Transacciones en InterBase® y Firebird (0.2.6)

Juan José Rodríguez
Cuenca, España
jrodriguez@kinobi.cjb.net

4 de julio de 2002

Resumen

Este documento presenta una visión general de las transacciones en Sistemas Gestores de Bases de Datos, y en particular para los servidores InterBase®[1] y Firebird[2].

Además de revisar los conceptos básicos de las transacciones, se describen los parámetros que afectan al comportamiento de éstas en los servidores antes citados. También se exponen las funciones disponibles a través del Interfaz de Programación de Aplicaciones (API) para que el desarrollador pueda aprovechar toda la potencia de las transacciones. Se describen características propias de los servidores InterBase y Firebird en las que las transacciones desarrollan un papel destacado, caso de la Arquitectura Multi-Generacional (MGA), el commit en dos fases. Por último se presentan las transacciones desde el punto de vista del desarrollador Delphi, a través del componente TIBTransaction que ofrece la biblioteca InterBase eXpress.

Este documento va dirigido a desarrolladores que hayan utilizado motores de datos que no dispongan de mecanismos de control transaccional, o aún disponiéndolos, y no conocen la implementación y funcionamiento que en InterBase tiene este mecanismo. En especial está orientado a desarrolladores que utilicen herramientas Borland Delphi[3] y Borland C++ Builder[4], aunque no exclusivamente.

A lo largo del documento se utilizará el término InterBase para referirse indistintamente a InterBase y a Firebird.

[1] **InterBase**® es una marca registrada por **Borland Software Corporation**. A partir de este punto, siempre que nos refiramos a InterBase en realidad lo estamos haciendo a InterBase®.

[2] **Firebird** es un proyecto Open Source, bajo licencia *InterBase Public License*, basado en el código fuente de InterBase® 6.0 liberado por Borland Software Corporation el 25 de Julio de 2000.

[3] **Delphi** es una marca registrada por **Borland Software Corporation**.

[4] **C++ Builder** es una marca registrada por **Borland Software Corporation**.

Índice

1. Introducción a las transacciones

1.1. Transacciones: qué son, para qué sirven y cómo se usan

Como[5] primera aproximación, podríamos definir una transacción como una ***unidad de trabajo[1]*** (o procesamiento) que se desarrolla dentro de un ***contexto atómico***. Es una unidad de trabajo porque el servidor la gestiona como un todo, aunque pueda estar constituida por varias operaciones individuales (inserciones, modificaciones o eliminaciones), contra una o varias bases de datos. Se desarrolla dentro de un contexto atómico, de tal forma que todas las operaciones involucradas en la transacción se confirman, o rechazan, todas en bloque de manera indivisible.

La función básica de las transacciones es garantizar que las operaciones que realizamos contra nuestra(s) base(s) de datos no ponen en peligro la ***coherencia*** y ***consistencia*** de la información que almacenamos en ella(s).

Profundizando en la definición inicial, podemos decir que los gestores de bases de datos deben garantizar cuatro propiedades básicas en las transacciones, conocidas como propiedades **ACID**[2]:

***A*tomicidad:** Todas las operaciones de una transacción deben ser tratadas como una única unidad; todas ellas, o ninguna, deben ser ejecutadas.

***C*onsistencia:** Una transacción debe llevar la base de datos desde un estado consistente a otro estado consistente.

***I*solation (Aislamiento)**: Cada transacción se ejecuta como si fuese la única transacción de la base de datos.

***D*urabilidad:** El resultado de una transacción nunca se pierde si ésta finaliza correctamente.

Las transferencias bancarias son un ejemplo típico para poner de manifiesto la razón de ser de las transacciones. Supongamos que somos clientes de un banco donde tenemos abierta una cuenta corriente y una cuenta de ahorros. Acabamos de recibir el ingreso de nuestra paga extra en la cuenta corriente y decidimos transferir parte de la paga a la cuenta de ahorros; pongamos 500 Euros[6]. Tras dar la orden oportuna al operario del banco, éste procede a retirar los 500 Euros de la cuenta corriente (nuestro saldo se reduce en esa cantidad en esa cuenta), e inicia la transferencia de fondos a la cuenta de ahorros. Pero, ¡oh, cielos!, en ese preciso instante, justo antes de que la transferencia se lleve a cabo, sucede el desastre. El sistema informático de nuestro banco sufre un repentino corte de energía, a la vez que nuestro ritmo cardíaco aumenta alarmantemente. Nuestros 500 Euros han "volado" de nuestra cuenta corriente, pero nunca han llegado a nuestra cuenta de ahorros[7]. En este punto es donde sale en nuestra ayuda el mecanismo transaccional. En realidad, nuestros 500 Euros nunca han llegado a "volar"

[5] El título de esta sección es un pequeño homenaje al primer micro-ordenador que adquirí (en realidad lo compró mi padre), el mítico ***Sinclair ZX Spectrum 48K***. Junto con el ordenador se regalaba un pequeño libro que enseñaba diversos trucos y técnicas para utilizar en el *ZX Spectrum*. El título del libro era (es) "*ZX Spectrum. Qué es, para qué sirve y cómo se usa*", de Dr. Tim Langdell. Corría el año 1984, coincidiendo con otro título famoso, el de la novela de Orwell.

[6] Desde el día 1 de Enero de 2002, el Euro es la moneda común de doce paises de la Unión Europea, entre ellos España.

[7] Cierto, también podría haber ocurrido al revés: primero incrementar el saldo de la cuenta de ahorros y después decrementar el saldo de la cuenta corriente. Entonces la taquicardia la tendría el operario, pero ... ¿realmente supone que los bancos trabajan así?.

de nuestra cuenta corriente, al menos lógicamente, ya que la transacción en la que está involucrada la transferencia de fondos de una cuenta a otra nunca llegó a finalizar. El decremento en 500 Euros del saldo de nuestra cuenta corriente está reflejado en la base de datos, pero no está confirmado. Cuando el sistema se recupere, el operario podrá continuar la operación donde se había interrumpido, transferir los 500 Euros a nuestra cuenta de ahorros, o bien cancelar todas las operaciones anteriores e iniciar una nueva transferencia. En todo caso, la base de datos siempre se ha mantenido en un estado consistente. Nuestros 500 Euros siempre han estado ahí, incluso cuando la transferencia de fondos se interrumpió a la mitad.

