Artigo Java Magazine 30 - O Novo HSQ LDB

Aprenda a configurar bancos de dados HSQLDB em uma rede local ou a embutir o engine deste banco de dados em suas aplicações Java.

Esse artigo faz parte da revista Java Magazine edição 30. Clique aqui para ler todos os artigos desta edição

Java Livre

O Novo HSQLDB

Conheça a nova versão do mais popular banco de dados Java

Aprenda a configurar bancos de dados HSQLDB em uma rede local ou a embutir o engine deste banco de dados em suas aplicações Java

Fernando Lozano

O HSQLDB é um banco de dados criado inteiramente em Java, capaz de operar embutido em uma aplicação ou como um servidor de rede independente. Suporta um rico dialeto SQL, incluindo triggers, integridade referencial, outer joins, visões, transações, campos BLOB, schemas, roles e consultas correlatas. O dialeto SQL do HSQLDB é mais rico do que muitos SGBDs tidos como "mais poderosos", por exemplo o MySQL. Tudo dentro de um pacote jar contendo menos de 200 Kb (se compilado sem as ferramentas gráficas de administração e servidor web embutido).

A popularidade do HSQLDB é inegável se olharmos para os projetos que o incluem como padrão, que vão desde servidores de aplicação J2EE como o JBoss, até ferramentas de desenvolvimento desktop como o iReport. A versão 2.0 do OpenOffice, a popular suíte de escritórios que se firmou como alternativa ao Office da Microsoft, mesmo não sendo uma aplicação escrita em Java, inclui o HSQLDB como servidor de banco de dados embutido na aplicação *OOo Base* (que fornece funcionalidade similar ao Access da Microsoft).

O HSQLDB já foi apresentado antes nesta coluna, na Edição 7, além de ter sido usado em vários exemplos de aplicações (como nas séries recentes sobre o Tomcat 5 e o NetBeans), e em outras matérias da Java Magazine. No presente artigo são apresentados novos recursos da versão 1.8 e funcionalidades avançadas não vistas no primeiro artigo. São fornecidas também informações suficientes para quem nunca antes usou o HSQLDB decidir se ele pode ser ou não a solução para suas necessidades, e para criar as primeiras aplicações para esse banco de dados livre 100% Java.

O corpo principal deste artigo apresenta a arquitetura, características e recursos do HSQLDB. Depois são apresentadas ferramentas para administração do banco. O artigo finaliza com dicas para a programação de procedimentos armazenados. O **tutorial** "Primeiros passos com o HSQLDB" apresenta a instalação e o uso básico do banco de dados, incluindo a criação de uma aplicação cliente.

Terminologia e conceitos gerais

Antes de prosseguir com o HSQLDB, vamos definir a terminologia utilizada. Chamamos de **banco de dados** um conjunto de informações organizadas para um propósito específico. Conceitualmente, um arquivo *.txt* pode ser um banco de dados, por exemplo, se este arquivo for a relação de "favoritos" do seu navegador web, ou suas senhas de acesso a serviços on-line, como o de provedor internet ou um portal frequentemente visitado. Até mesmo um caderno onde se anotam telefones e endereços de e-mail dos amigos também é um banco de dados pode ser considerado um banco de dados.

Já um **sistema gerenciador de bancos de dados** (SGBD) é o software que gerencia o acesso a um ou mais bancos de dados, em vez de deixar outras aplicações o acessarem diretamente. O SGBD garante performance e integridade no acesso e na modificação aos dados, simplificando a escrita de aplicações. Sendo muito comum se fazer o acesso a dados por meio de SGBDs, o termo "banco de dados" é usado freqüentemente como sinônimo de SGBD.
Se o SGBD está embutido em uma aplicação, em vez de executar como um servidor de rede, costuma-se chamá-lo de **engine (motor ou mecanismo) de banco de dados**.

Arquitetura do HSQLDB
O HSQLDB foi criado para ser um banco de dados leve, com pouca demanda de uso de processador, memória e armazenamento. Ele é voltado para uso embarcado, seja internalizado a uma aplicação desktop, ou dentro de um hardware especialmente projetado. O HSQLDB foi, por exemplo, utilizado com sucesso no PDA Zaurus da Sharp e como parte do sistema de apuração eletrônica das eleições no Brasil (no software fornecido para a imprensa, candidatos e partidos acompanharem as apurações em tempo real).

O Projeto JVending (jvending.sf.net), uma aplicação de comércio P2P, fornece um porte do HSQLDB 1.8 para o profile CDC do J2ME. É necessário também o pacote opcional (para o J2ME) JDBC-CDC (jcp.org/en/jsr/detail?id=169). A versão customizada do HSQLDB pode ser baixada em sf.net/project/showfiles.php?group_id=10291 e o JDBC-CDC em java.sun.com/products/jdbc/download.html#cdcfp

O coração do HSQLDB é um engine SQL que opera inteiramente em memória, sem usar arquivos temporários em disco. Isto torna o HSQLDB capaz, por exemplo, de operar inteiramente a partir de um CD, ou em ambientes apenas com (pouca) memória Flash. Mas traz uma limitação: todos os registros que satisfizerem a uma consulta devem ser mantidos em memória. O mesmo ocorre com os índices das tabelas. Ou seja, embora o HSQLDB seja capaz de lidar com bancos de dados ocupando até 8 Gb em disco (utilizando o tipo de tabela *cached*, apresentado mais adiante), ele não será capaz de retornar todos os registros deste banco como resultado de uma única consulta.

