Como travar um registro em Interbase / Firebird

Ou um bate-papo sobre travamento otimista e pessimista. Escrito por Alexander V. Nevsky e Alex Cherednichenko em russo, traduzido por Marina Novikova para o inglês e Eugênio Reis para o português.

Pergunta: É possível alguém engolir uma bola de sinuca?
Resposta: Sim, mas para quê?

Quando programadores novatos começam a desenvolver aplicações com IB/FB, a questão que eles freqüentemente fazem, especialmente se vieram de servidores SQL com filosofia de travamento, é como evitar que usuários alterem os mesmos dados e cubram as alterações uns dos outros em um ambiente multi-usuário. Eles buscam formas de travar registros, mas o IB e o FB não suportam travamento explícito de registros. Mais precisamente, o Firebird não dispunha desse recurso antes da versão 1.5, e falaremos a respeito disso no final do artigo.

Não é normal colocar travamentos explícitos em registros em um servidor SQL com arquitetura multi-gerações (MGA, em inglês). Isto é possível e, às vezes, pode até ser útil. Na maioria dos casos, entretanto, a escolha correta do nível de isolamento da transação e fluxo de trabalho da aplicação evitarão sobrecarga no servidor e travamento de recursos para outros usuários.

Primeiramente, é uma má prática projetar aplicações onde todas as alterações nos dados ocorrem no contexto de uma transação única e demorada sobre um grande volume de registros. Infelizmente, a suíte de componentes Interbase Express (IBX), embutida no Delphi e no C++ Builder, encoraja essa prática, especialmente no exemplo Employee (Empregado), onde um único componente TIBTransaction trabalha perpetuamente com o modo CommitRetaining. Na nossa opinião, faz mais sentido usar os componentes FIBPlus, desenvolvidos a partir do mesmo código básico original (FreeIBComponents, de Gregory Deatz).

A abordagem do FIBPlus é permitir executar o UpdateSQL de um dataset numa transação separada da que os comandos SelectSQL e RefreshSQL utilizam. Um conjunto de dados é selecionado dentro de um controle do tipo DBGrid através de uma transação prolongada apenas de leitura (a qual não retém o processamento do log de transações nem de coleta de lixo [garbage collection, em inglês]), enquanto os dados são modificados por uma outra transação curta, possivelmente com um nível diferente de isolamento. Para a suíte IBX, existe uma correção feita por Fanis Ghalimarzanov, permitindo essa mesma funcionalidade, disponível no seguinte endereço:

http://www.interbase-world.com/modules.php?name=News&file=article&sid=421

Nós acreditamos que esta é a abordagem correta e começaremos a discutir sua implementação. As razões básicas para usar travamento em aplicações de bancos de dados incluem:

1. Múltiplos usuários executando operações DML no mesmo atributo, tais como “uma quantidade de algo” (bens, dinheiro, etc.). Por exemplo, estoquistas registrando a entrada de produtos. Usando transações em modo Read Committed "com Versão de Registro (Record Version)" é suficiente, na medida em que a aplicação cliente evitar esse tipo de algoritmo de atualização:

   - Ler o valor atual

- Mudá-lo
- Gravar um novo valor

A abordagem correta seria gravar incrementos, como no comando:

```
UPDATE Itens_Em_Estoque
SET Quantidade = Quantidade + :Incremento
WHERE Cod_Item = :Cod_Item
```

O algoritmo para uma transação sem espera (NoWait) seria (usando um pseudo-código Pascal):

```
Repeat
  StartTransaction
    Try
      Update;
      Resposta := idOK;
    Except
      Resposta := MessageBox('Repete?' ….);
    End;
  Commit;
Until (Answer=idOK) Or (Answer=idNo);
/*Ok ou o usuário fará isso depois */
```

ou simplesmente use o modo de espera (wait). Neste caso, nós perderíamos a opção de terminar a atualização e fazer outra coisa enquanto tentamos de novo.
Raramente ocorrem conflitos. Mesmo quando múltiplos usuários estão trabalhando com os mesmos itens, conflitos poderiam ocorrer apenas no caso de um usuário ter feito mudanças durante o intervalo de tempo entre o StartTransaction-Update, sem ter dado Commit. Mesmo que isso tenha acontecido, nós apenas tentamos mais uma vez (em caso de NoWait) ou aguardamos (Wait).
Os autores deste artigo tentam evitar o modo de espera porque requer muita atenção no projeto de acesso aos dados onde múltiplos comandos de atualização vão ser executados dentro da transação. Em sistemas complexos, a possibilidade de "deadlocks" é maior (as mensagens de erro do IB/FB se referem a qualquer conflito de travamento como "deadlock", mesmo quando o conflito não é um deadlock propriamente dito). Falaremos mais sobre isso mais adiante, ao discutir novos recursos do FB 1.5.