Cómo puede garantizar el servidor la consistencia de la base de datos; es decir, cómo puede asegurar el mecanismo transaccional que un fallo en el sistema no comprometerá la coherencia de la información que almacena la base de datos. Básicamente existen dos vías para lograr este objetivo: podría utilizar un mecanismo de registro de operaciones, o ***bitácora***, donde la transacción registra cada uno de los cambios que lleva a cabo contra la base de datos. Este registro puede hacerse en paralelo con los cambios reales en la base de datos, de tal forma que ante un fallo del sistema todos los cambios puedan deshacerse siguiendo el rastro de las operaciones reflejadas en la *bitácora*, o bien registrar todos los cambios sólo en la *bitácora* y confirmarlos posteriormente en la base de datos. La alternativa a la bitácora es el mecanismo conocido como ***multiversión***[8]. Aunque profundizaremos en la *multiversión* posteriormente, adelantemos que éste es el método que utiliza InterBase y se basa en generar diferentes versiones de los registros modificados, asociando cada una de estas versiones a la transacción que las modifica. El papel que juegan las transacciones en este método para garantizar la consistencia de la base de datos, así como la concurrencia de múltiples usuarios, es vital.

El uso de las transacciones para el desarrollador de aplicaciones es extremadamente simple. Se comienza abriendo la transacción; a continuación se aplican los cambios (uno o múltiples) que desee contra la base de datos (una o varias), y por último se confirman (***commit***) o rechazan (***rollback***) en bloque los cambios aplicados en el paso anterior.

En este punto un comentario: si usted es un desarrollador Delphi que ya ha trabajado contra servidores InterBase a través del Borland Database Engine, o de bibliotecas de componentes tipo InterBase eXpress, InterBase Objects, etc., puede que no se sienta identificado con el método de uso de las transacciones expuesto anteriormente, ya que usted no precisa abrir la transacción, ni tampoco confirmarla o rechazarla. Bien, es cierto, ésto es debido a que estos componentes pueden encargarse de gestionar implícitamente las transacciones por usted, aunque también le dejan abierta la posibilidad de gestionarlas explícitamente.

Un segundo comentario: toda operación contra una (o varias) base(s) de datos InterBase **siempre** está bajo el control de una transacción, incluidas las operaciones de lectura. De no ser así, tras una operación de escritura no se podría garantizar el estado consistente de la(s) base(s) de datos, como hemos visto en el ejemplo de la transferencia bancaria. En el caso de las operaciones de lectura no podríamos garantizar cierto tipo de accesos a los datos, caso de las lecturas serializables. Este último asunto lo abordermos posteriormente.

[8] También conocido como ***versionado de registro***, o como se conoce en InterBase, ***Arquitectura Multi-Generacional*** (MGA).

1.2. Parámetros de una transacción

Esta es la sintaxis para la declaración (apertura) de una transacción en SQL:

```
SET TRANSACTION [NAME <nombre_transacción>]
  [READ WRITE | READ ONLY]
  [WAIT | NO WAIT]
  [[ISOLATION LEVEL] {SNAPSHOT [TABLE STABILITY]
    | READ COMMITTED [[NO] RECORD_VERSION]}]
  [RESERVING <cláusula_de_reserva>
    | USING manejadorbd [, manejadorbd ...]];
<cláusula_de_reserva> = tabla [, tabla ...]
  [FOR [SHARED | PROTECTED] {READ | WRITE}] [, <cláusula_de_reserva>]
```

Si usted es un desarrollador Delphi que utiliza bibliotecas de componentes como InterBase eXpress para acceder a su servidor InterBase, no intente utilizar la sentencia **`SET TRANSACTION`** para abrir una transacción. Si intenta ejecutar esta sentencia desde un componente **TIBSQL**, o **TIBQuery**[9], sólo logrará que el componente lance una excepción y su correspondiente mensaje de error. Por extensión, herramientas como **IBConsole** o **IBAccess[6]** también darán como respuesta un error si intenta ejecutar esta sentencia desde sus respectivas ventanas de *SQL Interactivo*, ya que estas herramientas están basadas en estos componentes. Tampoco la utilice dentro de procedimientos almacenados o disparadores (*triggers*); el analizador sintáctico del servidor le devolverá un error cuando intente introducirla dentro del cuerpo de cualquiera de estos dos elementos. En la práctica esto significa que **InterBase no soporta el anidamiento de transacciones**[10]. Si utiliza el SQL *embebido*[11] en otros lenguajes, o en herramientas como ***isql***, puede utilizarla sin ningún problema.

Por ahora la sentencia `SET TRANSACTION` nos servirá para presentar los distintos parámetros que definen una transacción e intervienen en su "comportamiento". Estos son los parámetros[12]:

- **Nombre de la transacción**: este parámetro, **`NAME <nombre_transacción>`**, posibilita que desde una misma conexión se puedan manterner simultáneamente varias transacciones activas.
- **Modo de acceso**: dos son las opciones, **`READ WRITE`**, indicando que la transacción tiene permiso de lectura y escritura; y **`READ ONLY`**, que indica que la transacción sólo tiene permiso de lectura.
- **Resolución de bloqueos**: en los accesos concurrentes de varias transacciones a los mismos datos en operaciones de escritura, sólo la primera actualización concluye con éxito, y además bloquea al resto de transacciones de escritura (actualizaciones o eliminaciones) hasta que finaliza, bien confirmando los cambios (*commit*), o rechazándolos (*rollback*). El resto de transacciones pueden esperar a

[9] En realidad TIBQuery lleva un dentro un TIBSQL para ejecutar su sentencia SQL.

[10] Transacciones dentro de transacciones.

[11] Traducción literal de "embebed". No me agrada, pero es la traducción más extendida, por tanto es la que utilizaré.

[12] Gran parte de la información que viene a continuación está basada en la documentación oficial de **InterBase 6.0[3]** y **4.0[4]** (para Linux), así como en un excelente artículo[5] que **Claudio Valderrama** tiene publicado en su página web.

que finalice la transacción que bloquea los datos (parámetro **`WAIT`**), o bien devolver el control inmediatamente a la aplicación cliente informando de la existencia de un conflicto de bloqueo[13] (parámetro **`NO WAIT`**).

- **Nivel de aislamiento** (**`ISOLATION LEVEL`**): el nivel de aislamiento determina que visión de la base de datos tiene la transacción respecto a los cambios que estén provocando en la base de datos el resto de posibles transacciones que se estén ejecutando concurrentemente. Los posibles niveles de aislamiento son:

  - **`READ COMMITED [RECORD_VERSION]`**: este es el nivel más bajo de aislamiento. La transacción tiene acceso sólo a la última versión de los registros modificados y confirmados (*commit*) por el resto de transacciones que se ejecuten concurrentemente con ella.
  - **`READ COMMITED NO RECORD_VERSION`**: La transacción tiene acceso a los registros modificados por otras transacciones que se estén ejecutando concurrentemente. Si los cambios aún no han sido confirmados (*commit*), y la transacción tiene el parámetro `WAIT` para la resolución de bloqueos, esperará hasta que se confirmen (*commit*) o rechacen (*rollback*). Si por el contrario tiene el parámetro `NO WAIT`, devolverá inmediatamente un error a la aplicación cliente. En cualquier caso, una transacción nunca puede tener acceso *real* a registros modificados, pero no confirmados (*commit*); InterBase **no soporta** el llamado nivel de asilamiento READ UNCOMMITED, también conocido como DIRTY READ.
  - **`SNAPSHOT`**: conocido también por *lectura repetible* (repeteable read). Ofrece un nivel de aislamiento alto, de tal forma que la transacción no puede ver los cambios provocados por otras transacciones; es decir, mantiene una *foto fija* del estado de la base de datos en el momento que se inició la transacción.
  - **`SNAPSHOT TABLE STABILITY`**: añade a las características del nivel anterior una más: en el momento en el que la transacción accede a una tabla la bloquea al resto de transacciones, sólo para las operaciones de escritura. Evidentemente debe ser utilizado con sumo cuidado, ya que bloquea tablas completas a otros posibles redactores.

