MIGRANDO PARA O AMBIENTE CLIENTE/SERVIDOR

A. Elementos de Programação de Banco de dados

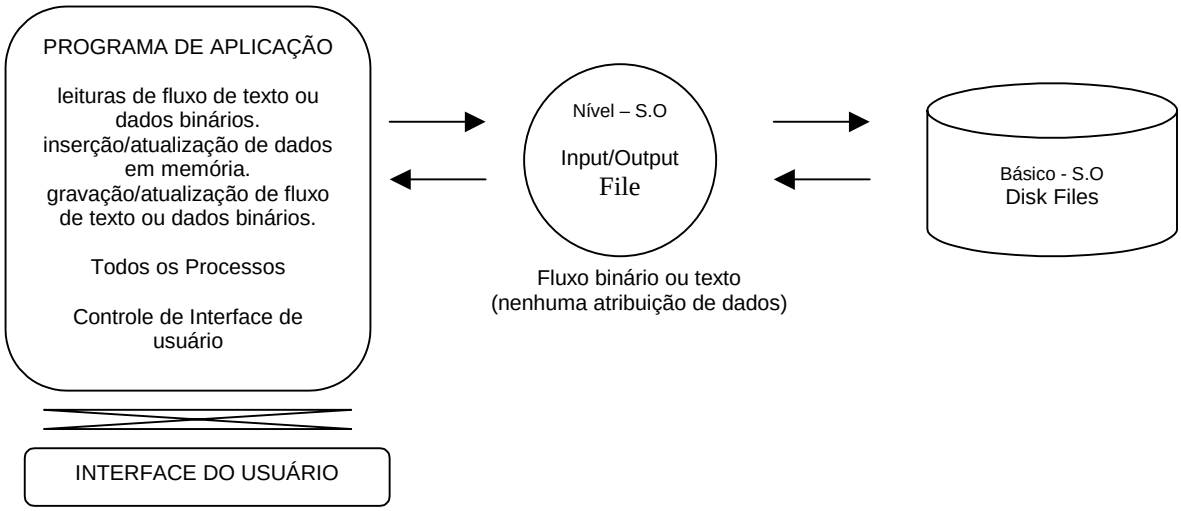
É útil começar examinando os elementos primitivos de qualquer ambiente de aplicação de banco de dados.

1. Um banco de dados. Simplesmente é um repositório de dados em um ou mais arquivos de disco.
2. Um ou mais modos para adicionar ("write") dados para o banco de dados.
3. Um ou mais modos para localizar e atualizar dados que já estão no banco de dados.
4. Um ou mais modos para retornar ("extract", "read") dados que já estão no banco de dados e em produção para usuários.
5. Os usuários (que podem ser as pessoas, ou outros processos).

B. "Flat File" - O Modelo Primitivo

No ambiente de aplicação de banco de dados mais simples, todos os cinco elementos são entrelaçados dentro de um único processo, um programa de aplicação. Este programa define estruturas para criação de dados e envia ao sistema operacional, instruções para armazenar e manter estas estruturas. Também é responsável por manter em sua memória, as regras que estas estruturas se comportam e interagem, bem como, cada parte individual de dados.

Este programa é responsável por manter a interface para a entrada de dados "crus" por um usuário ou um processo externo, por validar e processar aquela entrada, por armazenar, retornar e processar dados do banco de dados, para atualização pelo usuário do programa, ou outro processo.



Este modelo é característico de vários sistemas "velho-estilo", que operam em system-based (baseado em sistema), e/ou, em sistemas de armazenamento de dados proprietário. Eles usam ilimitado formatos de arquivo de disco que são reconhecidos pelo sistema operacional como "data files" (arquivos de dados). Para distinguir estes, com sistemas de banco de dados posteriores, mais estruturados, o modelo foi denominado de "flat file" (arquivo plano). O modelo "flat file" (arquivo plano), ainda é usado amplamente em muitos sistemas operacionais comuns, especialmente Unix. O Delphi realmente, nunca interagiu mais do que suporte de leitura, pela BDE, para formatos de dados DOS-baseados em suas arquiteturas de acesso à dados, embora seus antecessores, Turbo Pascal e Object Pal o fizeram.

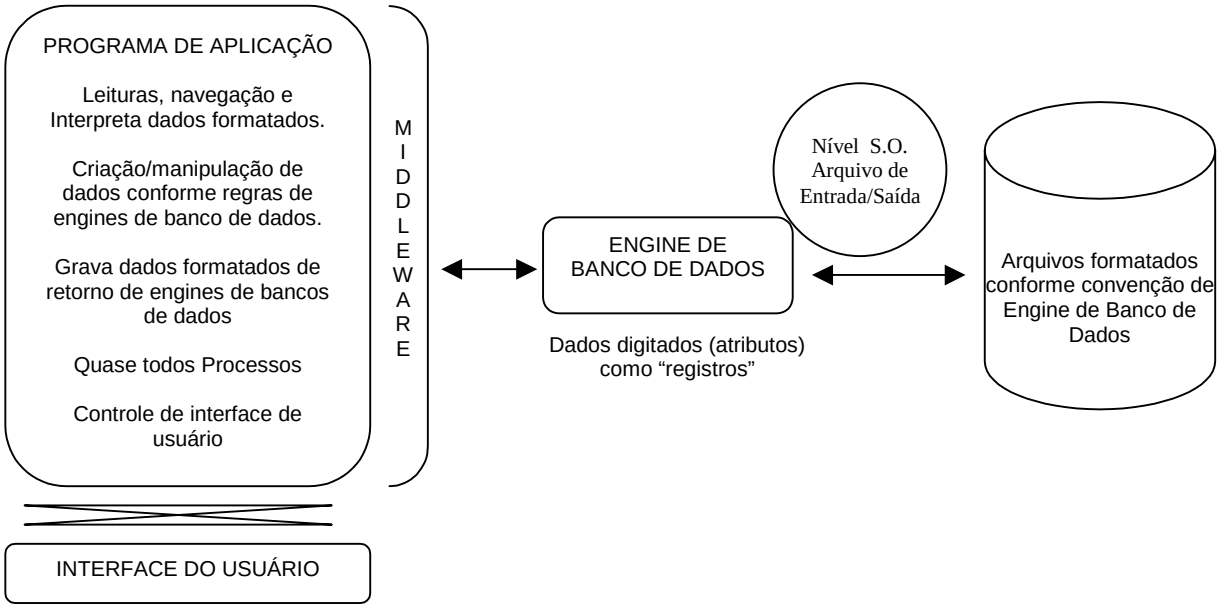
Desnecessário dizer, que o modelo de "arquivo plano", não é interessante a quaisquer dos benefícios de reutilização de código e desenvolvimento dirigido à objeto, como os que a geração atual de desenvolvedores, esperam num determinado ambiente de desenvolvimento.

C. Engines de banco de dados

Bancos de dados formalmente estruturados, com estruturas físicas controladas por um engine separado, removem muitas das responsabilidades do programa de aplicação. Porque o mecanismo para definir as estruturas e regras dos dados, é mantido independentemente da camada de aplicação, o programa de aplicação só precisa saber reconhecer essas regras e como entregar e receber dados.

Fica possível então, separar uma aplicação de banco de dados em várias camadas. A camada de armazenamento de dados é administrada pela engine do banco de dados. A camada de drivers e/ou API, são introduzidas para permitir a comunicação de um modo genérico, entre o banco de dados e um ou mais programas de aplicação. O próprio programa de aplicação, já não precisa definir estrutura de disco ou administrar a entrada e saída à nível físico. Precisa conhecer apenas a forma de comunicação com a camada de transporte de dados, e interpretar os atributos dos pacotes de dados recebidos e enviados.