2. Usuários estão competindo por algum recurso. Por exemplo, um vendedor pode reservar alguns produtos para algum cliente para garantir disponibilidade. O registro é feito na tabela de estoque antes dos produtos terem sido tirados do estoque físico. A solução é similar à anterior, exceto que neste caso adiciona-se um trigger do tipo Before Update. Ele verifica se o estoque disponível não está negativo e ajusta os dados no cliente. Se os produtos não estiverem mais disponíveis, não há razão para proceder com a reserva.

3. A mudança requerida não pode ser feita incrementando ou decrementando uma quantidade (por exemplo: quando os nomes de produtos são mudados), mas a probabilidade de conflito é baixa. A lógica da aplicação irá provavelmente permitir ao usuário redigitar novos nomes sem muita dificuldade. Para este caso, o nível de isolamento snapshot é o que atende melhor.

   O usuário navega nos datasets, escolhe um registro para alterar e então

   - o snapshot inicia;

- o registro é relido (os dados na tela podem estar desatualizados em relação ao banco);
- o registro é modificado com um controle visual e ocorre uma tentativa de gravação;

Neste caso, o conflito irá sempre ocorrer no modo nowait quando ocorrerem duas tentativas simultâneas de alterar o mesmo registro, independentemente do fato de alguém ter dado commit ou não. Se ocorrer conflito, uma das transações tem que ser desfeita (rollback). Se a gravação envolve vários comandos SQL, você precisa desfazê-la, se for apenas um comando, pode executar um commit, já que não houve nenhuma mudança dentro do banco de dados. É por isso que neste caso nós recomendados executar commit, porque o rollback aumenta o intervalo entre o OIT (oldest interesting transaction – transação significativa mais antiga) e o OAT (oldest active transaction – transação ativa mais antiga) e o loop se repete.

Nestes casos, com o IBX e o FIBPlus, o conteúdo dos controles db-aware é perdido depois de um commit/rollback, de forma que é desejável implementar algum tipo de cache de entrada na aplicação cliente para preservar as mudanças dos dados do usuário em caso de haver algum conflito. Uma solução simples é usar componentes não db-aware para a entrada de dados do usuário e então lançar os valores dentro de componentes db-aware. Em modo de espera (Wait), o servidor vai aguardar a resolução da transação conflitante. Se for um rollback, nossas alterações serão lançadas, se for um commit, iremos obter uma exceção de conflito de travamento e resolvê-la da mesma forma que se estivéssemos em NoWait.

Os casos a seguir são exemplos de quando é necessário engolir a bola de sinuca (veja o epigrama). Todas as abordagens descritas abaixo apresentam a desvantagem de permitir que algum usuário irresponsável impossibilite seus colegas de trabalhar por um longo período. Eis o motivo pelo qual nosso primeiro recurso deverá ser tentar formular a tarefa para funcionar com as técnicas descritas acima, com algumas medidas adicionais para controlar o comportamento do usuário, quando necessário.

4. Foram feitas alterações em documentos de texto e não queremos ficar em uma situação de ter que desfazer o trabalho e começar de novo..

   A transação de modificação inicia e o primeiro comando tenta executar:

```
UPDATE MinhaTabela
SET <campo escalar não indexado> = <campo escalar não indexado>
WHERE Chave_Primaria = <definição de registro>
```

   Esta é uma técnica que os programadores chamam de "dummy update" (tradução livre: “atualização preventiva”). Caso haja algum conflito, isso significa que o registro tem uma atualização pendente em algum lugar. A aplicação captura uma exceção de volta pela API e deve então resolvê-la desfazendo-a e não permitindo ao usuário que altere o registro. Deixar a situação sem resolução fará com que o registro fique inacessível para alterações por parte de outros usuários (transações) até o final da nossa transação. Se a atualização preventiva der certo, o registro pode ser relido e editado com a certeza de que suas alterações serão gravadas. Se alguma aplicação em paralelo não observar essas regras de procedimento de travamento pessimista, os efeitos serão:

   - se o nível de isolamento da transação for concorrente, serão inevitáveis os conflitos na hora da tentativa de gravar as modificações.

