Un razonamiento después de cenar acerca de bloqueos optimistas y pesimistas
Por Alexander V. Nevsky, Alex Cherednichenko, traducido al inglés por Marina Novikova.
Este artículo apareció inicialmente en InterBase World.
Traducido al español por Edgar Fernando Rodriguez P, apexcol@gmail.com

Pregunta: Puede uno tragarse una bola de billar?
Respuesta: Sí, pero para qué?

Cuando los novatos comienzan a desarrollar aplicaciones tanto con IB/FB, la pregunta que a menudo se hacen, especialmente si tratan con servidores SQL con políticas de bloqueo, es cómo prevenir a los usuarios de editar datos y sobrescribir los cambios de otro en un ambiente multi-usuario. Buscan formas de bloquear registros, pero IB y FB no soportan bloqueos explícitos de registro. Precisamente, Firebird no disponía de esto antes de la versión 1.5, y hablaremos de esto al final del artículo.

No es normal bloquear explícitamente los registros en un servidor SQL de arquitectura multi-generacional (MGA). Es posible y, algunas veces, puede ser incluso útil. En muchos casos, sin embargo, la opción correcta de nivel de aislamiento transaccional y el flujo de trabajo de la aplicación evitará tanto sobrecargar el servidor como bloquear el acceso a otros usuarios.

Primero, es una mala práctica diseñar aplicaciones donde todos los cambios en los datos ocurran en el contexto de una sola transacción de larga duración sobre grandes volúmenes de registros. Desafortunadamente, el suit de componentes Interbase Express, que viene con Delphi y BCB, nos lleva a tales prácticas, especialmente en el ejemplo Employee, donde un solo componente TIBTransaction opera sin descanso en modo CommitRetaining. En nuestra opinión, es más lógico usar los componentes FIBPlus, desarrollados bajo la misma idea original. (FreeIBComponents, por Gregory Deatz).

El objetivo de FIBPlus es permitir la ejecución de un UpdateSQL de un dataset en una transacción aparte de la que tienen las operaciones SelectSQL y RefreshSQL. Un grupo de registros se selecciona en un control de estilo DBGrid mediante una transacción de sólo-lectura prolongada (la cual no retiene la progresión del logging de transacciones ni la recolección de basura – Garbage Collection), mientras los datos se modifican por otra transacción corta, posiblemente con un nivel de aislamiento diferente. Para IBX hay un parche hecho por Fanis Ghalimarzanov que activa la misma funcionalidad.

Creemos que este es el alcance correcto, y comenzamos discutiendo su implementación. Las razones de usar bloqueos en aplicaciones multiusuario incluyen:

1. Multiples usuarios haciendo operaciones DML en el mismo atributo, como “una cantidad de algo” (bienes, dinero, etc.). Por ejemplo, el personal de una bodega guardando entradas. Usando transacciones Read Commited “con la versión del registro” es suficiente para esto, proveyendo a la aplicación cliente evita este algoritmo de actualización:
    - Lee el valor actual
    - Lo cambia
    - Escribe un nuevo valor

El correcto alcance es incrementar registro, como en la sentencia

```
update Bienes_en_Stock set Cantidad = Cantidad + :Incremento
where Bienes = :Bienes
```

Así sería el algoritmo para el modo de transacción NoWait (usando pseudocódigo de Pascal):

```
Repeat
  StartTransaction
    Try
      Actualice;
      Respuesta := idOK;
    Except Respuesta := Application.MessageBox('Repite?' ….);
    End;
  Commit;
Until (Respuesta = idOK) Or (Respuesta = idNo);
/*Ok o el usuario lo hará más tarde */
```

o simplemente usar el modo de transacción Wait. En ese caso, perdemos la opción de terminar la actualización y hacer algo más mientras reintentamos.