*Uma forma de contornar esta limitação é usar a cláusula **LIMIT** do comando **SELECT**, para limitar a quantidade de registros retornados, e assim transformar o que seria uma única consulta em uma sucessão de consultas que retornam, cada uma, uma fração dos resultados desejados.*

O engine do HSQLDB roda como um único thread, de modo que realiza um comando SQL por vez. Se este comando for um procedimento armazenado, este será totalmente executado antes que outro comando SQL possa ser processado. Pode parecer uma limitação muito séria, mas ela permite ao engine utilizar pouca memória e para gerenciar locks e transações. De fato, o manual do HSQLDB afirma que 170 Kb de RAM são suficientes para rodar o engine (fora a quantidade de memória utilizada pelos **ResultSets** JDBC abertos pela aplicação e pela memória ocupada pela própria JVM).

O fato do engine utilizar um único thread não impede que o servidor HSQLDB aceite várias conexões de rede simultâneas, sendo cada uma servida pelo seu próprio thread. Também não impede que cada conexão esteja percorrendo um **ResultSet** diferente ao mesmo tempo, pois cada thread de conexão mantém a referência aos seus dados retornados. Mas várias conexões simultâneas aumentam a demanda de memória do servidor HSQLDB, e a execução de comandos SQL um-a-um pode comprometer o tempo de resposta quando houver uma grande quantidade de usuários simultâneos.
Por outro lado, com vários threads independentes gerenciando as conexões, as atividades que envolvem acesso a rede (envio de comandos SQL e devolução de resultados) são executadas em paralelo. Assim o HSQLDB é capaz de atender a algumas dezenas de usuários concorrentes sem que se perceba, na maioria dos casos, alguma degradação de performance.
Se uma aplicação (local ou servidora) utilizar vários bancos de dados HSQLDB simultaneamente, cada banco terá seu próprio thread de engine, de modo que usuários de um banco não competem com usuários de outras.

Benchmarks criados pelo projeto PolePosition (polepos.org) colocaram o HSQLDB de modo geral com uma performance superior a outros bancos de dados livres, entre eles o MySQL e o Apache Derby (apresentado na Edição 29). As páginas do projeto dizem que a maioria dos bancos proprietários tiveram nos testes um desempenho bastante inferior ao MySQL (e portanto também ao HSQLDB), mas questões de licenciamento impedem o projeto de divulgar os resultados para estes bancos.

Modos de operação

Para dar maior flexibilidade ao desenvolvedor, o HSQLDB pode operar em quatro modos de operação, que determinam como aplicações-clientes se comunicam com o engine SQL: Server, Standalone, Web Server e Servlet. Três desses modos de operação (Server, Web Server e Servlet) permitem o uso de conexões seguras no padrão SSL/TLS, utilizando o JCE (Java Cryptography Extensions).

Standalone

No modo Standalone, o HSQLDB roda na mesma JVM que a aplicação. Pode haver várias conexões simultâneas ao banco, desde que todas partam de outros threads na mesma JVM. Este modo é geralmente o preferido para aplicações embarcadas, por não exigir a abertura de portas TCP, nem impor o consumo de memória e processamento adicional gerado pelo envio dos comandos SQL e pela serialização dos resultados.
O engine SQL em si roda em um thread separado, que só será finalizado ao ser encerrada a JVM (chamando-se o método **System.exit()**) ou se for enviado o comando SQL **shutdown**, da mesma forma que se faria no modo Server.
Containers web ou EJB podem usar o modo Standalone em vez do modo server. Terão assim um ambiente mais leve e mais seguro, já que o banco de dados não estará exposto a conexões externas. Esta possibilidade é nova no 1.8, pois antes um banco HSQLDB no modo Standalone só permitia uma conexão por vez (veja o **quadro** "Novos recursos da versão 1.8").

Server

O modo Server é o preferencial para desenvolvimento, ou quando se usa o HSQLDB como servidor de banco de dados departamental. Nesse modo de operação, são aceitas conexões em uma porta TCP (por padrão a 9001), utilizando um protocolo de aplicação próprio do HSQLDB. Dessa maneira, vários clientes em JVMs diferentes podem acessar o mesmo banco de dados simultaneamente.
O servidor só será encerrado se ele receber o comando SQL **shutdown**, que deve ser enviado por uma conexão realizada por um usuário com permissões de administrador do banco (por padrão o usuário "sa").

Web Server

Para situações em que se deseja fornecer acesso remoto ao banco de dados, mas há um firewall no meio do caminho, existe o modo de operação Web Server. Neste modo o HSQLDB aceita conexões TCP/IP encapsulando comandos SQL, e retorna os resultados pela mesma conexão.
Um cenário para o modo Web Server é o uso de applets Java ou do Java Web Start (JWS). Nesses casos, as configurações do *sandbox[1]* da JVM podem impedir que sejam criadas conexões TCP/IP a outro que não o servidor web de origem da aplicação. Assim, rodar o próprio HSQLDB como servidor web contorna esta restrições.
No modo Web Server, o HSQLDB também é capaz de responder a requisições GET e HEAD para arquivos estáticos (imagens, arquivos HTML etc.). Dessa forma, ele mesmo poderia fornecer a página web que contém o applet ou a aplicação JWS para iniciar a aplicação.
Diferentemente de conexões HTTP padrão, o HSQLDB no modo Web Server mantém a conexão aberta para receber múltiplos comandos SQL. Isso possibilita a realização de transações (o **quadro** "Novos recursos da versão 1.8" mostra mais detalhes).
O comando **shutdown** continua sendo necessário para finalizar o HSQLDB neste modo.