   - Se o nível de isolamento for read_commited rec_version, conflitos irão ocorrer no momento em que houver tentativa de gravar as modificações antes do nosso commit. Se as mudanças forem gravadas depois do nosso commit, nossas mudanças serão sobrescritas e perdidas.

Ao trabalhar com comandos de atualização escritos automaticamente, como por exemplo o objeto TTable do BDE e componentes TQuery, você deve lembrar que, se não houve mudanças durante uma sessão em modo Edit, a chamada ao Post será cancelada e o BDE não irá mandar nada para o servidor. Desta forma, não haverá nenhuma "atualização preventiva".

5. Existe algum "objeto" com dados de várias tabelas. Operações em campos dessas tabelas podem ser feitas por usuários em diferentes departamentos e os resultados de seu trabalho pode interferir nas ações uns dos outros. Por exemplo: ao falar das múltiplas etapas do transporte de bens consignados, as seguintes tabelas são conectadas com a tabela principal:

   - Rota
   - Pedido de compras e notas fiscais de fornecedores
   - Instruções para carga/descarga em cada ponto da rota
   - Estoque atual e projetado das unidades de transporte em cada segmento da rota
   - Documentos de controle de venda

   Cada tabela do "objeto" pode ser editada durante seu ciclo de vida, o que é normalmente feito passo a passo. Por exemplo, o supervisor de tráfego muda a rota e revisa o plano de carga/descarga. Se ao mesmo tempo o depósito obtiver instruções desatualizadas de carga, existirá um problema. E vice-versa, quando os itens já estiverem carregados e os documentos ainda estiverem sendo gerados, o supervisor de tráfego pode repentinamente decidir mandar outros itens ou mudar o destinatário.
   Neste caso, é fácil aplicar o método 4, onde a lógica da aplicação cliente, provavelmente em conjunto com stored procedures e triggers (quando aplicáveis), está adaptada para realizar uma tentativa de travamento dos dados principais antes de trabalhar com qualquer tabela. Observe que esse tipo de situação complexa não protege os dados de falhas na lógica do projeto de acesso aos dados nem de edição que pode ser realizada em um registro de uma tabela dependente por alguma ferramenta de administração de banco de dados. Na verdade, apenas o registro na tabela principal é travado de modo que, se alguém alterar o estoque deixando o registro destravado, a estrutura lógica de acesso aos dados dos dados (em relação à sincronização dos valores em campos relacionados) pode estar inconsistente e conflitos imprevisíveis de integridade de dados podem surgir em tabelas dependentes não manipuladas diretamente pela aplicação.
   Este problema pode ser parcialmente resolvido no IB/FB usando-se restrições de integridade referencial (chaves estrangeiras, veja item 6).

6. Uma variação mais complicada do item 5. Você precisa executar e gravar transações enquanto estiver trabalhando no "objeto" sem liberar o registro principal. A idéia de usar uma transação separada para travar o registro principal surge espontaneamente. Mas existe apenas um porém, como de costume. Ao tentar alterar dados em tabelas que referenciam o registro principal (ou registro pai) bloqueado como uma chave estrangeira, você irá esbarrar num conflito. A solução é criar mais uma tabela, com relacionamento 1 para 1 com a tabela principal e então travar os registros nessa outra tabela que correspondam aos da principal.

Agora é hora de falar sobre os novos recursos de travamento no Firebird 1.5. Uma atualização preventiva tem uma desvantagem: dispara os triggers da tabela modificada e é necessário implementar lógica condicional de forma que o trigger execute sua funcionalidade (como por exemplo gravação de log) no momento adequado.

O Firebird 1.5 inclui uma nova funcionalidade:

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

A sintaxe do Select For Update tem estado presente no Interbase por muito tempo, mas não tem nada a ver com travamento. Durante a execução de um Select normal, os registros são enviados para o clientes na forma de pacotes. Embora os registros sejam lidos pela aplicação um a um, o programa cliente (gds32, fbclient, libgds, etc.) recebe do servidor um pacote de registros e guarda-o. Durante a execução do Select For Update, os pacotes são formados de apenas um registro. O próximo pacote será formado e lido pelo cliente somente depois que a aplicação o requisitar. O Select For Update With Lock combina a functionalidade do Select For Update com uma atualização preventiva. Em outras palavras, uma nova versão do registro é criada no momento da leitura. É o mesmo que ocorre quando um update é executado, exceto que os triggers não são disparados. Portanto, este comando pode ser usado em todos os casos mencionados acima em vez de uma atualização preventiva e você pode se esquecer dos triggers. Ao usar a opção With Lock, lembre-se de que os travamentos são liberados apenas no momento do término da transação, e não após o fechamento da query.