Los conflictos raramente ocurren. Incluso si múltiples usuarios trabajan con los mismos ítemes, puede ocurrir conflictos sólo en el caso que otro usuario guardó los cambios durante el intervalo entre el inicio de la transacción (StartTransaction) y la actualización (Actualice), sin consignar datos (Commit). Incluso si esto sucedió, sólo volvemos a intentarlo (modo de transacción NoWait) o esperamos (Wait).

Los autores tratan de evitar esto usando el modo Wait, pero requiere mucha atención al diseño de acceso a los datosta donde múltiples instrucciones de actualizacion se ejecuten en la transacción. En sistemas complejos la posibilidad de un "deadlock" es más alta (los mensajes de error de IB/FB se refieren a cualquier conflicto de bloqueo como un "deadlock", incluso cuando un conflicto no es un bloqueo deadlock). Hablaremos de esto luego, discutiendo las nuevas características de FB 1.5.

2. Los usuarios compiten por algún recurso. Por ejemplo, un vendedor puede reservar algunos bienes para algunos clientes en caso de que se acaben. Tal opción se guarda antes de que los bienes físicamente se muevan fuera del inventario. La solución es similar a la anterior, excepto que un trigger tipo Before Update se añade. Revisa que el inventario disponible no sea negativo y ajusta el lado del cliente apropiadamente. Si se acaban los bienes, no hay razón para continuar la operación.

3. Los cambios requeridos no se pueden hacer incrementando o decrementando una cantidad, por ejemplo cuando el nombre de un ítem se cambia, pero probabilidad de conflicto es baja. La lógica de Aplicación probablemente permitirá al usuario re-escribir nuevos nombres sin mucha dificultad. Para esta escritura, el nivel de aislamiento de transacción "concurrency" (snapshot) es el que ajusta mejor de todos.

El usuario navega en datasets, escoge un registro para editar y luego

- el snapshot inicia;
- el registro se relee (los datos en la base pueden estar desactualizados);
- es editado en el control visual y se intenta un Update (Actualización);

En este caso el conflicto siempre ocurrirá en el modo nowait hasta que dos intentos simultáneos guarden cambios en el mismo registro, sin importar qué usuario consignó los datos (commit). Si el conflicto ocurre, una transacción tiene que revertirse. Si la escritura envuelve varias sentencias SQL, necesita revertirla, si una sentencia, puede solo consignarla mientras que actualmente no hubo cambios dentro de la base de datos. Esto es el porqué de que en este caso recomendemos hacer consignaciones (commit) mientras las reversiones (rollback) incrementen la falla entre OIT (transacción interesante más antigua) y la OAT (transacción activa más angigua) y el ciclo se repita.

En tales casos, con IBX y FIBPlus, los contenidos de los controles de datos se pierden después de un commit/rollback, así que es importante implementar alguna clase de cacheo de entrada en la aplicación cliente, para preservar los cambios del usuario en el evento de un conflicto. Una forma fácil es usar controles no basados en datos para la entrada del usuario y luego releer los valores en un componente basado en datos. En el modo Wait el servidor esperará a las acciones de la transacción en conflicto. Si hace un rollback, nuestros cambios se aplicarán, si consigna, tendremos una excepción de conflicto de bloqueo y manejarlo en la misma forma como en el modo NoWait.

Los siguientes casos son ejemplos de cuán necesario es tragar la bola de billar (como se mencionó antes en el epigrama). Todos los alcances descritos antes tienen la única desventaja que algún usuario irresponsable puede detener el trabajo de sus colegas por mucho tiempo. Esto es el porqué de formular como primera medida las tareas a funcionar con las técnicas descritas antes, con algunas mediciones adicionales para controlar el comportamiento del usuario, si se requiere.