Servlet

O modo Servlet atende a usuários de serviços de hospedagem compartilhada em sites web. Muitos destes serviços não fornecem acesso a um banco de dados, ou fornecem apenas uma opção padrão (a mais popular é o MySQL). O usuário pode fazer a instalação de pacotes war ou em alguns casos até pacotes ejb-jar e ear, mas não pode instalar novas aplicações no servidor do provedor, especialmente aplicações que escutem em portas TCP (pois isto pode comprometer a segurança do servidor e afetar outros usuários). Para contornar as limitações do serviço oferecido pelo provedor, o HSQLDB fornece um servlet que recebe comandos SQL como parte dos parâmetros da requisição HTTP e devolve os resultados como resposta à requisição.
O modo Servlet é praticamente igual ao Web Server – apenas o HSQLDB não responde diretamente a conexões HTTP, deixando que o container web (ou o servidor web que o contém) responda às requisições. Como no modo Web Server, ele utiliza conexões HTTP persistentes para permitir a realização de transações no banco de dados.
Da mesma forma que no modo Standalone, o engine é executado como um thread, em separado dos threads que executam o servlet do HSQLDB, e deve ser encerrado pelo comando **shutdown**.

Modo Standalone e servidores de aplicações

Foi dito, na descrição do modo Standalone, que esse modo pode ser adequado para aplicações web e EJB, mesmo para usuários de serviços de hospedagem compartilhada – se apenas a aplicação acessar o banco de dados. Entretanto, o modo Standalone não permite conexões remotas para realizar tarefas administrativas (como mudar as colunas de uma tabela) ou para depuração (verificar diretamente os dados armazenados em uma tabela, para conferir o resultado de uma consulta).

Por isso, alguns usuários preferem usar os modos Web Server ou Servlet – se o modo Server não foi possível por causa de firewalls ou restrições do provedor de hospedagem. Por outro lado, os modos de servidor deixam o banco exposto a conexões diretas e possíveis ataques de hackers, envolvendo captura ou adivinhação das senhas dos usuários. De modo geral, não é recomendado expor diretamente um servidor de banco de dados a conexões partindo de fora da rede local da empresa, embora o suporte a SSL no HSQLDB possa amenizar bastante os riscos.

Outros usuários usam um "console SQL" rodando como um servlet ou página JSP de uma aplicação web, que aceita comandos SQL genéricos e exibe os resultados da sua execução. Dessa forma, há a vantagem de performance do modo Standalone com a flexibilidade de realizar remotamente tarefas administrativas, sem necessidade de parar a aplicação web. Embora um console SQL como esse seja simples de escrever, deixá-lo disponível no provedor junto com a aplicação representa praticamente os mesmos riscos de segurança que deixar o próprio servidor de banco de dados aceitando conexões diretas.

Tipos de tabelas

O HSQLDB permite definir três tipos de tabelas. O tipo deve ser estabelecido no momento da criação da tabela, e determina a estrutura de armazenamento dos dados em memória e/ou em disco.

Em memória (Memory)

O tipo *Memory* é o padrão para a criação de tabelas. Com ele, todos os registros da tabela são mantidos em memória para acesso rápido. Entretanto, os dados são preservados permanentemente em disco na finalização do banco de dados, e também no log de transações, evitando a perda de dados em caso de falha no software ou hardware do servidor.

O uso de tabelas do tipo Memory pode levar a grandes demandas de memória na aplicação. Por outro lado, várias aplicações se beneficiam de ter seus dados inteiramente em memória, e é mais simples e confiável utilizar este recurso do HSQLDB do que usar mecanismos de cache com um banco de dados tradicional, pois o cache feito fora do banco de dados pode facilmente ficar desatualizado.

Em cache (Cached)

Tabelas *Cached* mantêm os registros acessados mais recentemente em memória, mas gravam todos o dados em disco de forma imediata. O tamanho do cache é configurado pelo administrador para a banco de dados como um todo, não por tabela ou por banco de dados, de modo que ele possa limitar o consumo de memória total.

Tabelas cached também ajudam a limitar o tamanho do arquivo *.script* do banco de dados e a duração de uma operação de **checkpoint** (mais sobre estes adiante).

Texto (Text)

Uma do tipo *Text* usa um arquivo texto comum (seguindo o padrão CSV) para armazenamento permanente dos seus registros, e a mesma área de memória para cache de registros que é utilizada para as tabelas cached. Tabelas de texto simplificam a troca de dados com fontes externas de dados, ao custo de uma pequena perda de performance.

A primeira linha do arquivo é um cabeçalho que fornece os nomes das colunas da tabela. Cada uma das linha restantes corresponde a um registro. O caractere separador de campos usado (a vírgula, como padrão), e o uso de aspas ou outro delimitador para strings podem ser definidos para cada tabela, com o comando SQL **set**, ou de forma global no arquivo de propriedades do banco de dados.

Definindo o tipo de uma tabela

O tipo da tabela é determinado no momento da sua criação. Entre as palavras-chave **create** e **table** deve ser indicado o tipo de tabela, e o comando SQL **create table** gera tabelas em memória. Dessa forma, os dois comandos a seguir são equivalentes:

create table <nome> (<colunas>...)

create memory table <nome> (<colunas>...)

A criação de uma tabela cached é feita desta maneira:

create cached table <nome> (<colunas>...)

Já a criação de uma tabela de texto é realizada em duas etapas: primeiro se cria a tabela, depois indica-se qual o arquivo CSV que contém os seus dados:

create text table <nome> (<colunas>...)

set table <nome> source '<caminho para o arquivo>'

Para especificar opções particulares para uma tabela de texto, inserem-se parâmetros separados por ponto-e-vírgula após o nome do arquivo. Por exemplo, para indicar que o separador de campos é um sinal de dois pontos:

set table <nome> source '<caminho para o arquivo>;fs=:'

O tipo de uma tabela não pode ser modificado depois da sua criação. É necessário uma exportação e uma re-importação dos dados. Mas o nome da tabela, suas colunas e os tipos de dados de cada coluna podem ser modificados a qualquer momento; também podem ser adicionadas e removidas colunas, utilizando comandos SQL **alter table**.