Modelo de aplicação de banco de dados Arquivo/Servido (File-Served)



O Delphi tem camadas adicionais deste modelo encapsulando a maioria deste "formulae" em suas classes de acesso de dados. As "nuts and bolts", são protegidos dentro da classe e só apareceram através de propriedades e métodos. Com a ajuda da BDE e drivers que estão vinculados aos atributos distintos e regras específicas da engines de banco de dados, o Delphi ameniza as diferenças e fornece uma camada de acesso de dados que é genérica, não só para todas as aplicações que têm acesso uma certa engine, mas para aplicações que têm acesso a qualquer engine que tem o drive BDE. ODBC que ameniza diferenças de engine "externas", pode apresentar engine de dados não suportado, pela camada de acesso à dados do Delphi .

O termo genérico para estes acessos de dados e camadas de transporte é "middleware". Como você está indubitavelmente atento, a "middleware" do Delphi é mas uma constelação de uma galáxia de modelos "middleware".

D. A Perspectiva Atual

Provavelmente, as engines de banco de dados que você trabalhou com o Delphi, estavam neste nível.

A menos que você já usou TQuery, ter acesso ao banco de dados era de uma forma direta, abrir os registros em uma tabela física em seu disco local, para locais físicos distintos (campos)

dentro de cada ou quaisquer desses registros. Você usando TTable, com ou sem um componente de TDatabase explícito. Acessa a tabela pela sua localização, abre-a e outros arquivos que contêm índices, constraints de integridade e assim por diante. Tal banco de dados é denominado "file-based" (baseado em arquivo) e a arquitetura da aplicação "file-served" ou "file-server".

Para trabalhar com dados em uma tabela, você permitiu para os usuários decompor tabelas inteiras em memória local. A estrutura em memória, imita a estrutura física dos registros da tabela e campos no arquivo de disco. Para uma aplicação de único-usuário com acesso exclusivo para estes arquivos físicos no local, isto é bom e, com drive nativo, também é rápido. É a solução ideal para uma aplicação de banco de dados de único-usuário.

Com este tipo de banco de dados, a aplicação está suportando ainda a maioria do peso de implementar as regras do negócio. Filtros, consultas e movimentação na tabela, são característica de uma aplicação a este nível. Todas as alterações e validação de dados crus, têm que ser executados em memória antes de qualquer tentativa para postar novo ou dados alterados, no banco de dados. Tal uma aplicação se apelida freqüentemente de "fat client".

Se a mesma aplicação requerer múltiplos usuários, acessando uma ou vários tabelas físicas simultaneamente, você precisa monitorar, o travamento de tabelas e registros entre os vários processos e usuários, assegurar que o trabalho de um usuário não interfira no trabalho de outro. Se uma estação de trabalho de usuário se chocar, interrompe outros usuários, enquanto corrompe às vezes o banco de dados.

Você pode ter encontrado problemas de resposta lenta, vários travamentos e corrupção de tabelas, com o tráfico pesado de ir buscar tabelas inteiras pela rede e recursos de rede, causados por desligamento ou crash individual de estação de trabalhos.

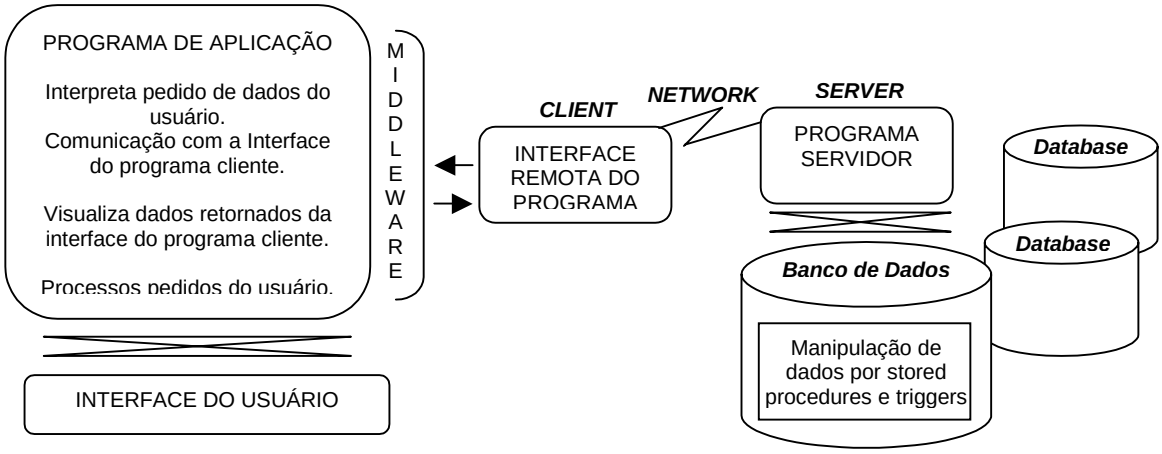
Embora características das camadas-objetos do Delphi, suportam bem o re-uso do código dentro de sua aplicação, o banco de dados ainda é logicamente dependente em seu código de aplicação, implementar as regras do negócio a aplicação precisa assumir - e torna possível - que seu usuário não tenha controle físico exclusivo dos dados, bem como, virtualmente todas as regras do negócio.

E. Cliente/Servidor - O Próximo Nível

Cliente/Servidor introduz completamente novos níveis de delegação e camadas. Considerando que antes, a aplicação e o banco de dados combinados, para formar uma lógica "camada de aplicação" o "usuário" e o "cliente" eram sinônimos, agora o modelo é mais distinto e mais distribuído.

As quatro camadas lógicas distintas são servidor, rede, cliente e o programa de aplicação.

Modelo de aplicação de banco de dados Cliente/Servidor



Servidor

Os Bancos de dados são separados dos programas de aplicação e estão completamente sob o controle de um programa servidor. O servidor pode estar controlando múltiplos bancos de dados e pode haver múltiplos servidores FireBird na rede. O programa de servidor e o banco de dados devem estar na mesma máquina física, e normalmente (recomendado) a máquina servidora não é usada para qualquer outra coisa.

Rede

Todos os dados são empacotados (packages) e tramitam de camada a camada por redes, usando protocolos de rede standards - TCP/IP, NetBeui, SPX/IPX, APPC por exemplo.

Cliente

Todo o acesso ao banco de dados, até mesmo por TTables, envolve declarações através de query a uma camada cliente de banco de dados. No FireBird, esta camada cliente sabe como a Interface Remota Cliente, e é implementado na compartilhada biblioteca gds (gds32.dll em Windows, gds.so em Unix e Linux). O cliente gds se mantém na estação de trabalho do usuário, ao lado de um ou mais programa de aplicação, e se comunica com o servidor através de um protocolo de rede. O programa cliente pode estar administrando conexões múltiplas para o usuário, para um único banco de dados ou para bancos de dados múltiplos. Também pode estar administrando conexões múltiplas para usuários múltiplos, em um única estação de trabalho ou de conexões remotas para um programa servidor de aplicação, para um ou mais bancos de dados.

O cliente gds responde a um comando CONECT negociando uma conexão entre o usuário requerente e o banco de dados especificado no servidor. A virtally pede toda a outra aplicação responde criando um contexto de transação no servidor, enquanto empacotando o pedido do usuário, enquanto passando o pedido ao servidor e escutando para respostas de servidor relativo àquele contexto de transação.