- **`RESERVING <cláusula_de_reserva>`**: las transacciones obtienen el acceso a las tablas de la base de datos en el momento que leen o escriben en ellas. Este parámetro permite garantizar a la transacción que obtendrá unos niveles de acceso específicos, para determinadas tablas, en el mismo momento en el que la transacción comience, sin tener que esperar a un acceso real a esta(s) tabla(s) en cuestión. La cláusula de reserva tiene dos posibles parámetros, además de los nombres de las tablas, para cada conjunto de tablas sobre las que se quiere aplicar la reserva:

  - **Modo de uso**: dos posibilidades, **`SHARED`**, se comparte el acceso a la(s) tabla(s) con otras transacciones concurrentes. La alternativa es **`PROTECTED`**, que no comparte el acceso.
  - **Modo de acceso**: fija los accesos de lectura (**`READ`**) y escritura (**`WRITE`**), en función del parámetro anterior.

[13] O conflicto de actualización.

Los dos parámetros anteriores, y sus cuatro posibles valores, ofrecen las siguientes posibilidades:

- **`SHARED READ`**: cualquier transacción puede leer la(s) tabla(s). Cualquier transacción, con modo de acceso `READ WRITE`, puede actualizar la(s) tabla(s).
- **`SHARED WRITE`**: cualquier transacción con nivel de aislamiento `SNAPSHOT` o `READ COMMITED`, y con modo de acceso `READ WRITE`, puede actualizar la(s) tabla(s). Si tiene modo de acceso `READ ONLY` sólo podrá leer.
- **`PROTECTED WRITE`**: cualquier transacción con nivel de aislamiento `SNAPSHOT` o `READ COMMITED` puede leer la(s) tabla(s). Sólo esta transacción puede actualizar la(s) tabla(s).
- **`PROTECTED READ`**: cualquier transacción puede leer la(s) tabla(s). Se niega el acceso de escritura a todas las transacciones.

- **`USING manejadorbd [, manejadorbd ...]`**: este parámetro limita el número de bases de datos a las que puede acceder una transacción, listándolas a través de los manejadores de base de datos que las representan.

Sería aconsejable crear una pequeña base de datos de pruebas, estableciendo para la transacción por defecto diferentes valores de los parámetros. Para ello puede utilizar la utilidad en línea de comando ***isql***. Si prefiere las herramientas gráficas le recomiendo **IBAccess**[6][14]. Lance varias instancias, bien de *isql* o *IBAccess*, cambie los parámetros de la transacción por defecto en una y otra instancia, realice varias operaciones contra la base de datos, y observe como en función de unos parámetros u otros el resultado obtenido es distinto.

1.3. Funciones del API InterBase

Independientemente al método escogido para el acceso desde sus aplicaciones a su servidor InterBase, todas las peticiones que haga deberán pasar por llamadas a las funciones que brinda el ***Interfaz de Programación de Aplicaciones*** (**API**[15]) de InterBase. Si se conecta a través de bibliotecas de componentes como InterBase eXpress, InterBase Objects, FIB Plus, o a través del Borland Database Engine, o por medio de Borland dbExpress, todos estos componentes encapsulan dentro de sus métodos las oportunas llamadas al API InterBase para poder realizar su trabajo.

Dentro del API InterBase existen funciones de todo tipo: para la gestión de bases de datos, para el manejo de tipos BLOB y array, funciones de conversión, para el manejo y gestión de errores y eventos, para obtener información de diversos objetos de las bases de datos, para la gestión de la seguridad del servidor y sus bases de datos, para controlar el propio servidor, y también, cómo no, para gestionar y manejar las transacciones.

Antes de poder utilizar ninguna de las funciones con las que se gestionan las transacciones, es necesario crear un **TPB**[16]. El TPB es un vector donde se almacenan los

[14] *IBAccess* es una herramienta de administración de bases de datos InterBase. Se la recomiendo además de por su comodidad y potencia, por ser *Open Source* (código fuente incluido), bajo licencia *Mozilla Public License*. Además tiene disponibles versiones para plataformas *Microsoft Windows* y *Linux*. Ver referencias.

[15] En la documentación de InterBase 6, existe un volumen[7] completo dedicado a las funciones del API. Es más que recomendable echarle un vistazo, aunque sea superficial, para comprender mejor el funcionamiento de InterBase desde el punto de vista del desarrollador de aplicaciones.

[16] Transaction Parameter Buffer.

parámetros[17] que van a determinar el comportamiento de la transacción. En realidad no es estrictamente necesario crear un TPB, ya que si no lo hacemos[18], el servidor le asignará uno por defecto, con un conjunto de parámetros predefinidos. En cualquier caso, en el TPB se fijarán los parámetros que hemos visto en la sección anterior y que queremos que determinen el comportamiento de nuestra transacción.

Veamos ahora una lista de las funciones disponibles, junto con una breve descripción:

isc_start_transaction: Arranca una transacción contra una o más bases de datos. Entre otros parámetros recibe un puntero a un vector de estado, un puntero al manejador (handle) de la transacción, puntero[19](s) a los manejadores de la(s) base(s) de datos sobre la(s) que actúa la transacción, y un puntero al TPB que determina los parámetros de la transacción.

isc_start_multiple: Realiza el mismo trabajo que la función anterior. Su existencia está motivada para proporcionar un mecanismo que permita arrancar una transacción contra varias bases de datos desde lenguajes que no soportan el paso de un número variable de argumentos en la llamada a una función.

isc_commit_transaction: Cierra la transacción, confirmando en la base de datos todos los cambios (inserciones, modificaciones y/o eliminaciones) que se hayan realizado durante la misma.

isc_commit_retaining: Confirma en la base de datos todos los cambios que se hayan realizado desde el comienzo de la transacción. Además se crea una nueva transacción que adquiere el contexto[20] de la transacción original. En principio su función es mejorar el rendimiento, eliminado la sobrecarga que supone adquirir los recursos necesarios al iniciar una nueva transacción. Como veremos posteriormente, también tiene su lado negativo, al desactivar la eliminación automática de versiones antiguas de registros dentro del mecanismo de multiversión, que a su vez repercute de forma negativa en el rendimiento del acceso a la base de datos.

isc_rollback_transaction: Cierra la transacción, cancelando en la base de datos todos los cambios que se hayan realizado durante la misma.

isc_rollback_retaining: Realiza el mismo trabajo que *isc_commit_retaining*, salvo que cancela todos los cambios en la base de datos realizados por la transacción desde que se arrancó.

isc_prepare_transaction: Lleva a cabo la primera fase de un *commit en dos fases*. El *commit en dos fases*, que veremos posteriormente, permite asegurar la consistencia de una transacción que implique a varias bases de datos. El *commit en dos fases* es un mecanismo autómatico por defecto, por tanto el uso de esta función implica tener que realizar posteriormente una llamada a *isc_commit_transaction* para llevar a cabo la segunda fase del cierre de la transaccion y la confirmación de los cambios *(commit*).