Tabelas temporárias

O HSQLDB suporta ainda tabelas temporárias, as quais são sempre armazenadas inteiramente em memória, e podem ser globais ou visíveis apenas pela mesma conexão em que foram criadas.

Para criar uma tabela temporária, utiliza-se o tipo de tabela “temp” (ou então “global temp”) em lugar de “cached” ou “memory” no comando SQL **create table**.

O comportamento padrão do HSQLDB é descartar os dados das tabelas temporárias ao fim da transação (seja com um **commit** ou um **rollback**), mas este comportamento pode ser modificado acrescentando-se a cláusula **on commit preserve rows** ao fim do comando SQL **create table**. De qualquer forma, as tabelas temporárias serão sempre descartadas no **shutdown** do engine do HSQLDB.

Índices

Índices no HSQLDB são sempre armazenados em memória, nunca em disco. Os arquivos do banco de dados registram apenas a definição do índice (comando **create index**), e o conteúdo dos índices é reconstruído a cada inicialização do engine. Portanto, definir uma grande quantidade de índices, como se faz normalmente com outros SGBDs, pode gerar um grande consumo de memória e aumentar o tempo de inicialização do HSQLDB.

Por outro lado, o HSQLDB, apesar do seu tamanho reduzido, possui um bom otimizador de consultas, que irá utilizar índices para acelerar joins, agrupamentos e seleções. Entretanto, os índices não serão utilizados para ordenar os resultados de uma consulta.

De modo geral, o impacto de criar índices adicionais será bem menor do que em um SGBD tradicional, especialmente quando são utilizadas tabelas em memória.

Arquivos do banco de dados

Quando se especifica um nome de arquivo para o HSQLDB, na verdade se está especificando o nome para um grupo de arquivos de vários tipos. Esses arquivos, organizados por sua extensão são mostrados a seguir.

.lck

Esses arquivos indicam que o banco de dados já foi aberto por um engine, e portanto não poderá ser aberto novamente por outra JVM. Em raras situações pode ser necessário remover este arquivo manualmente, após um término anormal do HSQLDB.

.properties

É o principal arquivo de um banco de dados HSQLDB e o único que irá existir sempre. Um arquivo texto, ele contém todos os comandos SQL necessários para recriar o banco de dados como estava no momento do último desligamento, exceto pelos dados em tabelas cached ou de texto.

Em algumas situações de migração ou compatibilização com novas versões, este arquivo pode ser editado diretamente, desde que não esteja em uso por uma aplicação.

.data

Dados binários das tabelas cached. Se existirem tabelas cached no banco de dados, todas compartilharão o mesmo arquivo.

Ocasionalmente poderá ser necessário compactar o arquivo *.data* para recuperar o espaço ocupado por registros removidos, ou que aumentaram de tamanho depois de comandos como **update** ou **alter table add column**. Como realizar essa compactação é mostrado na próxima seção.

.log

É o log de transações do banco de dados, em formato texto. Basicamente, contém todos os comandos SQL executados pelo banco desde que o banco foi aberto, ou desde o último **checkpoint**. Mais detalhes na próxima seção.

Pode parecer estranho usar um formato textual para o log de transações, mas é um formato tão bom em desempenho de entrada/saída e confiabilidade quanto os formatos binários utilizados por outros bancos de dados. Além disso, como o HSQLDB não fornece níveis de isolamento de transações, ele não tem necessidade de acessar os dados no log de transações para obter versões antigas de dados que são manipulados por uma transação[2].

.properties

É um arquivo de propriedades padrão do Java, onde podem ser armazenadas configurações específicas para o banco de dados.

O arquivo de propriedades é criado com valores default pelo HSQLDB e de modo geral pode ser editado diretamente pelo desenvolvedor. Apenas a propriedade **modified** tem um papel especial, e por isso não deve ser modificada, a não ser em casos extremos. Ela indica o estado atual do banco de dados: se está aberto (valor **yes**), com informações no arquivo *.log* que ainda não foram salvas nos arquivos *.script* e *.data*; se foi fechado (**shutdown**) corretamente (valor **no**), ou se o processo de **shutdown** estava em progresso e foi abortado inesperadamente (valor **yes-new-files**).

.backup

Este é um backup compactado do arquivo *.data* no momento do último **shutdown**. Ele existe para possibilitar a recuperação de um arquivo *.data* corrompido por alguma falha durante o próprio processo de finalização do HSQLDB.

.script.new e .backup.new

A presença destes arquivos indica que a última finalização do HSQLDB não foi completada. Felizmente, basta abrir novamente o banco de dados colocá-lo novamente em um estado consistente, desde que não tenha sido removido nem editado nenhum dos outros arquivos do banco de dados.

Em vez de sobrescrever diretamente os arquivos *.script* e *.backup*, a finalização gera novos arquivos com a extensão *.new*. Somente depois que estes arquivos tenham sido gerados em sua totalidade, os arquivos originais são removidos e então a extensão *.new* é removida dos nomes dos novos arquivos.

Banco de dados em um jar

A versão 1.8 traz uma nova facilidade que simplifica a cópia e o empacotamento de um banco de dados HSQLDB. É o acesso aos dados como um recurso, ou seja, de dentro de pacotes jar no classpath do engine (que é o mesmo da aplicação, no modo Standalone).

Um banco de dados num jar deve ser tratado como somente para leitura. Mesmo que não seja configurado como tal, o engine não irá atualizar os arquivos *.script*, *.data* e *.log*. Assim qualquer modificação feita pela aplicação será perdida no **shutdown**.