Administra o fluxo de dados retornado, em resposta a declarações SELECT, valida parâmetros que alguns pedem ou todas as linhas especificadas do servidor. Comandos INSERT, UPDATE, chamam stored procedures e dados definidos em query, responde empacotando a query e entregando para o servidor com um pedido EXECUTE.

Aplicação

Todas as negociações do programa de aplicação com o banco de dados são pelo cliente gds. Programas de aplicação nunca tocam fisicamente o banco de dados. Não há nenhum mecanismo para eles por assim fazerem. Um único usuário pode ter várias conexões ao

servidor sob a administração simultânea do cliente gds, ou, realmente, conexões para mais de um banco de dados.

NOTA: Interbase tem uma capacidade de "servidor local" para conveniência durante o desenvolvimento, mas não é planejado para desenvolvimento em produção. Em versões até e inclusive Interbase 6.0.x, o cliente de gds não está linha-seguro em uma conexão local.

A interface remota cliente, pode ser implementada em uma única máquina cliente/servidor conectando ao programa servidor por LOCALHOST usando um protocolo de rede. Desempenho à parte, tal configuração se comporta como se o cliente e o servidor estão em máquinas diferentes.

F. Delegação de Trabalho para o Servidor

Se você já tiver usado o FireBird, Oracle ou Microsoft SQL Server com a BDE e a VCL nativa, então sua aplicação de Delphi tem operado em modo cliente/servidor sem você precisar estar atento disto. O trabalho da BDE é aplainar as diferenças significantes entre bancos de dados arquivo-baseados, como Paradox e dBase, e bancos de dados cliente/servidor.

· O benefício para programadores é que estes bancos de dados cliente/servidor são feitos para se comportar como se fossem arquivo-baseados.
· O lado ruim, claro, é que esta emulação nega o acesso do programador à maioria das capacidades desses bancos de dados cliente/servidor.

Um das diferenças principais entre uma força-industrial de banco de dados servidor-baseado, como o FireBird e os tipos arquivo-baseados, são a habilidade para construir todas as regras do negócios no banco de dados, usando triggers e stored procedures. Esta delegação de trabalho requer uma visão completamente nova de aplicação de banco de dados e interface-usuário, se você quer que seus usuários sejam beneficiados. Porque virtualmente todos o dados-"crunching" e um nível alto de validação pode acontecer agora no servidor, programas de aplicação podem ser completamente menores e dedicados a prover uma interface de usuário amigável, eficiente.

Funções Definidas pelo Usuário - UDF

O FireBird disponibiliza a capacidade de escrever seu próprio costume, bibliotecas de função servidor-baseadas e embute a chamada deles/delas conecta direito dentro do banco de dados, para uso em SQL e declarações de idioma de procedimento. O FireBird transporta com a biblioteca de IB_UDF/IB_Utils que contém muitas matemática, strings e funções de data. Outras bibliotecas de UDF estão livremente disponíveis no domínio público. Você pode escrever suas próprias bibliotecas de função customizadas em C, C++ ou Delphi.

G. Dados como Sets, não Tabelas

A "Mentalidade" de Planilha eletrônica

Suas velhas aplicações com TTable, tipicamente preenchem tabelas inteiras para procuras e atualizações. Você depende de métodos como Locate, Next e FindKey para fornecer usuário o deslocamento nos registros. Você reduz a extensão da visão do usuário através de filtros e limites.

Porque TTables não pode envolver dados de múltiplas tabelas, enquanto os usuários pediam que você projetasse tabelas, que estavam como planilhas eletrônicas. Você tendeu a projetar estruturas de banco de dados com redundância, para evitar confiar em dispositivos cliente-laterais caros como campos lookup e calculated, para alcançar dados que deveriam ter sido armazenados em estruturas mais normalizadas. O controle que você usou na maioria, era o grid porque representava melhor sua percepção destas planilha eletrônica - iguais dados

físicos - registros e campos. Isso era efetivo porque o lay-out do grid parece com o lay-out físico dos arquivos de banco de dados arquivo-baseados.

Cliente/Servidor não tem nenhum lay-out assim. Armazena sets de dados, tipicamente um ou alguns sets que, se eles fossem de forma tabular, representaria uma ou algumas linhas mapeadas ao metadata de uma tabela. Um índice é outro set de dados. Um blob é outro. A engine aloca tipos diferentes de "páginas" para armazenar os vários tipos de sets. Fisicamente, páginas de banco de dados não são sob forma de tabela. Se você criar um banco de dados com um tamanho de página de 8Kb, então cada página de dados de tabela pode armazenar um ou mais linhas potencialmente em uma tabela - como muitos poderiam ser alocados em 8Kb sem precisar expandir.

Não são criadas páginas até que seja necessário e, uma vez criada, uma página pode ser armazenada virtualmente em qualquer lugar no disco. Se a engine do banco de dados não puder achar nenhum espaço desocupado em uma página existente, gera um novo e o grava simplesmente onde quer que haja espaço. Jogos de filas, índices e gotas que " pertencem " à mesma mesa são improváveis ser contíguo no plano físico no disco.

Pensando em termos de "sets de dados" em lugar de "registros de tabelas" é importante para projeto de tabelas e para visualização dos dados, que você retornem em sua aplicação através de declarações SELECT. Um RDBMS completamente caracterizado, como o FireBird precisa de uma estrutura muito abstrata. Em contraste com suas estruturas arquivo-baseadas, onde você tende a construir tabelas para construir especificações de saída, a estrutura precisa para representar as relações entre grupos de dados, fazer isto fácil e ir buscar sets econômicos.

H. Uso e Abuso de Grids

Por que Uso Grids?

TTable e grids tendem a encorajar a percepção que todos os bancos de dados consistem em tabelas que são linha em linha de registros correspondentes, depositados nos arquivos em uma seqüência ascendente. Aquela percepção pode ser apropriada para bancos de dados arquivo-servidos que são dispostos fisicamente como tabelas. O Delphi provê vários métodos muito úteis para TTables e grids habilitarem o desenvolvedor, para fazer bancos de dados se comportar como planilhas eletrônicas. Este UI força os usuários à procurar e escolher. Uma indústria lucrativa existe a construção de componentes grid ao redor para fazer esta atividade tediosa mais suportável.

Para bancos de dados locais, o Delphi nativo tira proveito do fato que os registros nestes arquivos tabelar têm números seqüentes internos que se referem através de índices. São copiados índices para a memória, enquanto habilitando blocos grandes de registros ser identificado por ordenar e mover de um lado para outro do disco à memória. Apesar do I/O contínuo, o custo de desempenho é baixo quando os arquivos de banco de dados forem locais à aplicação.

Grids para Clientes Remotos

Para aplicações de cliente remotas, podem ganhar realmente grids no qual continue exibindo listas de seleção. Um grid de front-end para uma questão como

Select CustomerID SELETO, CustomerName From Cliente

é ideal para prover um lista de escolha somente de leitura para uma Customer Maintenance ou Order Processing. Pode estar de pé ao lado de uma exibição de linha simples em cima de outra questão, refrescada por um grid scroll event, como em

Select * From Cliente

Where CUSTOMERID = :CUSTOMERID

A lista de escolha (pick list) pode servir para mais de um propósito. Na mesma forma, por exemplo, a pessoa poderia ter um grid em cima de um customer orders query, no qual prover um lista de escolha para um drill-down para order items, invoices, paymentes, etc. A query para apicklist de Orders poderia ser algo como:

Select OrderID SELETO, OrderDate, Descrição From Orders

Where CUSTOMERID = :CUSTOMERID

Order By ORDERDATE DESC

Esta aproximação trabalha eficazmente, com um baixo overhead no both cliente e rede e uso econômico de bens imóveis de UI, até mesmo quando a tabela Customer é muito grande.