[17] Ver sección 1.2

[18] O no lo hacen los componentes que utilicemos para acceder al servidor InterBase.

[19] Al ser un número variable de bases de datos, es necesario que el lenguaje desde el que se hace la llamada a la función permita el paso de un número variable de argumentos a la misma, por ejemplo el lenguaje C.

[20] Los recursos que el servidor destina para mantener la transacción.

isc_prepare_transaction2: Realiza el mismo trabajo que la función anterior, pero además recibe como parámetro un cadena de caracteres que se escribe en la columna RDB$TRANSACTION_DESCRITION, de la fila que corresponda a la transacción en la tabla del sistema RDB$TRANSACTIONS[21]. En la práctica ésto significa la desactivación del mecanismo automático de recuperación de un *commit en dos fases* que no finalice correctamente, quedando en manos del desarrollador dicha recuperación.

1.4. TIP, OAT y OIT. Estados de una transacción

Detrás de las tres siglas que dan título a esta sección, se esconden tres elementos muy importantes en toda base de datos InterBase, relacionados todos ellos con las transacciones y que tienen gran importancia en asuntos como la *Arquitectura Multi-Generacional*.

TIP: *Transaction Inventory Page(s)* (Página(s) Inventario de Transacciones). Toda base de datos InterBase se constituye[8] de páginas de tamaño fijo para cada uno de los archivos que forman la base de datos. Estas páginas son de diversas clases, en función del tipo de información que almacenan. Existe una página de cabecera, que determina información muy importante de la base de datos, como su *On-Disk Structure* (ODS). También existen páginas que almacenan información sobre los índices que se utilizan en las tablas de la base de datos, otras sobre los generadores, el espacio libre, ..., y también sobre las transacciones, son las páginas TIP. Una página TIP consiste en una cabecera que almacena el tipo de la página y un puntero a la siguiente TIP. A partir de esta cabecera una página TIP mantiene un vector que almacena los identificadores de las transacciones, junto con el **estado** en el que se encuentran. Una transacción InterBase puede estar en uno de cuatro posibles estados:

- **Activa:** La transacción todavía está abierta, pendiente de ser confirmada (*commit*) o rechazada (*rollback*).
- **Commited:** Confirmada. La transacción está terminada y los cambios que haya podido provocar en la base de datos han sido confirmados.
- **Rolled_back:** Cancelada (o rechazada). La transacción está terminada y los cambios que haya podido provocar en la base de datos han sido rechazados.
- **Limbo:** La transacción ha completado la primera fase de un *commit en dos fases*[22] y aún no está terminada.

OAT: *Oldest Active Transaction* (Transacción Activa más antigua). Es la transacción con el identificador más bajo y con estado "*activa*" dentro de las páginas TIP.

OIT: *Oldest Interesting Transaction* (Transacción Interesante más antigua). Es la transacción con el identificador más bajo y estado diferente a "*confirmada*" (*commited*) dentro de las páginas TIP.

Los valores actuales para la OAT y OIT pueden consultarse utilizando la utilidad ***gstat***, pasándole el parámetro «***h***». Ejemplo:

[21] Ver sección 2.4
[22] Ver sección 2.4 en la página 15

```
kinobi@aviles:$ gstat -h /home/kinobi/firebird/MiBase.gdb -u SYSDBA -p masterkey
Database "/home/kinobi/firebird/MiBase.gdb"
Database header page information:
        Flags                   0
        Checksum                12345
        Generation              17500
        Page size               8192
        ODS version             10.0
        Oldest transaction      17496
        Oldest active           17497
        Oldest snapshot         17497
        Next transaction        17498
        Bumped transaction      1
        Sequence number         0
        Next attachment ID      0
        Implementation ID       16
        Shadow count            0
        Page buffers            0
        Next header page        0
        Database dialect        3
        Creation date           Nov 15, 2001 12:09:53
        Attributes
```

La OAT se corresponde con el parámetro «*Oldest active*» (17497) y la OIT con «*Oldest transaction*» (17496), en este caso.

2. Concurrencia y otros aspectos

2.1. Arquitectura Multi-Generacional

En entornos de bases de datos, el control de la concurrencia de múltiples usuarios a los mismos datos se ha resuelto tradicionalmente con mecanismos de ***bloqueos***[9]. Los bloqueos se basan en establecer una prohibición de acceso (bien de escritura, o de escritura y lectura) sobre la información que queremos proteger. El bloqueo puede hacerse a varios niveles: a nivel de ***tabla***, el menos selectivo, ya que bloquea todos los registros de la tabla; a nivel de ***página***, bloqueando varias filas de una tabla, todas las que entren en la(s) página(s) bloqueada(s); a nivel de ***registro***, el más selectivo y menos restrictivo de todos. El momento en que se establece el bloqueo determina su tipo: ***pesimista***, si se establece en el mismo momento que accedemos a los datos a proteger; ***optimista***, si se establece en el momento que realmente realizamos los cambios sobre los datos a proteger. Los bloqueos han sido, y son, utilizados por muchos gestores de bases de datos con éxito.

InterBase adopta[23] otro enfoque para enfrentarse a la situación. Este enfoque es el denominado ***multiversión*** *de registro*, llamado pomposamente "***Arquitectura Multi-Generacional***" (MGA) por Borland. La *multiversión[10]* se basa en generar nuevas versiones de los registros "tocados" por cada operación de modificación o eliminación.

[23] InterBase no es el primer gestor de bases de datos que lo utiliza.