Bancos de dados voláteis

É possível configurar o HSQLDB para trabalhar com um *banco de dados volátil*, criado e mantido inteiramente em memória. Este banco nunca armazena nada em disco (nem mesmo os logs de transação) de modo que todo o seu conteúdo é perdido na finalização. Bancos voláteis, assim como bancos em jar, podem ser abertos em qualquer dos modos de operação do HSQLDB

A utilidade de um banco de dados volátil é atuar como um cache para outra fonte de dados. Por exemplo, imagine uma aplicação que exibe dados em forma de tabelas formatadas ou gráficos para o usuário, com várias opções diferentes de organização e ordenação dos dados e possibilitando a escolha de filtros e níveis de detalhes diferentes. Se cada operação exigir uma nova consulta a um banco de dados remoto, e o volume de dados for muito grande, o tempo de resposta e carga na rede poderão ser inaceitáveis. Mas se os dados são primeiro carregados para um banco do HSQLDB em memória, sendo as operações de filtragem e ordenação feitas sobre este banco local, é possível oferecer uma performance melhor sem onerar a rede e o servidor remoto.

*Note que poderia ser utilizada uma coleção Java (um **List** ou **Map**) como cache local em memória, mas estas coleções não têm a flexibilidade da linguagem SQL para especificar formas diferentes de organizar e agrupar dados.*

Como criar um banco de dados

O primeiro acesso a um banco de dados do HSQLDB cria o próprio banco se ele não existir, gerando os arquivos *.properties*, *.script* e *.log* vazios. Já o arquivo *.data* será criado apenas depois de criada uma tabela cached.

Entretanto, nem sempre esse será um comportamento desejado, pois um erro na digitação da URL de conexão JDBC irá provocar a criação de um novo banco de dados, em vez de um erro indicando que o banco requisitado não existe. Para garantir que só se abram bases de dados HSQLDB existentes, basta acrescentar ao final da URL de conexão a opção **ifexists=true**. Por exemplo, pode-se abrir o banco de dados do exemplo de "primeiros passos" deste artigo com o comando:

```
java org.hsqldb.util.DatabaseManagerSwing ⏎
    --url jdbc:hsqldb:file:/bd/teste;ifexits=true
```

Transações, checkpoints e compactação

O HSQLDB suporta transações basicamente da mesma forma que outros bancos de dados. Alterações realizadas sobre o banco têm que ser confirmadas com o comando SQL **commit**. Caso uma transação não seja confirmada, por falha na aplicação ou no próprio engine (como uma queda de energia) a transação é cancelada. As aplicações também podem cancelar transações explicitamente com o comando SQL **rollback**.

O modo auto-commit (ativado por padrão) pode ser configurado no banco de dados ou no driver JDBC para evitar a necessidade de enviar explicitamente comandos **commit**. Isso significa que cada comando SQL enviado é considerado como sendo uma transação por si só, sendo esta transação confirmada imediatamente ao final da execução do comando.

Algumas aplicações, que dependem do recurso de isolamento de transações para funcionar corretamente, podem contornar esta limitação do HSQLDB pelo uso de procedimentos armazenados. O engine do HSQLDB garante que um procedimento seja executado até o final, antes que comandos SQL vindos de outras conexões sejam executados. Dessa forma, uma transação realizada dentro de um procedimento armazenado nunca "verá" modificações realizadas por outras transações depois do seu início.

O HSQLDB registra todos os comandos SQL que modificam o banco de dados nos arquivos *.script* e *.log*, que poderão crescer bastante ao longo do tempo. Para que se possa limitar este crescimento, o HSQLDB fornece o comando **checkpoint**. Este realiza o mesmo procedimento ocorrido na finalização: a geração de um novo arquivo *.script* e a atualização do arquivo *.data* – com a diferença de que não fecha conexões ou encerra o engine. O arquivo *.script* fica menor porque ele passa a conter apenas comandos **create** e **insert**, em vez de conter também comandos **alter**, **update** e **delete**. Já o arquivo *.log* estará vazio após um comando **checkpoint**, a não ser que ainda haja alguma transação executando em outra conexão.
O engine poderá gastar alguns segundos (ou mesmo alguns minutos, para um banco de dados muito grande) durante a execução do comando **checkpoint**. Nesse período, qualquer outra conexão ficará aguardando o término da operação.
Será útil também usar o comando SQL **checkpoint defrag** em uma aplicação que execute por um longo tempo sem interrupções. Este comando, além de efetuar as tarefas do comando **checkpoint**, realiza a *compactação* (ou desfragmentação) do arquivo *.data*.
A fragmentação ocorre porque o HSQLDB limita a quantidade de memória utilizada para relacionar as áreas livres dentro do arquivo *.data*, causadas por remoção de registros ou expansões no seu tamanho. Então um alto volume de modificações pode fazer com que áreas livres deixem de ser utilizadas pelo banco de dados. Compactar o arquivo *.data* elimina estas áreas livres não-utilizadas.
Outra forma de compactar o arquivo *.data* é fechar o engine com o comando **shutdown compact**. É um pouco inconsistente, mas a compactação é mesmo feita pela cláusula **defrag** no comando **checkpoint** e pela cláusula **compact** no comando **shutdown**.
Operações de migração e atualização de bancos de dados HSQLDB poderão utilizar ainda o comando **shutdown script**. Esta opção salva no arquivo *.script* todos os registros de tabelas cached, eliminando assim o arquivo *.data*. Com isso, o arquivo *.script* passa a conter um script SQL completo para a criação das tabelas, índices e constraints, além da inserção dos dados nas tabelas.

Conectando ao HSQLDB

A conexão ao HSQLDB é feita por meio do seu driver JDBC. Não existe uma "API nativa" de acesso ao servidor.
A URL de conexão inicia sempre por "jdbc:hsqldb:". Em seguida vem um subprotocolo "hsql:", "file:", "res:", "http:" e "mem:", que indicam, respectivamente, a conexão a um banco de dados com o engine no modo Servidor; Standalone; Standalone com banco somente leitura num jar; Web Server ou Servlet; e banco em memória.
Um cliente remoto também pode utilizar os subprotocolos "hsqls:" e "https:", que fazem a conexão no modo Server, Web Server ou Servlet, utilizando uma conexão segura, criptografada, via JCE.
Quando se inicia um servidor HSQLDB, a opção **--database.<n>** (veja a seguir) também pode indicar os mesmos subprotocolos (mas só faz sentido utilizar as opções "file:", "res:" e "mem:").