Compare as implicações de um grid editável:

Select * From CUstomer

Por que Evitar Grids?

- *Engasgamento de rede*

Quando estes arquivos tabelar vivem em um drive de rede compartilhado, alguns dos benefícios de custo deste modelo desaparecem com o trocar do contínuo I/O sobre a rede. Refreshing de scroll de grids como usuários, constantemente mantém a rede acentuada, enquanto transportando números grandes de pacotes em demanda. Embora cada usuário na verdade alguns precise, o UI força todos os usuários a requisitar a trazer cada tempo uma ocorrência de evento de scroll. Mais os usuários, o mais baixo desempenho. Em uma aplicação com um UI grid-driven, grid-refreshing dominam a rede.

Acontece freqüentemente aquele tempo de resposta pobre está curado, dando poder para cima de bandwidth na rede em preferência para melhorar o design ruim. É comum para desenvolvedor culparem o BDE ou o middleware alternativo do próprio banco de dados, uma vez por desempenho pobre, às interfaces de grid quando entram em produção.

- *Mal Desenho de Tabela = Pobre Otimização de Query*

Como um banco de dados servidor, o FireBird foi projetado para criar result set fixo muito eficazmente. Suas camadas de parser internas otimizam as declarações de SQL desajeitadas, genéricas e usa algoritmos complexos para decidir como extrair os sets pedidos. Em um banco de dados bem afinado, o optimizer do FireBird é muito eficiente. Infelizmente, o mal desenho de banco de dados e índices redundantes podem incapacitar o optimizer muito facilmente. Estruturas de tabela que foram projetadas para povoar grids são tipicamente fatais à própria otimização de querys.

- *Selecione * = mal modelagem*

Uma declaração que seleciona todas as colunas de uma mesa é a companheira natural do grid e o inimigo natural de uma rede de cliente/servidor eficiente. Sinaliza a probabilidade de redundância de dados (Select * From de tabelas corretamente normalizadas, geralmente não retorna um set que bem visualiza em uma exibição de grid!) Uma declaração como

- *Select * From ATABLE*

na propriedade de SQL de um poderoso dataset bem como seja um TTable.

A essência do bom desenho de banco de dados para consultas remotas, é um bom modelo abstrato que pressupõe uma engine de banco de dados, que possa tirar proveito de relações altamente normalizadas. Tal engine prover apoio completo para JOINS, subselects e integridade referencial em cascata. O FireBird é semelhante a uma engine. Tabelas que são projetadas para povoar grids por Select * ou por via de TTables não utilizam tais características de banco de dados.

Não é a intenção desta nota ensinar os fundamentos de uma boa modelagem de dados. Há riqueza de literatura disponível a ajudar o desenvolvimento cliente/servidor como newcomers - ou "seat of the pants" old-timer! Olhe para os oferecimentos da Livraria da pagina IB Objects na web site (www.ibobjects.com). Um livro particularmente excelente é Data Modelling: Analysis, Design and Innovation (Second Edition) by Graeme Simsion and Graham Witt (Coriolis Books, 2000). Parte deste livro é um lúcido tratamento dos fundamentos, para modelar suas regras de negócios. Os capítulos restantes, passam a aspectos mais avançados e consideram alguns dos aspectos de modelar um banco de dados relacional, para uso em um ambiente de desenvolvimento objeto-orientado. É certamente um bom investimento .

Um outro essencial para a eficiência de querys remotas, é uma indexação inteligente. A próxima seção olha para alguns aspectos importantes, a serem considerados ao criar seu banco de dados cliente/servidor.

I. Indexando

Índices versus Chaves

- *Índices e chaves não são a mesma coisa.*

Uma chave é uma coluna, ou set de colunas que identificam uma linha. Uma chave UNIQUE é uma coluna, ou set de colunas em torno das quais o banco de dados, obriga uma regra que um certo valor, não deve se aparecer mais de uma vez nesta coluna ou set de colunas. Uma chave primária deve ser sem igual e o FireBird obriga isto.Também obriga a uma constraint NOT NULL em chaves primárias. Se você veio de Paradox, isto pode causar um problema porque Paradox não suporta NULL. Ao invés, deixa como DEFAULT para zero-equivalente, qualquer valor para colunas que seriam NULL no FireBird.

Pense cuidadosamente aproximadamente se você deveria permitir NULL em chaves estrangeiras. Tem um pouco de efeitos indesejáveis, não menos o risco de linhas details "órfãs" sob algumas regras de integridade referencial, por exemplo ON UPDATE ou ON DELETE.

Um índice é uma estrutura interna que o banco de dados mantém para uma tabela organizar os valores contida em uma coluna, ou set de colunas (conhecido como uma "combinação") em uma certa ordem de precedência. Fisicamente, um índice é um tipo de self-referencing da tabela. Logicamente, é uma árvore binária. Um índice pode ser UNIQUE e o FireBird não permitirá criar um índice em uma coluna ou combinação que faltam a constraint NOT NULL.

O FireBird cria índices ascendente em cima colunas de chave primária (primary key) e colunas de chaves estrangeiras (foreign key) automaticamente. É importante estar atento a isto porque, se você criar um índice idêntico, o otimizador da query não podem usar qualquer um dos índices.

Index Fobia

Esses cuja experiência principal esteve freqüentemente com bancos de dados arquivo-baseados trazem com eles um tabu contra usar índices. Com estes bancos de dados, toda mudança para um índice requer uma operação de entrada/saída a um arquivo físico separado e o enorme overhead de manter estes índices justifica esta reserva.

Com cliente/servidor, o tabu é desnecessário. Índices bem-projetados são o óleo que faz o RDBMS executar suavemente. O FireBird mantém páginas de índice junto com outras páginas dentro do banco de dados sem o overhead de entrada/saída do arquivo de sistema operacional e você têm nenhuma razão para evitar índices como um modo para melhorar o desempenho.

Pelo contrário, evitando indexar é um modo efetivo para destruir o desempenho. Em bancos de dados arquivo-baseados, você pensou em índices como um modo para definir ordenação para suas saídas. Em cliente/servidor, todos os pedidos de dados são através de querys. Índices em colunas de não-chave são importantes para ordenar saídas. Porém, eles têm um maior papel para representar nos passos do otimizador de query, leva para ajuntar o set intermediário fixo que usa para extrair os dados pedidos. O uso (ou abuso) de índices, pode ser a diferença entre uma query ser segundo-substituto e levar 10 segundos ou mais muito tempo.

O assunto de criar índices bons é um assunto por si só. Não obstante, algumas táticas de indexação que impactam mal desempenho são valor mencionados aqui.

Seletividade de Índices

Um das características de índices é a seletividade. Um índice é altamente seletivo se nenhuma de suas partes é duplicada freqüentemente. Um índice UNIQUE tem a seletividade mais alta porque não ocorre nenhuma duplicação de partes. Um índice em uma coluna que tem uma gama pequena de possíveis valores, como Account_Type em uma tabela detail de Fatura, tem muita baixo seletividade. Isto causa a árvore de índice a formação de cadeias longas de partes duplicadas quando novas linhas são inseridas. Desde que a operação índice-buscando envolve partes minuciosas para eliminar entradas não-emparelhands, enquanto navegando estas longas cadeias duplicadas tipicamente reduz a velocidade de procura, tanto que uma procura não-indexada -"ordem natural" - seria exponencialmente mais rápida.