A julgar pela experiência obtida em grupos de discussão, podemos dizer que muitos novatos aplicam as novas funcionalidades com propósitos diferentes daqueles pensados pelos desenvolvedores do servidor. Voltando ao ponto deste artigo, a fim de evitar o esforço de entender como as transações funcionam em diferentes níveis de isolamento, eles travam tudo e perdem o benefício da velocidade que o travamento otimista traz para sistemas multi-usuários. A sintaxe do Select For Update With Lock impõe travamentos em todo o escopo do seu cursor de dados, uma linha de cada vez até, que todo o cursor esteja lido e travado. Não se trata de travamento de linha. Esta é a razão pela qual advertimos você para não usar este recurso como forma de evitar ter que aprender a resolver conflitos de de travamento e concorrência.

Digamos, por exemplo, que precisamos travar 100 registros. Com o Oracle, a mesma sintaxe de lock irá criar um cursor no servidor. Dependendo do nível de isolamento e do modo da transação, ele irá resolver conflitos para o cursor como um todo. De forma que o cliente terá um cursor previamente travado ou uma condição de erro (exceção). A arquitetura do Firebird não pressupõe este comportamento, de forma que iremos ler os registros um a um e eles só eles serão travados no momento da leitura por parte do cliente. Assim sendo, se precisarmos de fazer algo com os 100 registros e nem todos eles possam ser manipulados, é perfeitamente possível operar sobre 99 e então encontrar um conflito com o último.

É possível ler todos os registros de uma só vez, sobrecarregando a rede muito mais do que depois de um simples Select (lembre-se de que cada registro é formado em um pacote separado com um cabeçalho e um rodapé). Você poderia usar o nível de isolamento read_commited rec_version junto com o modo de espera (wait), mas não é bom usar esse recurso dessa forma. Em tese, o read_commited / rec_version / wait garantem 100% que você vai obter os resultados necessários. Mas este modo de travamento requer mais atenção durante o projeto de acesso aos dados. Suponhamos que você trave 100 registros com Order By ID e seu colega faça o mesmo com Order By ID, Desc. Vai ocorrer um Deadlock porque um processo reteve o recurso A e está esperando pela liberação do recurso B. O outro reteve o recurso B e está esperando pela liberação do recurso A. Na prática, o deadlock pode não ser tão óbvio, na medida em que os registros a ser travados são alvo de diferentes pessoas (até mesmo de diferentes departamentos) e baseados em dados de diferentes tabelas. Pode haver muito mais do que duas queries envolvidas com dados de diferentes... De fato, esta é uma via direta para o inferno. É interessante notar que se deixarmos de lado o travamento, a execução do Select For Update With Lock no modo read_commited rec_version irá parecer uma mudança do nível de isolamento do comando Select. É como se estivesse sendo executado em uma transação com isolamento read_commited no_rec_version.

Quando escrevemos este artigo, a versão corrente do FB 1.5 era o Release Candidate 3, sem restrições na estrutura do Select quanto ao uso da opção Lock, e o analisador de sintaxe omitia a construção

```
For Update [Of <lista de campos>] With Lock
```

Depois da análise de sintaxe, o trecho [Of <lista de campos>] era simplesmente ignorado. Para aqueles que gostam de testar Release Candidates, nós mencionaremos peculiaridades da execução do Select For Update With Lock com estruturas mais complexas:

- uma query com agregações e um Group By é executada e não trava nada;
- uma query com Union é executada mas não trava nada;
- uma query com Join trava apenas a tabela principal. Como qualquer otimizador pode mudar o plano de execução a qualquer momento, este recurso torna o travamento imprevisível;
- uma query com Distinct trava apenas os últimos registros filtrados pelo Distinct durante a execução da ordenação externa of the external sorting. Exemplificando:

  ```
  TESTE_DIST
  ID          ATRIB
  =========== ===========
  1           3
  2           2
  3           2
  4           3
  ```

  dois registros serão bloqueados depois da execução de um Select Distinct Atrib From Teste_Dist For Update With Lock. Na maior parte dos casos eles serão arbitrários porque a ordenação será feita pelo método Natural;
- uma query em uma View apenas de leitura é executada, mas não trava nada;
- provavelmente não há razão para mencionar uma query sobre uma stored procedure, mas o faremos: a query executa sem travar nada também.