Múltiplos bancos de dados por servidor

Um mesmo servidor HSQLDB pode fornecer acesso a até dez bancos de dados simultaneamente. Os bancos são numerados de zero a nove – daí o **<n>** na opção **--database.<n>**. A URL de conexão ao servidor faz referência sempre ao banco de dados por meio de um *alias* (nome alternativo). O alias deve ser especificado na inicialização do servidor pela opção **--dname<n>**.
Um alias vazio é considerado válido, e será usado caso não seja especificada a opção **--dname<n>**. Mas só pode ser definido um alias para cada banco de dados.
Assim sendo, a linha de comando a seguir inicializa um servidor HSQLDB que fornece o acesso a dois bancos de dados distintos:

```
java org.hsqldb.Server --database.0 /bd/teste ↵
      --dbname.0 teste --database.1 /dados/logs --dbname.1 logs
```

Um cliente poderia se conectar ao primeiro banco de dados utilizando a URL de conexão a seguir:

```
jdbc:hsqldb:hsql://127.0.0.1/teste
```

E para conectar ao segundo, será necessário especificar o alias correto:

```
jdbc:hsqldb:hsql://127.0.0.1/logs
```

Para simplificar a configuração de um servidor que fornece acesso a vários bancos de dados, pode ser criado um arquivo chamado *server.properties*, que será lido do diretório corrente de execução do servidor. O arquivo de propriedades a seguir define os mesmos dois bancos de dados:

```
server.database.0=/bd/teste
server.dbname.0=teste
server.database.1=/dados/logs
server.dbname.1=logs
```

E, para iniciar o servidor, bastará usar a linha de comando a seguir:

```
java org.hsqldb.Server
```

Ferramentas de administração

O pacote *hsqldb.jar* padrão inclui uma série de utilitários para administração do HSQLDB, mas que podem também ser utilizados para realizar tarefas similares em outros servidores de bancos de dados, desde que seja especificado o driver JDBC apropriado. Todos esses utilitários são classes no pacote **org.hsqldb.util**. Para executá-los basta incluir o *hsqldb.jar* no classpath do sistema e em seguida usar o comando:
java org.hsqldb.util.NomeDoUtilitario

A maioria dos utilitários reconhece a opção de linha de comando **--help** para informar a sua sintaxe, ou então pode ser utilizada interativamente, solicitando-se os parâmetros necessários.

No tutorial é apresentado um desses utilitários, o Database Manager. Na verdade existem duas versões do Database Manager: uma utiliza o AWT (**DatabaseManager**), para que possa ser executada em JVMs que fornecem apenas os recursos previstos no Personal Java (e que já funcionam com várias JVMs livres como Kaffe e JamVM). Outra é baseada no Swing (**DatabaseManagerSwing**), que exige portanto uma implementação completa do J2SE.

Outra ferramenta bastante útil é o *SqlTool*, um console SQL criado para operar interativamente em modo texto, ou por meio de scripts (arquivos *.sh* e *.bat*). O próprio SqlTool pode ser utilizado como um interpretador de scripts, oferecendo variáveis, lógica condicional e loops. É suficiente para codificar operações complexas de extração, migração e conversão de dados.

O SqlTool suporta comandos interativos para obter informações sobre objetos do banco de dados, como tabelas, índices e triggers. É seguido o mesmo estilo de comandos prefixados por uma contra-barra ("\") adotado pelo utilitário *isql* do Ingres e Sybase, ou o *psql* do PostgreSQL.

O SqlTool exige o arquivo de configuração *sqltool.rc* no diretório pessoal do usuário (a pasta *$HOME* no Linux ou *C:\Documents and Settings\<Usuario>* no Windows). Para acesso ao banco de dados de testes do tutorial, o conteúdo do arquivo seria:

```
urlid teste
url jdbc:hsqldb:file:/bd/teste
username sa
password
```

Aqui está um exemplo de uso:

```
$ java org.hsqldb.util.SqlTool teste
... mensagem de boas-vindas omitida ...
sql> \dt
TABLE_SCHEM  TABLE_NAME
-----------  ----------
PUBLIC       CONTATO
sql> \d contato
name   datatype  width  no-nulls
-----  --------  -----  --------
ID     INTEGER      11  *
NOME   VARCHAR      40  *
EMAIL  VARCHAR      30  *
sql> select * from contato;
```

```
ID  NOME                    EMAIL
--  ----------------------  -------------------------
 0  Fernando Lozano          fernando@lozano.eti.br
 1  Osvaldo Pinali Doederlein  osvaldo@visionnaire.com.br
 2  Java Magazine            info@javamagazine.com.br
3 rows
sql> \q
```

O símbolo "$" representa o prompt do sistema operacional e, claro, não deve ser digitado. Os comandos a serem digitados são destacados em negrito, e o restante são as saídas do SqlTool.

Observe que, no SqlTool, comandos SQL só serão executados quando terminados por um sinal de ponto-e-vírgula. O comando **\q** encerra o SqlTool.