Um modo para solucionar o problema de baixa seletividade é não usar nenhum índice nesta coluna. Uma solução melhor (leia-se "mais rápido, mais efetivo") é elevar a seletividade deste índice. Faça um índice composto que junte coluna de baixa-seletividadecoluna com outra coluna de seletividadealto. Para nosso exemplo de tabela detail Invoice, este índice composto seria ideal:

```
CREATE UNIQUE INDEX XU_ACCOUNT_TYPE
ON INVOICEDETAIL(ACCOUNT_TYPE,INVOICEDETAIL_ID);
```

Para índices compostos, a ordem de coluna é significante. Para a elevar a seletividade, coloque a coluna de baixo-seletividade primeiro.

Sugestão: Evite Chaves Estrangeiras em Colunas de Controle

Não coloque uma constraint FOREIGN KEY neste tipo de coluna. Se você deseja updates e deletes em cascata de tabelas de controles à qual este tipo de coluna se refere (por exemplo Account_Type em nosso exemplo), codifique o comportamento de cascata em triggers.

Índices não balaceados

Índices são estruturas de árvore binárias. Eles trabalham eficazmente quando a árvore inteira consistir em nodos que são distribuídos uniformemente à esquerda e direita ao longo dos nodos de seus pais. No curso normal de eventos, quando transações estão postando suplementos e apagam ao acaso, o equilíbrio é mantido. Objetos obtidos de varreduras normais ao cuidado de nodos que saem ligeiramente de equilíbrio.

Grandes lotes de inserts ininterruptos para uma única tabela podem causar desequilíbrios nos índices, como são acrescentados nodos consecutivamente a nodos sem qualquer varredura. Quando a manutenção dos lotes terminar, os usuários notarão que os seus acessos para a

tabela ficam muito lentos. O lentidão freqüentemente fica evidente até mesmo antes do lote terminar - uso de CPU sobe a 90% ou mais alto e o processo parece parar.

Por isto, normalmente é uma boa idéia dar acesso temporariamente exclusivo para a tabela, em um processo de um grande lote de insert sucessivos, e inativar temporariamente os índices, assegurando que o processo proceda em velocidade máxima.

Se acesso exclusivo não for uma opção, você pode restabelecer um índice para equilibrar depois que a operação de grupo terminar, executando um script de administração para desativar e re-habilitar os índices.

Sugestão: Se, com o passar do tempo, o banco de dados inteiro ficou lento por razões irresponsáveis, índices não balanceados estão quase certamente entre as causas. Você pode re-energisar o banco de dados inteiro fazendo um shutdown controlado, executar gbak backup e restore e reiniciar usando o restore da cópia.

J. Dependências de Metadata

Em seus bancos de dados desktop, você terá sido usado a dependências em seu metadata. No Paradox, por exemplo, tabelas têm as famílias de arquivos com o mesmo nome de arquivo e uma variedade de sufixos de arquivo, por exemplo Employee.db (os arquivos de dados), Employee.px (o arquivo de chave primária), Employee.mb (o arquivo que contém dados blob) e vários Employee.??? arquivos que contêm índices secundários, regras de validação e assim por diante.

Por causa dos arquivos Paradox, serem arquivos de sistema operacional "ordinários", é muito fácil de corromper o bancos de dados ou perder referências de dependência, apagando inadvertidamente ou re-escrevendo nestes pedaços.

No FireBird, todos o metadata do banco de dados são armazenados dentro de um arquivo de banco de dados, junto com os dados. Este metadata são armazenados em tabelas de sistema, tudo com o prefixo RDB$.

Quando você desenvolve suas definições de banco de dados, o FireBird armazena detalhes de todas as dependências. No idioma de definição de dados (DDL) , ocorre comandos DROP (delete) e ALTER (altere), para muitos dos tipos de objeto de banco de dados que você cria. Às vezes, será impedido de executar tal comando por causa da dependências existentes entre as estruturas dos objetos.

Adicionalmente, será impedido às vezes de executar comandos de manipulação de dados (DML) declarações por causa de dependências nos dados.

Infelizmente, as mensagens retornadas de uma violação de dependência às vezes não são muito informativas. Porém, várias ferramentas de tabela, inclusive o grátis IB_WISQL que você pode baixar de www.ibobjects.com, podem exibir as dependências para todos os seus objetos de metadata. Isto pode ser muito útil quando você for diagnosticar uma violação de dependência.

Em IB_WISQL, abra o browser e selecione um objeto. Clique na aba de dependências para exibir a informação.

K. Triggers

Talvez um das maiores diferenças entre o banco de dados arquivo-servido e o mais poderoso sistemas de RDBM cliente/servidor, são a superior habilidade de executar linguagem de procedimento embutido, (P/L) - código; em relação a uma linguagem de manipulação de dados (DML) - evento. Este código embutido pode ser escrito completamente pelo programador depois que uma tabela for criada, como triggers procedures (usualmente chamada trigger) ou

pode ser escrito pela própria engine de banco de dados durante criação da tabela, como parte do processo de aplicar constraints de integridade referencial na tabela.

Triggers tornam isto possível para então delegar a execução da maioria, se não tudo, da validação e criação de dados como regra geral do servidor. Esta característica só pode ser responsável por uma redução significativa em código do ladoo-cliente. Também permite ao programador escrever aplicações de múltiplas clientes, sob o mesmo banco de dados sem sempre ter que preocupar sobre consistência na aplicação das regras de negócio.

Triggers podem ser tão simples, como buscar um valor de generator para povoar uma chave primária, bem como para aplicar cálculos complexos a mesma ou outras tabelas baseado nos valores dos conteúdos submetidos na declaração DML.

O FireBird provê uma ordem rica de possibilidades para a cronometragem de processamento de triggers. A pessoa pode embutir código de trigger para ser executado ANTES (before) e DEPOIS (after) DE INSERT, DELETE ou UPDATE; e todas as Triggers têm um parâmetro de POSIÇÃO (position), de forma que o processo da trigger pode ser desviado da sucessão ordenada do processamento de eventos.

A trigger específica as colunas pelas variáveis: NEW. e OLD. São avaliadas para distinguir entre valores novo submetidos na declaração (insert e updates only) e os valores que estavam antes da transação começar (update e delete apenas).

Abaixo um exemplo de uma trigger simples que testa se a coluna da chave primária tem um valor válido e, se não, vai buscar um valor de generator a ser usado:

```
CREATE TRIGGER BI_ATABLE_0 FOR ATABLE
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF ((NEW.ID < 1) OR (NEW.ID IS NULL)) THEN
    NEW.ID = GEN_ID(AGENERATOR, 1);
END
```

Triggers Que não Executam

Uma reclamação comum de programadores é "Minha trigger não teve nenhum efeito". A causa mais comum é escrever Triggers AFTER (depois de) que tentam mudar valores de coluna na sua própria linha. Apenas Triggers BEFORE (antes de) podem mudar ou podem criar valores de coluna para a linha em operação. Use Triggers AFTER para operações de INSERT ou UPDATE que você quer fazer em outras tabelas, quando está completada. Um uso típico de um Triggers AFTER é para escrever linhas para um arquivo de log. Também use Triggers AFTER para efetuar UPDATES ou DELETES em cascata, por exemplo:

```
CREATE TRIGGER AD_CANCEL_ACCOUNT FOR ACCOUNT
ACTIVE AFTER DELETE
AS
BEGIN
  UPDATE ORDER
  SET ORDER_STATUS = 'CANCELLED'
  WHERE ACCOUNT_ID = OLD.ACCOUNT_ID;
END
```

Tal uma operação poderia ativar uma trigger na tabela afetada, por exemplo:

```
CREATE TRIGGER AU_CANCEL_ORDER FOR ORDER
ACTIVE AFTER UPDATE
AS
```

```
BEGIN
  IF NEW.ORDER_STATUS = 'CANCELLED'  THEN
  BEGIN
    DELETE FROM ORDER_DETAIL
    WHERE ORDER_ID = OLD.ORDER_ID;
  END
END
```

IB Objects e Triggers

Aplicações cliente em geral e os componentes IB Objects, não podem ter acesso ao código de Triggers em particular porque é compilado dentro do banco de dados. Porém, IB Objects provêem alguns mecanismos para permitir para o programador alertar a aplicação "sobre" Triggers.