Cada vez que se inserta un registro en una base de datos InterBase se almacena, junto con los datos, una referencia a la transacción a la que está asociada la operación de escritura. A su vez, cada transacción que tiene lugar en la base de datos, sea de escritura o lectura, se asocia con un identificador, un número entero. Este identificador sigue una secuencia ascendente, de tal forma que las transacciones más antiguas tienen identificadores más bajos que las más recientes. Cuando se modifica un registro, se genera una nueva versión del registro asociada a la transacción que lo ha modificado. Esta nueva versión mantiene un enlace[24] a la versión antigua del registro y, recíprocamente, la versión antigua mantiene un enlace a la nueva, formando una especie de cadena. Cuando se genera la nueva versión del registro, se compara con la versión antigua para generar un bloque con las diferencias encontradas, denominado **BDR**[25]. Este registro con las diferencias entre versiones se almacena en una nueva localización en la base de datos, y la versión nueva del registro se sitúa en el lugar que ocupada la antigua versión. De esta forma se garantiza que el versionado de registros sólo afectará a los campos que se han cambiado realmente, de tal forma que no se almacena por cada versión una copia completa del registro, sólo de los campos que hayan sido cambiados por la operación de actualización. En las eliminaciones, el BDR almacena la versión antigua del registro, mientras que la nueva versión simplemente almacena la referencia a la transacción que eliminó el registro y una marca de borrado.

La ventaja fundamental del mecanismo de *multiversión* es que permite ”proteger” un registro sin necesidad de bloquear explícitamente el acceso al mismo. Los redactores no impiden el acceso a los lectores, y a otros redactores sólo lo hacen en el caso de cambios en el registro. Veamos un ejemplo:

Supongamos que hemos abierto una transacción con identificador 100 (desde ahora *t100*), con la intención de modificar (*update*) un registro. Otro usuario, antes de aplicar nuestros cambios al registro, inicia una transacción (*t101*) con el mismo objeto. Este usuario es más rápido y aplica sus cambios antes de que lo hagamos nosotros; su transacción (*t101*) se termina y queda confirmada. En la práctica esto significa que la versión más moderna del registro tiene el identificador de la transacción *t101*. Posteriormente, nosotros intentamos aplicar nuestros cambios y nos encontramos con un mensaje de error del servidor, indicándonos la imposibilidad de llevar a cabo la modificación, debido a que el registro ha sufrido una alteración desde el inicio de nuestra transacción. A esta conclusión se llega comparando nuestro identificador de transacción (*t100*), que es menor que el indetificador de la transacción de la última versión del registro (*t101*). Sin haber utilizado un bloqueo explícito del registro, la transacción *t101* ha conseguido proteger sus cambios, incluso ante una transacción iniciada con anterioridad a ella misma. Solución: nuestra transacción (*t100*) debe cerrarse e iniciar una nueva (*t102*) para poder aplicar sus cambios.

Las reglas[10] que siguen el comportamiento de la concurrencia visto en el ejemplo anterior son:

- Si el identificador de la transacción es menor que el indetificador de la transacción de la última versión del registro, entonces la transacción **no** puede ver o modificar el registro.
- Si el identificador de la transacción es igual que el indetificador de la transacción de la última versión del registro, entonces la transacción **sí** puede ver y/o modificar el registro.

[24] Puntero.
[25] Back Difference Record.

- Si el identificador de la transacción es mayor que el identificador de la transacción de la última versión del registro, y además esta última fue confirmada (*commit*) antes de que la transacción comenzará, entonces ésta **sí** puede ver y/o modificar el registro.

Siguiendo el mismo razonamiento deducimos el mecanismo empleado para dar soporte a los diferentes niveles de aislamiento de las transacciones. Este se basa en comparar el identificador de nuestra transacción con el de las transacciones de cada una de las versiones de los registros. En función de que sean menores, iguales, o menores, del estado en el que se encuentren (obtenido de las páginas TIP), y el nivel de aislamiento de nuestra transacción, se podrá acceder a unas, u otras, versiones de cada uno de los registros.

Las características vistas hasta ahora en el mecanismo de *multiversión* son realmente interesantes, y constituyen un método elegante y eficaz para resolver el problema de la concurrencia de múltiples usuarios y los diferentes niveles de aislamiento de las transacciones en bases de datos InterBase. ¿ Demasiado bonito para ser verdad ? ... cierto, la *multiversión* también tiene su lado oscuro, y está precisamente ahí, en la generación de múltiples versiones de los registros. Aunque hemos visto que en cada nueva versión de un registro sólo se registran los cambios reales aplicados, estas nuevas versiones pueden llegar a ocupar mucho espacio innecesariamente, además de someter a los procesos que acceden a la base de datos a una sobrecarga al tener que recorrer las cadenas de las versiones de cada registro, seguramente muchas de estas versiones ya obsoletas. Esta *basura* que genera el mecanismo de *multiversión* es responsable de una pérdida gradual del rendimiento en el acceso a la base de datos, que debe ser recuperada a través de mecanismos de *limpieza*, como veremos en la siguiente sección.

2.2. Limpiando la base de datos: sweeping, gbak y gfix

Como consecuencia del mecanismo de *multiversión*, se generan en la base de datos versiones de los registros que acaban siendo obsoletas. Estas versiones obsoletas deben ser retiradas de la base de datos para que no afecten al rendimiento del acceso a los datos. Existen tres vías, no excluyentes, para conseguir esta *recolección de basura*:

- En primer lugar está la ***recolección cooperativa de basura***[26]. Este tipo de limpieza se realiza cada vez que un registro es tocado por una lectura (*select*), una modificación (*update*), o una eliminación (*delete*). Cada vez que ésto sucede, el servidor InterBase lanza el proceso[27] de recolección, comparando el identificador de la transacción de cada uno de los BDR[28] de las versiones del registro con el identificador de la OIT[29] que obtiene de la página de cabecera de la base de datos. Si el identificador de la transacción que generó el BDR es menor que el identificador de la OIT, esa versión del registro puede ser eliminada de la base de datos y su espacio puede ser utilizado por nuevos datos. Se debe tener en cuenta que este tipo de recolección de basura sólo actúa contra transacciones ya confirmadas (*commited*), pero no contra transacciones en estado cancelado (*rolled_back*), o *limbo*. Tampoco recupera el espacio ocupado por los BDR que correspondan a registros eliminados.

[26] Cooperative garbage collection.
[27] En las versiones InterBase 6.x Super Server se lanza en un hilo independiente.
[28] Ver la sección 2.1 en la página 10.
[29] Ver la sección 1.4 en la página 9.

- En segundo lugar tenemos el ***sweeping***[30]. La recolección de basura que realiza el *sweeping* es idéntica al caso anterior, pero además se añaden las transacciones en estado cancelado (*rolled_back*) y los BDR de los registros eliminados. El mecanismo de *sweeping* es lanzado por el servidor de manera automática tras alcanzar una diferencia de 20,001[31] unidades entre el identificador de la OIT y el identificador que se asignará a la próxima transacción[32]. Aunque el *sweeping* puede llevarse a cabo en línea con las operaciones normales contra la base de datos, puede repercutir negativamente en el rendimiento de los accesos a la misma. En determinados casos puede ser conveniente desactivar el control automático que lanza el proceso de *sweeping* y hacer que el administrador del servidor lo realice manualmente, o programarlo para ser realizado en momentos de baja actividad del sistema, por ejemplo por las noches. El *sweeping* avanza el identificador de la OIT tanto como sea posible; en el caso de un acceso exclusivo a la base de datos (a través de un *shutdown* de la base de datos), fijará el identificador de la OIT en una unidad menos que el identificador que se asignará a la próxima transacción.