4. Se ha editado algún otro documento y no queremos volver a la posición de revertir el trabajo y empezar de nuevo.

   La transacción en este caso inicia y la primera sentencia intenta hacer esto

   ```
   update MiTabla
   set = <atrib_escalar_no-indexado> = <atrib_escalar_no-indexado>
   where Columnas_de_Llave_Primaria = <definición_de_registro>
   ```

   Esta técnica es llamada por los programadores como "dummy update". Si esto causa un conflicto, significa que el registro ya tiene una actualización pendiente en alguna parte. La aplicación obtiene una excepción desde la API de FB/IB y debería manejarse revirtiendo y no permitiendo al usuario editar el registro. Dejándolo sin resolver hará el registro inaccesible para otros cambios de usuario (transacciones) hasta el final de nuestra transacción. Si es exitoso el dummy update, el registro puede ser releído y editado con certeza de que los cambios serán guardados.

Si alguna aplicación no está con esta disciplina de bloqueo pesimista, los efectos serán:

- Si el nivel de aislamiento de su transacción es Concurrency, el conflicto será inevitable cuando intente guardar los cambios actuales.
- Si el nivel de aislamiento es read_commited rec_version, el conflicto ocurrirá en el evento que intente guardar los cambios antes de nuestra consignación (commit). Si guarda los cambios después de nuestra consignación, nuestros cambios serán re-escritos.

Cuando se trabaja con sentencias automáticamente construidas, por ejemplo los componentes del BDE Ttable y Tquery, debería recordar eso, si no hay cambios durante una sesión en modo de Edición, la llamada a Post abortará y el BDE no enviará nada al servidor. Como resultado no habrá actualizaciones tipo dummy update.

5. Hay algunos “objetos” con datos en varias tablas. Las operaciones en atributos de esas tablas pueden crearse por usuarios de distintos departamentos y sus resultados de trabajo pueden influenciar las acciones de otros. Por ejemplo hablando acerca del transporte de bienes entre diferentes estaciones, las siguientes tablas se conectan con la tabla encabezado (maestra en maestra-detalle).

   - Ruta
   - Ordenes de compra y facturas de proveedores
   - Directivas para cargar/descargar en cada punto de la ruta
   - Inventario actual y anual proyectado de unidades de transporte para cada segmento de la ruta
   - Documentos de venta.

   Cualquier tabla del "objeto" puede editarse durante su ciclo de vida, usualmente esto se hace paso a paso. Por ejemplo, el superintendente de tráfico cambia la ruta y trabaja con el plan para la carga y descarga del inventario. Si al mismo tiempo se le da a la bodega instrucciones de carga desactualizadas y comienza a cargar, hay un problema. Y viceversa, cuando los itemes ya han sido cargados y toda la documentación necesaria se genera, el superintendente de tráfico puede repentinamente decidir enviar algunos itemes más o cambiar el destino.

   En este caso es fácil aplicar el método 4, donde la lógica de la aplicación cliente, probablemente junto con su conjunto de procedimientos almacenados y triggers (si los hay) se adaptan para permitir un intento de bloqueo al encabezado antes de intentar trabajar con cualquier tabla. Note que esta clase de situación compleja no protege los datos de los errores de diseño del desarrollador o de los efectos de edición que pueden realizarse en un registro en una tabla dependiente por alguna otra herramienta de administración de base de datos. Actualmente sólo el registro en la tabla encabezado se bloquea de tal forma que si alguien cambia el inventario dejando la tabla encabezado desbloqueada, la estructura lógica de los datos (con respecto a la sincronía de los valores de los atributos relacionados) puede romperse o pueden surgir conflictos de integridad

referencial de datos en tablas dependientes que no se tocaron directamente por la aplicación.

Parcialmente este problema puede resolverse con la integridad referencial IB/FB usando constraints de llaves foráneas (foreign key) (ver punto 6).

6. Esta es una variación complicada de 5. Se necesita ejecutar y consignar transacciones mientras se trabaja en el "objeto" sin liberar el registro maestro (encabezado). La idea es usar un bloqueo de transacción separado en la tabla maestra, la cual bloquee el registro, se sugiere en sí mismo. Pero hay un solo inconveniente, como es usual. Cuando se intenta cambiar registros en las tablas referentes al registro bloqueado en la tabla maestra por una Llave Foránea en las transacciones activas, se generará un conflicto. La solución es crear una tabla adicional, que tenga una relación 1:1 con el encabezado y luego bloquear registros de esta tabla que correspondan al encabezado.