Por exemplo, a nativa Delphi VCL, requer um valor não-nulo (not null) para todas colunas NOT NULL. No caso de uma coluna povoada por um gerador, IBO tem a propriedade GeneratorLinks e o método GenID(), que ocupará o novo valor novo, cuida no momento certo no lado de cliente, evitar um erro de 'campo requerido'. Esta é a aproximação indicada.

Porém, se você realmente quiser que suas triggers povoem sua chave primária, IBO o deixa fazer isto ao invés de fazê-lo. Para componentes de acesso à dados TIB_ * (IBO nativo) , simplesmente vão para a página de Coluna de Atributos do editor de campos e fixam o atributo EXIGIDO (required) Falso (false) para aquela coluna. Para componentes TIBO * (TDataset-compatível), você pode fixar isto modificando o atributo em execução.

L. Cascatas

Cascatas - integridade de constraints em cascastas - será novo para você se você estiver trocando para FireBird de desktop ou um RDBMS de peso leve como o MySQL.

Você pode ter usado previamente integridade referencial de constrainst em Paradox. No FireBird, você pode opcionalmente definem cláusulas constraints que causam as tabelas referenciadas relações para atualizações (updates) e deleções (deletes) automaticamente, quando atualiza ou apaga na tabela de referência. O conceito de "cascata" vem do fato que este processo automático pode fluir por cadeias de relacionamentos de tabelas referenciadas. Cascateando atualizações especificam que, quando a chave primária referenciada pela constraint chave estrangeira é atualizado, a mudança será copiada a todo custo na coluna da chav estrangeira. Cascateando deleções especificam que, quando uma linha na tabela referenciada é apagada, toda linha na tabela cuja coluna da chave estrangeira se refere, também será apagada.

As subcláusulas de cascata são aplicados à declaração da CHAVE ESTRANGEIRA da tabela referenciada e levam a forma:

ON UPDATE CASCADE

ON DELETE CASCADE

Consulte o FireBird Data Definition Guide para adquirir a sintaxe exata para cascatas.

Também note que CASCATA não é o único comportamento ON UPDATE/ON DELETE que você pode especificar. NO ACTION, SET DEFAULT (para alterar todos valores de coluna da chave estrangeira a um valor válido que é certo estar presente na coluna da chave primaria referenciada) e SET NULL (para alterar todos valores da coluna da chave estrageira para NULL, válido se a coluna da chave estrangeira permitir NULL).

M. Stored Procedures

Um intervalo principal do trabalho realizado de um arquivo-servido (back-end) é a sua habilidade para se mudar processamento de dados completamente da aplicação cliente, delegando o trabalho ao servidor como procedimentos armazenados. O benefício de desempenho em redes é óbvio - já não necessita datasets para processar e ser transportado de um lado para outro pela rede entre o banco de dados e a memória do cliente.

Alguns dos benefícios menos óbvios incluem:

- Funcionalidades para processar dados e validação são centralizadas no banco de dados, enquanto lhes permitindo estar constantemente disponível para qualquer aplicação

- Alterações ou adições para regras de negócio ficam disponíveis simultaneamente a todas as aplicações

- Em desenvolvimento grande, é atribuída responsabilidade para implementar regra de negócio facilmente a um único indivíduo ou grupo de projeto

- Aplicação Cliente o código é reduzido, trazendo desenvolvimento menores e baixo overhead nas estações de trabalho

A linguagem de procedimentos do FireBird (P/L) é simples, estruturado e fácil de aprender. Inclui a valiosa capacidade para gerar datasets de múltiplas-linhas como saída. Se você teve que lutar para criar sets de saída calculados em outros sistemas RDBM, você apreciará o valor disto! Tal procedimentos armazenados são freqüentemente chamado "selecionáveis" porque você pode usar uma declaração SELECT para os chamar, por exemplo

SELECT <OutputColumnList> FROM MyProc(<InputList>)

Classe IBO - Suporte para Procedimentos Armazenados

O IBO tem classe de componentes de procedimentos armazenados (TIBOStoredProc e TIB_StoredProc) mas, no caso de procedimentos selecionáveis, TIBOQuery ou TIB_Query podem ser usados como objeto de acesso aos dados para gerar o set de saída navegável. O dataset pode ser criado "ao vivo", também, aplicando suas propriedades UpdateSQL.

Execução de processos

Podem ser delegadas execução de processos periódicos, como contas de fim de mês, completamente ao servidor, aninhando chamadas de procedimento armazenadas. Por exemplo, você poderia escrever um procedimento armazenado que é efetivamente um "batch script" para conectar com a UI, aceitar parâmetros de usuário e devolver um código de sucesso final. Este procedimento testaria condições e executaria outros procedimentos, passando e recebendo entradas e retornando parâmetros de acordo com as condições.

Avaliação para o Cliente

Além do sistema integrante de códigos de exception/error, o FireBird usa para comunicar com o programa cliente de gds (e por isso com a aplicação cliente), o FireBird tem duas características de interface para prover avaliação a usuários de procedimentos armazenados.

1 - a exceção definida pelo usuário, que você cria em metadata para uso em qualquer procedimento armazenado, por exemplo

```
CREATE EXCEPTION PERIODO_INVALIDO ' O período especificado não foi definido';

...
```

```
if (not(exists(select PERIOD_ID from PERIOD where PERIOD_ID = :InputPeriodID))) then
    begin
            EXCEPTION PERIODO_INVALIDO;
          EXIT;
      end                          / * Caso contrário, continue pelo procedimento * /
       ...
```

Neste exemplo, processando o procedimento armazenado até o final, serão desfeitas todas as mudanças já executadas e a mensagem de texto associou com a exceção será retornada ao cliente. Sua aplicação pode ler a propriedade de ErrorMessage do EIB_Error retornado (ou EIBO_Error) em uma bloco try...except e controlar a própria exceção, por exemplo executando um rollback na transação, enquanto solicitando ao usuário para entrar com um parâmetro válido e chamando a stored procedure novamente.

Como a extensão de um bloco de exceção é o begin...end, é possível, se apropriado, apanhar a exceção dentro do próprio procedimento armazenado e controlar silenciosamente, enquanto usando a declaração WHEN. Neste caso, exclua a declaração EXIT e permita o controle para passar ao próximo bloco de código. Consulte o Guia de Definição de Dados para detalhes de sintaxe e sugestões adicionais para manipulação de exceção de lado-servidor.

2 - O Alerta de Evento permite passar a avaliação de dentro da aplicação cliente, a um procedimento armazenado postando um evento de servidor. O IBO deixam sua aplicação "escutar" até 16 destes eventos de servidor (o máximo permitido pelo cliente de gds) por via do objeto TIB_Events. A declaração no procedimento armazenado ou trigger para postar um evento é muito simples:

```
POST_EVENT ' Passo 4 completo';
```

NOTA: É limitado o comprimento da string de evento a 15 bytes, um fato que precisa ser considerado ao usar caracter sets multi-byte.