A maior utilidade do SqlTool está na execução de scripts contendo comandos SQL para criar tabelas em um banco de dados e preenchê-las com dados iniciais. Digamos que os comandos SQL necessários para inicializar o banco de dados de teste estejam num arquivo chamado *teste.sql* (**Listagem 2**). O script seria executado com a linha de comando a seguir:

```
java org.hsqldb.util.SqlTool teste teste.sql
```

Observe que o SqlTool, ao contrário do Database Manager, executa por padrão com o auto-commit desligado, por isso é necessário executar o comando SQL **commit** ao fim do script. Além disso, o leitor deve se lembrar de fazer a finalização do HSQLDB se ele for acessado no modo Standalone.

Uma versão mais simples do SqlTool é fornecida com o nome de ScriptTool. Esta versão é mantida para compatibilidade com versões anteriores do HSQLDB e não fornece comandos internos para scripts como **\dt**, nem loops.

Funções e procedimentos armazenados em Java

Para o suporte a funções definidas pelo usuário (conhecidas em alguns servidore de BD como "procedimentos armazenados" ou "stored procedures") o HSQLDB utiliza a própria linguagem Java.

Funções definidas pelo usuário são métodos estáticos e públicos em qualquer classe presente no classpath de execução do HSQLDB (ou da aplicação, com o HSQLDB em modo Standalone). É possível chamar diretamente estas funções pelo comando SQL **call**, por exemplo:

```
call "java.lang.Math.max" (3, 4) ;
```

(Observe os sinais de ponto-e-vírgula, necessários para a execução via SqlTool; eles podem ser removidos se a execução for feita pelo Database Manager).

Se a função retornar dados, ela também poderá ser usada em comando SQL onde seja possível inserir expressões, como em comandos **select** e **update**. Também é possível simplificar a sintaxe de chamada a uma função utilizando o comando SQL **create alias**:

```
create alias maximo for "java.lang.Math.max" ;
call maximo (9, 3) ;
```

Caso se deseje que uma função execute comandos SQL (ex.: para atualizar múltiplos registros, ou para colocar a lógica de validação dentro do banco de dados) basta que a função receba como primeiro argumento um parâmetro do tipo **Connection** do JDBC. Neste caso, a função tem que ser chamada obrigatoriamente pelo comando **call**.

Retornando múltiplos registros usando coleções Java

Uma função do HSQLDB retorna sempre um **ResultSet** contendo uma única linha e uma única coluna. Não é possível criar procedimentos armazenados que retornem o resultado de consultas SQL. Mas é possível "simular" este recurso retornando um array ou coleção Java

A **Listagem 3** define a classe **ProcedimentoArmazenado**, que fornece três exemplos de funções do HSQLDB. A primeira (**maiusculas()**) demonstra como modificar registros, utilizando o parâmetro **Connection**. A segunda (**terminaEm()**) é uma função simples para testar o final de uma string; a terceira (**dominioEmail()**) usa a segunda função para retornar apenas os registros cujo endereço de e-mail esteja no domínio indicado como argumento.

Os aliases correspondentes podem ser criados com os comandos:

```
create alias maiusculas for "ProcedimentoArmazenado.maiusculas";
create alias terminaem for "ProcedimentoArmazenado.terminaEm";
create alias dominioemail for "ProcedimentoArmazenado.dominioEmail";
```

As duas primeiras funções podem ser chamadas diretamente, conforme o exemplo a seguir:

```
sql> call maiusculas('contato', 'nome');
  3
sql> select * from contato;
ID  NOME                    EMAIL
--  ----------------------  -------------------------
 0  FERNANDO LOZANO           fernando@lozano.eti.br
 1  OSVALDO PINALI DOEDERLEIN  osvaldo@visionnaire.com.br
 2  JAVA MAGAZINE             info@javamagazine.com.br
3 rows
sql> call terminaem ('lozano.eti.br', '.br');
true
sql> call terminaem ('lozano.eti.br', '.com');
false
```

Já a terceira não poderá ser chamada diretamente pelo Database Manager nem pelo SqlTool, pois essas ferramentas não saberão como representar um **java.lang.Object** genérico. Mas ainda assim poderá ser utilizada por uma aplicação Java. Um exemplo de aplicação é apresentado na **Listagem 4**.

Observe que a lista de arrays (com um array por registro) foi recuperada utilizando o método **getObject()** definido na interface **ResultSet** do JDBC.

Retornando múltiplos registros usando tabelas temporárias

Outra forma de simular um procedimento armazenado que retorna dados é criar uma tabela temporária, cujo nome pode ser inclusive fornecido pelos argumentos da função. Esta técnica é demonstrada na **Listagem 5**, criando uma variação da função **dominioEmail()** apresentada anteriormente. Para testar esta função, execute os seguintes comandos no SqlTool:

```
sql> create alias dominioemailtemp for ↵
    "ProcedimentoArmazenadoTemp.dominioEmailTemp";
sql> create alias dominioemailtemp2 for ↵
    "ProcedimentoArmazenadoTemp.dominioEmailTemp2";
sql> call dominioemailtemp('.com.br', 't');
  3
sql> select * from t;
ID  NOME                    EMAIL
--  ----------------------  -------------------------
 1  Osvaldo Pinali Doederlein  osvaldo@visionnaire.com.br
 2  Java Magazine           info@javamagazine.com.br
3 rows
```

Uma vantagem de se criar uma tabela temporária em vez de retornar coleções Java é a possibilidade de conferir os resultados pelos utilitários de administração do HSQLDB. Não é necessário remover as tabelas temporárias, pois elas serão descartadas ao fim da transação corrente. Como desvantagem, temos a necessidade de enviar comandos SQL adicionais para ler os resultados da tabela temporária.

A segunda variação da função (**dominioEmailtemp2()**) serve apenas para lembrar que muitas vezes é possível usar o comando SQL **select into** para gerar a tabela temporária. É uma possibilidade que às vezes é negligenciada pelos desenvolvedores. Mas haverá também situações em que a função terá que criar a tabela temporária e inserir os valores nela manualmente.