- La tercera posibilidad es el uso de la utilidad en línea de comandos ***gbak***, para llevar a cabo un ***backup*** y un posterior ***restore*** de la base de datos. Al hacer un *backup* se fijan las versiones actuales de los registros a las últimas confirmadas (*commited*), eliminando los BDR. Además, se eliminan de las páginas TIP aquellas transacciones que han quedado obsoletas, al igual que las transacciones que no han sido confirmadas (*commited*), y por último se inician los identificadores de las transacciones a los valores iniciales. Como podemos deducir, la operación *backup/restore* lleva a cabo la limpieza más profunda de las tres vistas, y es especialemente adecuada en casos en que una elevada cantidad de versiones de registros dentro de la base de datos estén provocando una pérdida de rendimiento en los accesos.

Como comentamos en la sección 1.3, el uso de *isc_commit_retaining*[33] tiene ciertas ventajas a la hora de ganar tiempo en el cierre de la transacción y en la apertura de una nueva, pero también sus inconvenientes. Uno de estos inconvenientes es que *isc_commit_retaining* no lanza el mecanismo de *recolección cooperativa de basura*, ya que el identificador de la OIT no avanza hasta que se produzca un *hard-commit* por medio de una llamada a *isc_commit_transaction*. El uso de *isc_commit_retaining* debe controlarse con sumo cuidado, ya que se puede caer en una pérdida de rendimiento en bases de datos con una elevada actividad.

Anteriormente apuntamos la posibilidad de desactivar el control que lanza automáticamente el *sweeping*. Para desactivarlo podemos utilizar la utilidad en línea de comandos ***gfix***[34]:

```
gfix -h[ousekeeping] n ruta_base_de_datos
```

Siendo ***n*** un valor entero que fija la diferencia que se debe alcanzar entre el identificador de la OIT y el identificador de la próxima transacción para que se lance automáticamente el *sweeping*; el valor predeterminado es 20,000. Si fijamos el valor a 0 el mecanismo autómatico de *sweeping* se desactiva y habrá que lanzarlo manualmente con:

[30] Sweep: barrer, borrar.

[31] Este valor es el predefinido para toda base de datos InterBase, aunque es configurable.

[32] Esto es lo que aparece en la documentación oficial[11]. Algunos autores[12] sostienen que la diferencia debe estar entre los identificadores de la OIT y la OAT.

[33] O los correspondientes métodos de los componentes que llamen a esta función.

[34] Con determinadas herramientas gráficas también es posible hacerlo, caso de **IBConsole** e **IBAccess**.

```
gfix -s[weep] ruta_base_de_datos
```

La utilidad *gfix* también permite recuperar las transacciones que estén en estado ***limbo***, aquellas que han superado la primera fase de un *commit en dos fases*, pero no han llegado a finalizar la segunda. Para ello disponemos de dos opciones:

```
gfix -c[ommit] [Id | all] ruta_base_de_datos
```

Confirmará (*commit*) aquella transacción en estado *limbo* especificada por su identificador (*Id*), o bien todas (*all*).

```
gfix -r[ollback] [Id | all] ruta_base_de_datos
```

Cancelará (*rollback*) aquella transacción en estado *limbo* especificada por su identificador (*Id*), o bien todas (*all*).

Por último, es interesante comentar que el mecanismo de *multiversión* no sólo afecta a los registros de datos, también se utiliza en los *metadatos* que describen la estructura de nuestra base de datos. Estos *metadatos* se almacenan en tablas[35] que pueden generar gran cantidad de versiones de registros si son expuestas a muchos cambios, por ejemplo en bases de datos que están en fase de diseño. Terminada esta fase de diseño sería conveniente someter a la base de datos a un proceso *backup/restore*, para limpiarla de versiones obsoletas de registros en los *metadatos* que podrían repercutir negativamente en el rendimiento.

2.3. Bloqueos muertos, dead-locks

Un ***bloqueo muerto*** es una curiosa situación que puede darse durante el acceso concurrente de varias transacciones a los mismos datos. Para describir[16] qué es un *bloqueo muerto*, imaginemos la siguiente situación:

Supongamos que un usuario[36] inicia una transacción (*t100*) contra una base de datos, leyendo y modificando un registro de una tabla. A continuación, un segundo usuario inicia otra transacción (*t101*), leyendo y modificando otro registro diferente. Supondremos que ambas transacciones utilizan el parámetro `WAIT`, lo que implicará que ambas transacciones esperarán el desbloqueo del registro ante un conflicto de actualización[37]. En este punto el primer usuario lee y modifica el registro que leyó y modificó el segundo usuario, todo ello dentro de la transacción *t100*, que sigue en estado *activo*, no confirmada (*commited*). El segundo usuario lee y modifica el primer registro que leyó y modificó el primer usuario, dentro de su transacción, la *t101*, que también sigue en estado *activo*.

Ante esta situación se produce el *bloqueo muerto*, ya que la transacción *t100* no puede confirmar sus cambios debido a los cambios que ha realizado el usuario en la transación *t101*. El segundo usuario, recíprocamente, tampoco podrá confirmar los suyos por el mismo motivo. Además, debido al parámetro `WAIT` de la transacción, se quedará esperando indefinidamente hasta la resolución del conflicto de actualización de los mismos datos. La solución para este ”abrazo mortal” sólo puede venir por la cancelación de los cambios de una de las transacciones, que a su vez habilitará el que se puedan confirmar los de la otra.

[35] Denominadas *tablas del sistema*.

[36] Como hemos comentado, el *bloqueo muerto* se produce durante el acceso concurrente de varias transacciones. Para que sea más clarificador supondremos que las transacciones pertenecen a usuarios diferentes, pero podrían corresponder a un sólo usuario, incluso a una única aplicación.

[37] Ver sección 1.2

2.4. Commit en dos fases

El mecanismo conocido como ***commit en dos fases***[38] permite que varias operaciones, que involucran a varias bases de datos, se ejecuten todas ellas dentro del control de una única transacción. Ésto significa que todas estas operaciones finalizarán con éxito (*commit*) o serán canceladas (rollback) en todas las bases de datos implicadas[14]. El mecanismo en sí es completamente transparente para el desarrollador, queriendo ésto decir que se aplicará como si en la transacción sólo estuviese implicada una base de datos. Determinados métodos de acceso, caso del BDE, no soportan el commit en dos fases, ya que su diseño no permite que una transacción pueda asociarse a varias bases de datos; otros métodos, por ejemplo IBX, IBO, FIBPlus, ..., etc., sí lo soportan. Es importante comentar, como hemos visto en la sección dedicada al API InterBase, que también existen funciones para controlar paso a paso el mecanismo[39].