Podem ser definidos eventos para qualquer coisa - erros, fases de processamentos, atualização de tabelas ou colunas, etc. O DML caching do IB Objects provê um exemplo excelente de usar eventos para manter os exemplos de aplicação de usuários individuais sincronizados em um ambiente multi-usuário.

N. Transações

Controle de transação

O FireBird, como muitos outros bancos de dados cliente/servidor, provê controle de transação para proteger a integridade de dados no ambiente multi-usuário onde muitos usuários estão requesitando simultaneamente para adicionar, atualizar e apagar dados das mesmas tabelas. Controle de transação protege dados de mudanças intempestivas, protege o trabalho de um usuário ser modificado por outros e pode comunicar resultados ao cliente.

Fisicamente, uma transação é uma unidade discreta de trabalho que pertence a um único usuário e pode executar em paralelo com outras transações mas que também pode ser isolado de qualquer outra transação. Quase tudo o que acontece no FireBird acontece dentro do contexto de transações.

Cada transação física tem toda vida que inicia quando o programa cliente GDS notifica o servidor que tem um "handle" para uma transação e finaliza quando o programa cliente GDS instruir o servidor para COMMIT ou ROLLBACK a transação. COMMIT salva no arquivo de banco de dados físico, todas as DML pendentes ou atualizações DDL especificadas através de declarações na transação; ROLLBACK cancela tudo.

Os IB Objetos têm uma "visão" mais compreensiva de uma transação viva, entre quando começa e finaliza no servidor, nós usamos o termo "transação física" para a parte da transação que acontece no servidor.

Post e Commit

Se você usava antes o Paradox, ou o FireBird sob a BDE, você poderia esperar (injustamente) que o Post e Commit é a mesma coisa. Considerando que uma única conexão de BDE pode suportar só uma transação, as propriedades, métodos e eventos desta transação são mesclados com essa relativa conexão, no componente de TDatabase. As configurações padrões de dados de Delphi nativos têm acesso pela transação e seus métodos. Processamento de transação-relacionada está normalmente "implícito" - significando isto acontecer automaticamente como parte do processamento de datasets.

Em bancos de dados desktop, sem cached updates, o método Post de TDataSet's gravam os dados atualizados fisicamente no arquivo das tabelas. Para transação-controlada pelo bancos de dados como o FireBird, um Post de um dataset despacha um SQL executável (DML) declarações para o cliente para todas mudanças pendentes em todo o datasets, junto com instruções para COMMITar a transação. Assim, o inteiro contexto de transação é totalmente escondida e é possível enganar o programador que o FireBird e Paradox trabalham precisamente semelhantes.

O IBO faz este comportamento opcional de "Post=Commit". Parece quando a propriedade de AutoCommit da transação for True. Se AutoCommit for falso, o Post armazena as mudanças no datasets sem os COMMITar no banco de dados. Sua aplicação tem que chamar o COMMIT em algum ponto, depois de postar e COMMITar as mudanças a todos datasets desde que a transação começou.

Há duas condições onde controlar implícitamente a transação não é possível. Uma é quando são habilitadas atualizações de cached (cached updates). Em Delphi nativo e IBO, o método Post economiza mudanças para o cached, até onde elas permanecerão a menos que as atualizações sejam intencionalmente aplicadas. Para o FireBird, isto significa uma transação física e deve ser iniciada. O método ApplyUpdates posta as mudanças para os datasets e o COMMIT deve ser chamado para os salvar permanentemente.

A outra condição é quando você decide usar transações explícitas.

Transações explícitas

Você já pode estar familiarizado com a transação explícita. Quando você chama o StartTransaction, e os métodos Commit e Rollback do TDatabase do Delphi, lhe foi dado controle explícito sobre a transação mas você não teve nenhuma necessidade para saber o que existe para realizar isto. Você ler a propriedade InTransaction para determinar se uma transação explícita foi iniciada.

No IBO transações explícitas trabalham exatamente como eles fizeram com a BDE, assim se você estiver fazendo coisas aí mesmo, você deve não ter nenhum problema sério, se trabalhar do modo anterior.

Iniciando uma transação explícita altere o comportamento de AutoCommit se foi fixado para TRUE. O comportamento de AutoCommit retoma depois que a transação explícita seja terminada por um Commit ou Rollback (Commit, CommitRetaining, Rollback, RollbackRetaining, Refresh(), Close). O comportamento de AutoCommit não é, porém, restabelecido por um SavePoint que faz todas as mudanças permanente àquele ponto, mas não é tratado como um fim de transação.

Nível de isolamento

Uma transação pode ser isolada de outra transação a uma gama de níveis. O mais baixo é DIRTY READ, onde transações vêem a mais recente versão de toda a linha, inclusive uncommitted o trabalho em desenvolvimento dentro de outras transações. FireBird não suporta "dirty read" em todas.

READ COMMITED - IBO nível tiCommitted - a transação compartilha acesso simultâneo a tabelas com outras transações. Quando a transação começar, pode ver mudanças COMMITadas por outras transações. Continuará vendo mudanças externas recentemente COMMITadas ao longo de sua duração, enquanto provendo uma interface auto-refresh que não exige uma transação nova para mostrar mudanças. Isso é ideal para uma interface de browse para o usuário. É o próprio nível de isolamento para usar ao auto-COMMITAR.

tiCommitted é errado para executar um relatório ou uma exportação porque as mudanças de dados podem causar inconsitência ao relatório.

SNAPSHOT REPEATABLE READ - IBO nível tiConcurrency - a transação atual compartilha acesso simultâneo com outras transações, mas uma vez iniciado, não pode ver mudanças COMMITadas por outras transações a menos que você aplique um hard Commit ou Rollback. É ideal para relatórios ou exportações que precisam poder trabalhar de certo modo com uma visão consistente dos dados. Isso não bloqueará outros de continuar atualizando os dados, contanto que a transação esteja aberta, é garantido adquirir exatamente inúmeras vezes a mesma visão dos dados.

tiConcurrency não é ideal para interfaces de usuário que envolvem browse e editam registros "pt-shot". Como os usuários trabalham nos dados que outros bloquearam ninguém pode ver qualquer outro trabalho commitado.

ESTABILIDADE de TABELA - IBO nível tiConsistency - o nível de isolamento mais restritivo, bloqueia acesso de tabela completamente a outras transações que tentam escrever na mesma tabela, indiferentemente dos níveis de isolamento delas. Uma transação com este nível de isolamento não poderá adquirir acesso à tabela se outra transação estiver escrevendo. Freqüentemente, isto significa que outras transações nem podem mesmo, executar um SELECT em uma tabela para a qual uma transação aberta está segurando uma transação de escrita.

é entendido que tiConsistency garante que se uma transação escreve a uma tabela antes de outra leitura simultânea e escreve transações então terá acesso de tabela exclusivo. Isto deve ser usado com grande cuidado. Tabela bloqueadas geralmente deve ser evitado a menos que você tenha uma razão específica para precisar delas.

O. Programas servidor-baseados

Aplicações de único-usuário

O Firebird não é um banco de dados desktop, mas é perfeitamente possível executar o servidor e o cliente em uma única máquina e desdobrar uma aplicação para uso de um único usuário. Deve-se ter em mente que o banco de dados executa melhor quando não tiver que compartilhar recursos de máquina.