   Ahora es tiempo de hablar acerca de las nuevas habilidades de bloqueo introducidas en Firebird 1.5. Una actualización tipo dummy update tiene una desventaja: hace que los triggers de la tabla modificada se disparen y es necesario implementar lógica condicional allí, tal como escribir tablas log, etc.

   Firebird 1.5 incluye una nueva funcionalidad – opción Lock para las sentencias de Selección (Select). Al principio (en la Candidata 3) esta opción se permitía para las sentencias Select For Update y cualquiera posterior en cualquier Select. Consideraremos su uso en esta secuencia histórica, así:

   ```
   Select ... From Tabla[Where ...] [Order By ...] For Update With Lock.
   ```

   La sintaxis de Select For Update ha sido presentada en IB/FB por mucho tiempo pero no tenía nada que ver con bloqueos. Durante la ejecución de un Select normal, los registros se envían al cliente como paquetes. Mientras los registros se pasan a la aplicación uno tras otro, el programa cliente (gds32, fbclient, libgds, etc.) obtiene del servidor un paquete de registros de un tamaño a petición y lo reserva. Durante la ejecución de Select For Update los paquetes se forman de exactamente un registro. El siguiente paquete será formado y obtenido por el cliente sólo después de que la aplicación lo solicite. Select For Update junto a With Lock combina la funcionalidad de Select For Update con una actualización tipo dummy update. En otras palabras una nueva versión del registro se crea en el momento de la petición. De la misma forma ocurre cuando una actualización se envía, excepto que los triggers no disparan. De tal forma que esta sentencia puede usarse en todos los casos antes mencionados en vez de la actualización tipo dummy update y puede olvidarse acerca de la funcionalidad de los triggers. Cuando se usa la opción With Lock recuerde que los bloqueos se liberan cuando la transacción termina, no después de que la consulta (query) se cierra.

   Tomando juicio de la experiencia obtenida en los grupos de noticias, podemos decir que muchos novatos aplican muchas funciones para diferentes propósitos de los que pueden tener los desarrolladores de servidores. Volviendo al tema de este artículo, para evitar el esfuerzo de entender cómo funcionan las

transacciones en distintos niveles de aislamiento, ellos bloquean todo y pierden el beneficio de la velocidad de transferencia que el bloqueo optimista a nivel de registro brindan los sistemas multi-usuario. La sintaxis de Select For Update With Lock impone bloqueos en su cursor, un registro a la vez, hasta que el cursor entero se retorna al cliente y se bloquea. No es bloqueo a nivel de fila. Esto es el porqué de advertirles acerca de usar esta opción como una solución para evitar aprender cómo manejar conflictos de bloqueo de concurrencia (concurrency).

Decir, por ejemplo, que necesitamos bloquear 100 registros. Con Oracle, la misma sintaxis de bloqueo creará un cursor en el servidor. Dependiendo del nivel de aislamiento y el modo transaccional, resolverá de esta forma los conflictos para el cursor como un todo. Así en el cliente tendremos tanto un cursor bloqueado y vacío previamente o una excepción. La arquitectura de Firebird no implica tal opción de tal forma que obtendremos registros uno tras otro, y se bloquearán en el momento de enviar datos al cliente. Así que si necesitamos hacer algo con los 100 registros y no todos ellos pueden manejarse, es altamente posible operar en 99 de ellos y luego obtener un conflicto en el último.

