios realizados. Con esta información, podrá deshacer el cambio o rehacerlo, según el caso.



El LSN es muy importante. Cada vez que se hace un vuelco de datos a disco, el sistema almacena el LSN de la última operación volcada. Así, cuando arranca, comprueba el LSN de la última entrada del registro y la del último vuelco. Si no son el mismo, el sistema tiene un problema, pues significa que tras la última detención no se volcaron todos los datos a disco. Para resolver el problema, realiza lo que se conoce formalmente como autorrecuperación.

Groso modo, se distingue los siguientes tipos de entrada:

- Inicio de transacción (*transaction start*). Marca el inicio de una nueva transacción.

  Contiene el LSN y el identificador de la transacción que se inicia.
- Fin de transacción (*transaction end*). Marca el fin de una transacción.

  Contiene el LSN y el identificador de la transacción que se finaliza.
- Lectura de datos (*item read*). Representa una operación de acceso en modo lectura a un determinado dato.

  Contiene el LSN, el identificador de la transacción a la que pertenece y el dato accedido.
- Inserción de datos (*item insert*). Representa la inserción de un nuevo dato.

  Contiene el LSN, el identificador de la transacción a la que pertenece y el dato insertado.
- Actualización de datos (*item update*). Representa una operación de modificación.

  Contiene el LSN, el identificador de la transacción, el valor anterior al cambio y el nuevo valor.
- Supresión de datos (*item delete*). Representa una supresión de datos.

  Contiene el LSN, el identificador de la transacción y los valores suprimidos.

Autorrecuperación

La autorrecuperación (*autorecovery*) es una acción implícita que realiza el motor cuando arranca y detecta que la instancia no se cerró formalmente. Lo que se produce cuando se produce un fallo que impide hacerlo así, como por ejemplo un corte eléctrico o un fallo interno de la instancia.

Es una operación automática. Cuando la instancia arranca de nuevo, detecta esta situación y realiza la operación. Para ello, se sirve del registro de transacciones para rehacer todas aquellas operaciones que pertenecen a transacciones confirmadas y que no se volcaron a disco. Y por otra parte, deshacer todas aquellas operaciones de transacciones que en el momento de la detención no habían finalizado.

En caso de fallo o desastre, cuando se realiza una restauración, los LSNs definen la cadena de restauración (*restore chain*), o sea, determinarán el orden en que debe aplicarse las operaciones de las entradas del registro, primero las más viejas hasta llegar a las más recientes, con objeto de dejar la base de datos en un estado consistente.

Búfer del registro de transacciones

El búfer del registro de transacciones (*transaction-log buffer*) es un búfer que contiene las últimas entradas del registro de transacciones. El nivel de importancia del registro de transacciones es enorme. Cualquier entrada no volcada en el archivo del registro, no se podrá utilizar en las recuperaciones del sistema. Las palabras se las lleva el viento y cuando no se encuentran por escrito, más todavía. Eso significa que cada vez que se realiza una operación, se debe crear su entrada y volcar de inmediato a disco. El problema es que la E/S a disco es el principal cuello de botella de los sistemas de bases de datos.

Para mejorar el rendimiento, lo que hace la mayoría de fabricantes es almacenar temporalmente las últimas entradas en la memoria, en el búfer del registro de transacciones. Periódicamente, realizan vuelcos de su contenido a disco. Así, en vez de volcar una entrada cada vez, se vuelca varias, mejorándose enormemente el rendimiento. Cuanto mayor sea el búfer, mayor número de entradas se podrá almacenar en esta caché interna y mejor será el rendimiento.

Pero no hay que olvidar una cosa, si se cae el sistema, las entradas no volcadas, se perderán. Para resolver este pequeño problema, pero de vital importancia, los motores vuelcan el contenido de este búfer con mucha más frecuencia que los demás datos cacheados. Es más, cuando una transacción finaliza, el sistema se asegura de volcar las entradas. El búfer es muy bueno para mantener las entradas de las operaciones no finalizadas todavía. Pero tan pronto como una transacción finaliza, todo su contenido se vuelca para asegurar la propiedad de durabilidad de las transacciones. Si no se volcase, no se podría asegurar.