¿Cómo funciona?[15] En la primera fase se marca dentro de la tabla del sistema **RDB$TRANSACTIONS**, de cada una de las bases bases de datos implicadas, que la transacción va a ser finalizada. La tabla RDB$TRANSACTIONS sólo existe para ésto, para llevar un registro de las transacciones que implican a varias bases de datos. En la segunda fase puede ocurrir que todas las operaciones, en todas la bases de datos, se confirmen (*commit*) realmente o que alguna rechace los cambios. Este último caso implicaría que el resto de operaciones serían canceladas (*rollback*), siguiendo el registro de la tabla RDB$TRANSACTIONS de cada una de las bases de datos. En ambos casos, al finalizar el proceso no debe quedar ninguna referencia a la transacción en las tablas RDB$TRANSACTIONS de cada una de las bases de datos implicadas. Una tercera posibilidad es una interrupción del proceso antes de que se confirmen (o hipóteticamente se rechacen) todas las operaciones, por ejemplo ante un fallo general del sistema, un corte de energía, etc. En este caso, las transacciones permanecerán en estado ***limbo***, pudiendo ser recuperadas o desechadas posteriormente[40].

En todo caso, el *commit en dos fases* no implica que se puedan *mezclar objetos*[41] de las diferentes bases de datos en una misma operación, por ejemplo en una consulta (*select*), una iserción (*insert*), una modificación (*update*), o una eliminación (*delete*). El commit en dos fases involucra las operaciones, no los *objetos*, contra varias bases de datos en única transacción. ¿Qué utilidad tiene? Traspaso de datos de una base de datos a otra, y en ambas la información que almacenan está relacionada y además debe permanecer en un estado consistente. Como ejemplos podemos citar la transferencia bancaria del inicio del documento, suponiendo que nuestras cuentas de ahorros y corriente se almacenan en bases de datos independientes. Otro ejemplo lo tenemos en las herramientas de replicación de bases de datos.

2.5. Generadores

Comentábamos anteriormente que toda operación contra una base de datos InterBase está dentro del control de una transacción ... ¿Toda? ¡No! Existe un grupo de irreductibles elementos que se resisten a ser controlados[42]. Estos elementos son los ***generadores***.

[38] Two-phase commit, o *2PC*.
[39] Ver sección 1.3.
[40] Por ejemplo con ***gfix***. Ver sección 2.2.
[41] Tablas, procedimientos almacenados, generadores, etc.
[42] "Año 50, antes de Jesucristo. Toda la Galia está ocupada por los romanos... ¿Toda? ¡No! Un poblado habitado por irreducibles galos resiste, invicto siempre, al invasor". Incombustible Asterix.

La definición y uso de los generadores está perfectamente descrita en la documentación[13], por tanto aquí nos centraremos en algunos aspectos curiosos, e interesantes, de estos elementos.

¿Por qué motivo los generadores deben estar fuera del control transaccional? La respuesta deberían darla los desarrolladores InterBase, pero en mi opinión es debido a la propia razón de ser de los generadores. Estos fueron introducidos en InterBase como respuesta a la carencia de un mecanismo que facilitara la creación de valores únicos, y en secuencia, que pudieran ser utilizados como valores para las claves primarias de los registros. Si estuviesen dentro del control de la transacción desde la que se lee (y posiblemente se modifica) el generador, cabría la posibilidad, por ejemplo por una cancelación (*rollback*) de la transacción, que se intentase asignar un valor del generador ya asignado.

El que los generadores estén fuera del control transaccional plantea un problema: seguramente sabrá que los generadores de una base de datos se almacenan en una de las tablas del sistema, concretamente en **RDB$GENERATORS.** Por mucho que busque en esta tabla no encontrará el valor de los generadores que almacena, y ésto es debido a que las tablas del sistema, como cualquier otra tabla que cree en su base de datos, debe ser accedida dentro del control de una transacción. ¿Dónde se almacena entoces este valor? Existen unas páginas especiales en las bases de datos InterBase para esta función, las ***páginas de generadores***.

Otra cuestión interesante sobre los generadores es que una vez que se modifica uno de ellos, por medio de la función incorporada **GEN_ID**, no es posible deshacer el cambio por medio de un *rollback*. Esto parece evidente al estar fuera del control de la transacción desde la que se llama a la función, pero ... ¿cómo se puede establecer entonces un generador a un valor determinado desde un procedimiento almacenado o un disparador (*trigger*)[43]? La respuesta es sencilla, pero a la vez tiene trampa. Puede crear un procedimiento almacenado que aproveche la función incorporada GEN_ID. Normalmente utilizamos esta función para proporcionarnos, junto a un generador, valores enteros únicos en secuencia ascendente - GEN_ID(Generador, 1) -, pero nada impide a GEN_ID tomar saltos distintos a 1, e incluso negativos. Estas dos características son las que utilizaremos para fijar el nuevo valor del generador:

```
SET TERM !! ;

CREATE PROCEDURE Establecer_GENERADOR_X (
  ParNuevoValor INTEGER)
AS
DECLARE VARIABLE ValorActual INTEGER;
BEGIN
  /* 1 .. recuperamos el valor actual del generador */
  ValorActual = GEN_ID(GENERADOR_X, 0);

  /* 2 .. fijamos el nuevo valor del generador */
  ValorActual = GEN_ID(GENERADOR_X, (:ParNuevoValor) - :ValorActual);
END; !!

SET TERM ; !!
```

Lo que hace es aprovecharse de la posibilidad de leer (pero no modificar) el valor actual

[43] Donde no es posible utilizar la sentencia **SET GENERATOR ...**

de un generador (/* 1 .. */), y que los generadores pueden tomar saltos mayores que 1 y negativos si es necesario (/* 2 .. */).

¿ Dónde está la trampa? Si **después** de recuperar el valor actual del generador (/* 1 .. */), y **antes** de fijar el nuevo valor (/* 2 .. */), otro usuario[44] hiciese "saltar" el generador (hacia delante, o hacia atrás), ya no se podría garantizar que el generador quedará fijado al valor que pasamos como parámetro al procedimiento. Esto es debido a que los generadores (y por extensión la función GEN_ID) no están sujetos al control de la transacción que los modifica. En entornos de un sólo usuario (o que garanticemos que cuando se utilice el procedimiento sólo hay un usuario conectado a la base de datos), el procedimiento funcionará correctamente, en entornos multiusuario no es posible garantizarlo.