Es posible obtener todos los registros de una, sobrecargando la red más que después de un simple Select (recuerde que cada registro se forma en un paquete separado con un encabezado y un contenido). Debería usar el nivel de aislamiento read_commited wait, pero no es bueno usar esta opción en tal forma. En teoría el modo de espera read_commited wait garantiza 100% que se obtendrán los resultados necesarios. Pero este modo requiere más atención durante el diseño de acceso. Supongamos que se bloquee 100 registros con un Pedido con Order by ID, y su colega haga esto con Order by ID DESC. Los bloqueos muertos Deadlock ocurren porque un proceso ha asegurado un recurso A y está esperando la liberación del recurso B. Otro ha asegurado el recurso B y está esperando para la liberación del recurso A. En la realidad el deadlock puede no ser así de obvio, tanto que los registros a bloquear son enviados por distintas personas (tal vez incluso por distintos departamentos) y basados en datos de distintas tablas. Hay mucho más que dos consultas que tienen datos de distintas tablas... Actualmente *esto es el camino al infierno*. Es interesante que si nos apartamos del bloqueo, la ejecución de Select For Update With Lock en una transacción read_commited rec_version parecerá como un cambio de nivel de aislamiento de la sentencia Select. Es como si se hubiera ejecutado en una transacción con aislamiento read_commited no_rec_version.

Cuando estábamos escribiendo la primera edición de este artículo, la versión actual de FB 1.5 era liberada como Candidata 3, sin restricciones en la estructura Select en caso de usar la opción de bloqueo, y el analizador de sintaxis omitió la construcción

```
For Update [Of] With Lock
```

Después del análisis sintáctico de la parte [Of] era simplemente ignorado. Para aquellos que disfrutan de experimentar con los Candidatos mencionaremos

peculiaridades de la ejecución Select For Update With Lock con estructuras más complejas:

- una consulta con agregados y Group By se ejecuta pero no bloquea nada;
- una consulta con Union se ejecuta pero no bloquea nada;
- una consulta con Join siempre bloquea sólo la tabla principal. Como el optimizador puede cambiar el plan de ejecución de la consulta en cualquier momento, esta opción hace que el bloqueo sea impredecible;
- una consulta con Distinct bloquea sólo los últimos registros filtrados mediante Distinct durante la ejecución del ordenamiento externo, esto es en la tabla

```
TESTDIST
  ID          ATTR
  =========== ===========
  1           3
  2           2
  3           2
  4           3
```

dos registros serán bloqueados después de ejecutar Select Distinct Attr From TestDist For Update With Lock. En un caso general serán arbitrarios porque el ordenamiento será hecho de forma Natural;
- una consulta en una Vista de sólo lectura se ejecuta pero no bloquea nada;
- probablemente no amerita mencionar una consulta en un procedimiento almacenado pero lo haremos: la consulta se ejecuta pero tampoco bloquea nada.

Después de una cantidad de experimentos y discusiones los desarrolladores de FB decidieron activar sólo un simple Select de una tabla usando la opción With Lock en la siguiente versión para evitar bloqueos impredecibles y confusión alrededor de las sentencias que se ejecutan pero no bloquean nada. Ahora los intentos de ejecutar tales sentencias elevan una excepción.

El uso de la opción With Lock en la sentencia ordinaria Select requiere las mismas precauciones que para Select For Update pero se complica con la siguiente consideración: no se puede predecir fácilmente cuantas filas serán bloqueadas a la vez cuando se abra una consulta si no se seleccionó una sola fila. Incluso si la consulta no está conectada a un control tipo DBGrid que fuerza una petición de datos de la cantidad de registros a ubicarse en pantalla y se tiene la intención de procesar las filas una a una, el servidor pedirá la cantidad de filas suficientes para llenar un paquete de red o incluso dos – uno a enviarse al cliente inmediatamente, otro para estar listo para la siguiente solicitud de cliente. Esta cantidad depende de el tamaño del paquete usado en su red, el tamaño de una fila en la tabla y el algoritmo de compresión del protocolo de red. Así cuando se use un simple Select With Lock en los cursores no sólo bloqueará una fila ni el cursor completo sino una cantidad desconocida de filas en la consulta abierta. Intencionalmente recomendamos usar bloqueos sólo con selecciones simples para evitar comportamientos impredecibles de la aplicación. Si se necesita aún bloqueos de cursor haga una petición total de registros inmediatamente después de abierta la consulta si quiere bloquear todas las filas o use Select For Update si quiere hacer proceso paso a paso. No use esta opción With Lock para bloquear

tablas enteras. Para hacer esto pegue una mirada a el nivel de aislamiento de consistencia en la documentación.