Depois de vários experimentos e discussões, os desenvolvedores do FB decidiram habilitar apenas um Select simples de apenas uma tabela, mas sem agregações, o Group By e o [Of <lista de campos>] ficam para a próxima versão.

A utilização da opção With Lock em Selects comuns requer as mesmas precauções de um Select For Update, com mais uma agravante: você não pode facilmente prever quantas linhas serão travadas de uma vez quando você abrir a query se não selecionar nenhuma linha. Mesmo que a query não esteja conectada a um dbGrid (o qual força a leitura da quanitdade de linhas a exibir na tela) e você tenha a intenção de processar as linhas uma por uma, o servidor irá ler a quantidade de linhas necessárias para preencher um pacote de rede ou até mesmo dois – um para ser enviado imediatamente ao cliente e outro pronto a próxima requisição. Esse número é dependente do tamanho do pacote usado na sua rede, do tamanho da linha na tabela e do algoritmo de compressão do protocolo de rede. Portanto, ao usar um mero Select With Lock em cursores, você não bloqueia apenas uma linha ou o cursor todo, mas sim uma quantidade desconhecida de linhas na abertura da query. Nós recomendamos usar travamento apenas com Selects em registros únicos para evitar o comportamento imprevisível da aplicação. Se você precisar de travamento em um cursor, faça um FecthAll imediatamente após abri-lo, caso queira travar todas as linhas ou use um Select For Update se quiser processar registro a

registro. Não use a opção With Lock para travar tabelas inteiras. Para conseguir isso, faça uma pesquisa na documentação, no tópico consistência do nível de isolamento, .

Ao desenvolver aplicações clientes, os autores usam uma versão do IBX derivada do IBX4.42 e desenvolvida por eles mesmos. Depois das pimeiras tentativas de aplicar o Select For Update With Lock, eles viram que o IBX não suporta o Select For Update mesmo sem a cláusula With Lock. Logo abaixo você irá encontrar uma correção para o IBX, adicionando essa funcionalidade do FB 1.5. Este problema se originou da suíte de componentes Free IB de Gregory H. Deatz, a qual serviu como base para o IBX e o FIBPlus.

A idéia principal desta correção é resolver o SQLType SelectForUpdate como uma variante do SQLType Select. Possuidores de versões posteriores do IBX podem tentar aplicar esta correção ou maquinar sua própria correção baseada na idéia principal.

É necessário modificar dois módulos: IBSQL.pas e IBCustomDataSet.pas. Três métodos da classe TIBSQL devem ser modificados no 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
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Substitua 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
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Substitua por:

```
    SQLSelect, SQLSelectForUpdate: begin
...
```

```
//********************************************
function TIBSQL.GetUniqueRelationName: String;
begin
```

```
  if FPrepared and (FSQLType = SQLSelect) then
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Substitua por:

```
  if FPrepared and (FSQLType in [SQLSelect, SQLSelectForUpdate])
```

No módulo IBCustomDataset.pas, você deverá modificar os métodos TIBCustomDataSet.InternalExecQuery e 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
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Substitua 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
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Substitua por:

```
  if FQSelect.SQLType in [SQLSelect, SQLSelectForUpdate] then begin
...
```

O FIBPlus introduziu mudanças similares em seu código para permitir suporte ao Select For Update na versão 5.0. Versões anteriores não implementam o recurso. O IBObjects não é muito popular na Rússia, de forma que não sabemos nada sobre seu suporte ao Select For Update. Seremos muito gratos a quem nos enviar quaisquer comentários sobre essa questão.

Os autores são muito gratos a Helen Borrie por sua excelente revisão e comentários, Dmitry Yemanov, Dmitry Kuzmenko, Nikolay Samofatov e Serg Buzadzhy por seus comentários e acréscimos e a todos os leitores da primeira versão por apontarem erros tipográficos e outras falhas.

O original em russo está aqui.

Endereço do artigo em inglês:

http://www.interbase-world.com/modules.php?name=News&file=article&sid=423

Artigo Original: **Alexander V. Nevsky e Alex Cherednichenko**	
Tradução e adaptação: **Eugénio Reis** eugenio.reis@comunidade-firebird.org	**Comunidade Firebird de Língua Portuguesa** **Visite a Comunidade em:** **http://www.comunidade-firebird.org**
A Comunidade Firebird de Língua Portuguesa foi autorizada pelo Autor do Original para elaborar esta tradução.	