3. Utilizando las transacciones desde Delphi

Existen diversas formas de acceder desde Delphi a un servidor InterBase. Podemos utilizar llamadas directas a las funciones del **API** InterBase, podemos utilizar los mecanismos de acceso universal a bases de datos propuestos por Borland: el viejo **Borland Database Engine (BDE)**, o el nuevo **dbExpress**; también podemos utilizar controladores **ODBC**. En estas tres últimas alternativas introducimos un nivel adicional dentro de la cadena de llamadas que van desde nuestra aplicación hasta el API InterBase; en principio ésto no es bueno por cuestiones de rendimiento, pero también puede tener sus ventajas dentro del diseño de la arquitectura de cada uno de estos métodos de acceso, especialmente en el caso de dbExpress. Por último tenemos las llamadas bibliotecas de componentes de *acceso nativo* a InterBase. Entre las más destacadas podemos citar: **InterBase Objects[17]** (también conocido como **IBO**), **FIBPlus**[18], **Zeos Library[19],** y la que acompaña a Delphi de manera estándar, **InterBase eXpress**. Nos centraremos exclusivamente en esta última por estar al alcance de cualquiera que posea Delphi, aunque varias de las citadas tienen prestaciones similares e incluso, en algunos casos, superiores.

3.1. InterBase eXpress (IBX)

InterBase eXpress (también conocido como **IBX**) es el heredero directo de los FreeInterBase Components, de Gregory Deatz. Actualmente **Jeff Overcash**[20] (**TeamB**) es el responsable de desarrollo de IBX. Borland los distribuye de manera gratuita y con código fuente incluido, bajo licencia *Mozilla Public License*.

3.1.1. El componente TIBTransaction

TIBTransaction es el componente IBX destinado a la gestión de las transacciones desde sus aplicaciones Delphi, kylix o C++ Builder. Estas son las principales propiedades y métodos del componente:

Propiedades

Active: En lectura informa si la transacción está en estado *activo*. En escritura provoca una llamada al método *StartTransaction* (valor True), o al método *Rollback* (valor False).

[44] En realidad otra transacción.

AutoStopAction: Determina el tipo de acción a realizar cuando se cierra el último DataSet asociado al componente TIBTransaction. El tipo de acción puede ser una llamada a uno de los métodos siguientes: *Commit*, *CommitRetaining*, *Rollback* o *RollbackRetaining*. En caso de querer un control esplícito del cierre de la transacción, debe tener el valor *saNone*.

Databases: Es una lista que contiene los componentes TIBDatabase asociados a la transacción.

DefaultAction: Determina el tipo de acción a realizar cuando expira el temporizador *IdleTimer*. Será una llamada a uno de los métodos siguientes: *Commit*, *CommitRetaining*, *Rollback* o *RollbackRetaining*.

IdleTimer: Tiempo en milisegundos que deben transcurrir para que la transacción en estado *inactivo* (idle) haga saltar la llamada al método definido en la propiedad *DefaultAction*.

InTransction: Propiedad Booleana que determina si una transacción está en *progreso*.

Params: Es un TStrings que almacena los *parámetros*[45] de la transacción. El componente también tiene un editor de propiedades interno para especificar estos parámetros (*Transaction Editor...*), al que se puede acceder haciendo clic al botón derecho mientras tiene seleccionado el componente.

SQLObjects: Es una lista que contiene los DataSets y TIBSQL asociados al componente TIBTransaction.

TPB: Es un puntero al TPB[46] que define los parámetros de la transacción. Sólo lectura

Métodos

Commit: Encapsula la llamada a la función del API *isc_commit_transaction*.

CommitRetaining: Encapsula la llamada a la función del API *isc_commit_retaining*.

Rollback: Encapsula la llamada a la función del API *isc_rollback_transaction*.

RollbackRetaining: Encapsula la llamada a la función del API *isc_rollback_retaining*.

StartTransaction: Se encarga de generar un TPB para la transacción y encapsula la llamada a la función del API *isc_start_multiple*.

4. Licencia

[45] Puede encontrar una lista de los parámetros disponibles en la documentación[7].
[46] Ver sección 1.3

5. Historial

0.2.4 24.Junio.2002 Primera versión pública del documento.

0.2.5 25.Junio.2002 Corrección de errores y erratas.

0.2.6 04.Julio.2002 Corrección de errores y erratas.

Referencias

[1] **Henry F. Korth, Abraham Silberschatz**. *"Fundamentos de bases de datos"*, (1ª edición). McGraw-Hill.

[2] **George T. Heineman**. *"A transaction Manager Component for Cooperative Transaction Models"*. "1.1 Concurrency Control".

[3] **Inprise/Borland**. InterBase 6, *"Language Reference"*. Capítulo 2, "SQL Statement and Function Reference", SET TRANSACTION.

[4] **Borland International, Inc**. InterBase 4, *"Programmer's Guide"*. Capítulo 4, "Working with transactions".

[5] **Claudio Valderrama**. *"Transaction options explanation"*, http://www.cvalde.com/document/TransactionOptions.htm.

[6] **Toni Martir**. *IBAccess*. http://www.ibaccess.org/

[7] **Inprise/Borland**. InterBase 6, *"API Guide"*.

[8] **Ann W. Harrison**. *"The InterBase On-Disk Structure"*. http://www.ibphoenix.com/ibp_ods.html

[9] **Bill Todd**. *"Interbase: What Sets It Apart"*. "Locking schemes". http://community.borland.com/article/0,1410,27007,00.html

[10] **Paul McGee**. *"Managing Your InterBase Server"*. "Multi-generational Architecture".

[11] **Inprise/Borland**. InterBase 6, *"Operations Guide"*. Capítulo 6, "Database Configuration and Maintenance".

[12] **Paul Beach**. *"InterBase and the Oldest Interesting Transaction"*. http://www.ibphoenix.com/ibp_oit.html

[13] **Inprise/Borland**. InterBase 6, *"Data Definition Guide"*. Capítulo 11, "Working with Generators".

[14] **Yuri Breitbart, Héctor García-Molina, Avi Sliberschatz**. *"Overview of Multidatabase Transaction Management"*. "6.1 Two Phase Commit" (p. 25).

[15] **Claudio Valderrama**. *"Multiple database transactions"*. http://www.cvalde.com/features/f4/MultipleDatabaseTran.htm

[16] **IBPhoenix**. *"Explanation of Deadlocks"*. http://www.ibphoenix.com/ibp_exp_deadlocks.html

[17] **Jason Wharton**. *InterBase Objects*. http://www.ibobjects.com/

[18] **Sergey Buzadzhy**. *FIBPlus*. http://www.fibplus.net/

[19] **Zeos Developmen Group**. *Zeos Library*. http://www.zeoslib.org/

[20] **Jeff Overcash**. *InterBase eXpress*. http://codecentral.borland.com/codecentral/ccweb.exe/author?authorid=102

Nota: Para la redacción de este documento he utilizado más referencias, alrededor de unas cincuenta; la citadas anteriormente son las más significativas. Algunas de ellas pertenecen a bibliografía de mis tiempos de estudiante, y que probablemente ya estén descatalogadas en bibliotecas y librerías; otras, la mayor parte, son documentos obtenidos en Internet; lamentablemente, de algunas he perdido la referencia, pero supongo que el lector interesado no debería tener grandes dificultades para encontrarlas a través de algún buscador.