Mientras desarrollan aplicaciones cliente, los autores usan una versión IBX derivada de IBX4.42 y sportada por ellos mismos. Después de los primeros intentos de aplicar Select For Update With Lock vieron que IBX no soportaba Select For Update incluso sin la opción With Lock. A continuación se puede ver un parche para IBX, soportando esta funcionalidad de FB 1.5. Este problema se origina del conjunto inicial de componentes Free IB Components por Gregory H. Deatz, que sirve como base para los IBX y los FIBPlus.

La idea principal de este parche es manejar SQLType SelectForUpdate como una variante de de SQLType Select. Los propietarios de las últimas versiones de IBX pueden intentar aplicar este parche para concertar el suyo propio basado en la idea principal.

Es necesario modificar dos módulos: IBSQL.pas y IBCustomDataSet.pas Tres métodos de la clase TIBSQL deberían cambiarse en el módulo IBSQL.pas: TIBSQL.Close, TIBSQL.ExecQuery, TIBSQL.GetUniqueRelationName:

```
//*********************************************
procedure TIBSQL.Close;
var
  isc_res: ISC_STATUS;
begin
  try
    if (FHandle <> nil) and (SQLType = SQLSelect) and FOpen then
begin
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   Reemplazar por:
       if  (FHandle  <>  nil)  and  (SQLType  in  [SQLSelect,
SQLSelectForUpdate])
      and FOpen then begin
...

//*********************************************
procedure TIBSQL.ExecQuery;
var
  fetch_res: ISC_STATUS;
begin
  CheckClosed;
  if not Prepared then Prepare;
  CheckValidStatement;
  case FSQLType of
    SQLSelect: begin
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  Reemplazar por:
    SQLSelect, SQLSelectForUpdate: begin
...

//*********************************************
function TIBSQL.GetUniqueRelationName: String;
begin
  if FPrepared and (FSQLType = SQLSelect) then
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Reemplazar por:
  if FPrepared and (FSQLType in [SQLSelect, SQLSelectForUpdate])
```

En el módulo IBCustomDataset.pas se debería cambiar los métodos TIBCustomDataSet.InternalExecQuery y TIBCustomDataSet.InternalOpen

```
//*********************************************
procedure TIBCustomDataSet.InternalExecQuery;
var
  DidActivate: Boolean;
begin
  DidActivate := False;
  try
    ActivateConnection;
    DidActivate := ActivateTransaction;
    if FQSelect.SQL.Text = '' then
      IBError(ibxeEmptyQuery, [nil]);
    if not FInternalPrepared then
      InternalPrepare;
    if FQSelect.SQLType = SQLSelect then
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  Reemplazar por:
    if FQSelect.SQLType in [SQLSelect, SQLSelectForUpdate] then
...

//*********************************************
procedure TIBCustomDataSet.InternalOpen;
.......................
begin
  ActivateConnection;
  ActivateTransaction;
  if FQSelect.SQL.Text = '' then
    IBError(ibxeEmptyQuery, [nil]);
  if not FInternalPrepared then
    InternalPrepare;
  if FQSelect.SQLType = SQLSelect then begin
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  Reemplazar por:
   if FQSelect.SQLType in [SQLSelect, SQLSelectForUpdate] then
begin
...
```

FIBPlus introdujo cambios similares en el código para dar soporte a Select For Update en la versión 5.0. Las versiones anteriores no lo soportan**.** Los IBObjects no son muy populares en Rusia así que no sabemos acerca de su soporte para Select For Update. Estaremos muy agradecidos por cualquier comentario acerca de este tema.