Firebird Local (LIBS) - O Firebird completo sem capacidade de cliente remoto - transportado pela Borland como ferramentas de desenvolvimento durante alguns anos. Um único usuário pode compor quatro conexões diretamente a um local de diretório físico, semelhantemente para o modo de conexão de Paradox, dBase e Access. Era realmente planejado como um modo conveniente para programadores desenvolverem aplicações de banco de dados em uma única, auto-suficiente máquina e ter certas limitações. Por exemplo, porque o servidor de LIBS compartilha que comunicações de espaço de inter-processo com o cliente GDS, o modelo não é seguro.

A utilidade de LIBS diminuiu com a liberação de plataformas Windows que transportam com TCP/IP incorporada no sistema operacional. Podem ser construídas aplicações de único-usuário mais seguras usando a plataforma-interna de TCP/IP host server, LOCALHOST.

Camadas de Servidor de banco de dados

Este é outro reino que diferencia bancos de dados desktop de cliente/servidor RDBMS. Uma aplicação servidor-baseada conecta ao banco de dados por um cliente GDS que é executado no servidor. Clientes remotos conectam a esta camada de aplicação que age como um agente de marshalling para uma variedade de tarefas, de agrupar conexões a executar todas as chamadas de banco de dados em nome de seus clientes. As funções de camada de servidor podem ser divididas em camadas múltiplas, ou "fileiras" (tiers). Podem ser envolvidos vários servidores de banco de dados, junto com um ou camadas mais distributivas (por exemplo Corba) envolvendo conexões a outras redes.

O Asta uma gama de ferramentas para servidores de distribuição em desenvolvimento, suporta o IB Objetos como uma opção de middleware.

Serviço de Aplicações

A habilidade ter aplicações servidor-baseado pelas quais aplicações de cliente remotas podem conectar à servidora de banco de dados, permite uma balsa de possibilidades para segurança e administração. Um serviço de backup, replication, manutenção de usuário e senha e estatísticas de querys são exemplos onde esta aproximação pode ser muito útil.

P. SQL

O que é SQL?

SQL - originando de Linguagem Estruturada de Pesquisa - é uma indústria standard para extrair dados de bancos de dados. Apesar dos padrões, vendedores de RDBMS têm uma tendência forte para implementar os próprios "sabores" individuais. O resultado é que a SQL pode diferir ligeiramente ou drasticamente de um RDBMS a outro. É certo que o Firebird é o mais complacente com o padrão de SQL-92 atual. Embora não implementando tudo representado no padrão, aproximadamente 80 por centavo desses elementos que são implementados estão em conformidade.

SQL Local não é suficiente

Delphi com o BDE lhe dá "SQL Local", um stripped-down subconjunto da velha linguagem SQL-89 standard, lhe permite query local, não-SQL engines de DB como Paradox, DBase e Access, através de intérpretes de run-time imbutidos nos drivers "nativos". O Access aparece como uma máquina de SQL que não é. Se parece uma máquina de QBE, até mesmo em seu próprio ambiente nativo, com uma linguaguem bastante estranha e temperamental "SQL" . Ainda SQL Local é uma introdução boa a SQL, não bastará quando você se mudar para o Firebird.

Em particular, seu legado de aplicação Paradox ou de Access podem requerer atenção séria às declarações de SQL, para corrigir declarações inválida da linguaguem ou pobremente otimizar as queries. Por exemplo, note que o operador de concatentation de string no Firebird é o duplo-pipe (' || '), em conformidade com padrão, não o ' +' símbolo usado em Delphi, Paradox, Access e o Servidor MSSQL. Conseqüentemente, esta expressão devolve um erro no Firebird:

```
FirstName + ' ' + LastName As FullName
```

considerando que

FirstName || ' ' || LastName As FullName

está correto.

Documentação

A Sintaxe de SQL é bem documentada totalmente no Guia de Definição de Dados e a Referência de Linguagem, livros da documentação do Firebird. Infelizmente, os exemplos oferecidos são normalmente triviais e não útil para ajudar resolver seus problemas de query. Vários livros excelentes estão disponíveis, para ajudar a avançar suas habilidades de SQL. Esses que usam os padrões de SQL-92 são muito pertinentes a usuários de Firebird. Uma gama de recomendações podem ser achadas na página da livraria do website de www.ibobjects.com.

Q. Está Convertendo para o IBO o suficiente?

A conversão de sua aplicação de BDE existente para IB Objetos é simples. Dentro de minutos, sua aplicação é atualizado e executa livre da BDE! Mas é bastante para dar para seus usuários o que eles querem? Ou o que eles precisam?

Se sua aplicação velha usava TTables sob a BDE e a conversão usa uma versão de IB Objetos abaixo de 4, então o desempenho com a substituição de TIBOTable pode desapontar. Será especialmente notável quando a tabela for grande, pior onde o número de colunas for muitos. Encarecimentos estão entrando na v.4 que aperfeiçoará a emulação de IBO de TTable.

TTable não é a ótima aproximação se você estiver convertendo para o IBO, como parte de um exercício de crescimento (base cliente crescente, tabelas maiores). Pode estar na hora de planejar um gradual re-desenho do banco de dados e a interface de usuário. Uma troca para os componentes IBO nativos, levar vantagem de muitas otimizações implementadas neles.

Resumo dos Assuntos

Banco de dados

Aplicações cliente não entram fisicamente em contato com os dados armazenados. Constantemente deve-se ter em mente que aplicações de cliente trabalham com sets de dados, não com tabelas físicas. Estruturas armazenadas precisam ser altamente abstratas, eliminar redundância e prover a melhor possível flexibilidade para o design dos sets de produção. Liberte-se você e seus designes das constraints de estruturas de tabela que mapeiam os requerimentos das saídas.

A eficiência na extração de dados depende totalmente da boa modelagem de dados (abstração de chaves) e efetiva indexação. Índices redundantes matam o otimizador do Firebird.

Entender bem do SQL do Firebird e da linguagem procedural do Firebird é essencial.

Projetando e Codificando

Fuja do programa de interface de usuário do tipo "controle-dirigido" sob TTable e grids e reconsidere seu modelo em termos de usuário, tarefas e desempenho de rede. Quando possível, pense em "eventos de dados" em lugar de "controle de eventos". Reduza o "caça-e-escolha" e randomness para o mínimo absoluto. Projete com pequenos sets de dados (única-linha é perfeita!) e tarefas bem definida com seus objetivos.

Desenhos de Interfaces que fazem fácil e rápido para usuários alcançarem os objetivos da tarefa delas. Lista de seleção e procuras servidor-centric são a chave a aplicações de cliente remotas amigáveis, rápidas.

Delegue validação e dados-mastigados para o servidor através de triggers, stored procedures e udfs (funções definidas pelo usuário).

Explore o poder do IB Objetos! Como SQL é o único meio de comunicação entre sua aplicação de cliente e o servidor, use bem a linguagem e faça uso do componente TIB_Monitor ao longo de seu trabalho de desenvolvimento. Não escreva código de manipulador genérico a menos que você saiba com certeza que já não é suprido pelo IBO. Leve um tempo para estudar o arquivo de ajuda exemplos. Pode salvar meses de trabalho.

Como sempre, se você tem qualquer comentário ou perguntas sinta-se livre para contatar-me ou o servidor de lista pública dedicados ao IBO.

Jason Wharton
http://www.ibobjects.com
mailto:jwharton@ibobjects.com

Artigo Original: http://www.ibobjects.com **Jason Wharton** jwharton@ibobjects.com	
Tradução e adaptação: **Antonio Porfírio - (Tony)** antonioporfirio@ipdal.com.br tonydmaz@yahoo.com.br	**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.	