Conclusões

O HSQLDB é um banco surpreendentemente rico e poderoso. Embora não seja um substituto para bancos "peso-pesado" como Oracle e PostgreSQL, será uma alternativa satisfatória para uma grande quantidade de situações, incluindo armazenar o conteúdo de aplicações J2EE como portais.

Para uso embutido em aplicações, bancos de dados em memória ou em jar permitem coisas que impossíveis com outros bancos de dados, por exemplo rodar a aplicação inteiramente dentro de um applet Java ou diretamente a partir de um CD.

Novos recursos da versão 1.8

A versão 1.8 do HSQLDB é um marco em qualidade e quantidade de novos recursos. Entre as muitas novas funcionalidades da versão 1.8 em relação à 1.7, temos:

- Comandos SQL para modificar a estrutura de tabelas, como adicionar e remover colunas, mudar o tipo de dados de uma coluna ou seus constraints. Este é principalmente o resultado do trabalho conjunto com o projeto OpenOffice.org, que demandava estas capacidades para um banco de dados desktop.

- Novos comandos SQL como **case .. when**.
- Suporte a schemas, permitindo separar tabelas de aplicações diferentes em uma mesmo banco de dados, de modo que não haja conflito de nomes.
- Suporte a grants e roles, permitindo definir regras sofisticadas de controle de acesso.
- Novos tipos de joins. Agora são suportados **right outer join** e **cross join**, além do **left outer join** suportado em versões anteriores.
- Consultas correlatas envolvendo operações de conjuntos, como **union**.
- Implementação de catálogo padrão ANSI, no schema **information_schema**. (O catálogo é um conjunto de tabelas que descreve as tabelas, índices e funções existentes no próprio banco de dados.)
- Suporte a tabelas temporárias globais.
- **DatabaseManagerSwing**, que estava com vários bugs na série 1.7, obrigando o uso da versão baseada em AWT, voltou a ser utilizável.
- O **DatabaseManager** e **DatabaseManagerSwing** agora salvam configurações de conexões utilizadas previamente, evitando a redigitação de parâmetros.
- **SqlTool**, nova ferramenta genérica para execução de scripts de backup, migração, conversão e inicialização de bancos de dados.
- Um único engine HSQLDB agora pode gerenciar vários bancos de dados simultaneamente, seja nos modos Standalone ou nos modos de servidor.
- Mesmo no modos Standalone e em memória, é possível ter várias conexões ao mesmo banco de dados (desde que todas as conexões partam da mesma JVM).
- Aderência mais estrita aos padrões ANSI SQL, configurável por parâmetros do banco de dados.
- Comando SQL **set** para modificar parâmetros do engine e do banco de dados sem necessidade de modificar os arquivos *.properties* e reiniciar o engine.
- O engine agora é mais resistente contra a perda de dados em caso de término abrupto.
- Suporte a configurações de ordenação (**collation**) específicas para cada banco de dados.

Links

hsqldb.org

Página principal do HSQLDB

java.sun.com/products/jdbc

Sobre a API JDBC

java.sun.com/products/jdbc/download.html#cdcfp

JDBC para J2ME/CDC

jvending.sf.net

sf.net/project/showfiles.php?group_id=10291

Versão do HSQLDB que funciona na configuração CDC do J2ME

polepos.org

Benchmark para acesso a bancos de dados em Java, que cruza bancos de dados, mecanismos de persistência e drivers JDBC para indcar as melhores combinações

Tutorial: Primeiros passos com o HSQLDB

Para os que gostam de partir logo para a prática ou que ainda não conhecem o HSQLDB, este tutorial apresenta a instalação o HSQLDB, e como acessar um banco de dados usando o console SQL fornecido. Será mostrado também como fazer o acesso a partir de uma aplicação Java. (Este tutorial pressupõe pouco conhecimento de SQL e bancos de dados em geral).

Baixando e instalando o HSQLDB

Para obter o HSQLDB, visite o site do projeto em *hsqldb.org* e siga o link "download", que irá redirecionar o usuário à página de arquivos do HSQLDB no portal Sourceforge. Escolha então o arquivo *hsqldb_1_8_0_2.zip* (ou uma versão estável mais recente) e faça o download.

Descompacte o arquivo em uma pasta qualquer, criando a estrutura de diretórios apresentada na **Figura 1**. O pacote de download inclui os fontes (pasta *src*), documentação (*doc*) e aplicações de exemplo (*demo*). Mas o que realmente interessa no momento é o pacote *hsqldb.jar* na pasta *lib*. Este jar, de menos de 630 Kb, contém o servidor do HSQLDB e aplicações gráficas e de linha de comando para administração, bem como o driver JDBC para clientes remotos. É possível gerar novos pacotes contendo, por exemplo, apenas o servidor, ou apenas o driver JDBC – mas o conjunto todo é tão pequeno que a maioria dos desenvolvedores usa este pacote "tudo em um".

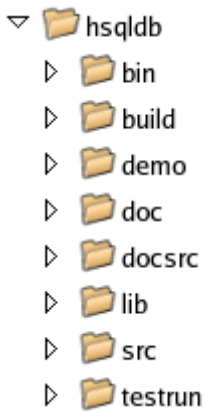

Figura 1. Estrutura de diretórios da instalação do HSQLDB

Considerando que o leitor já tenha o Java SDK 1.4.2 ou 5.0 instalado e configurado para uso direto pela linha de comando, o roteiro a seguir permite que se experimente um pouco com o banco de dados.

Supondo que o HSQLDB foi descompactado para a pasta */java/hsqldb* no Linux, basta configurar o classpath do sistema operacional com o comando a seguir:

```
export CLASSPATH=$CLASSPATH:/java/hsqldb/lib/hsqldb.jar
```

De forma similar, se a distribuição do HSQLDB tiver sido descompactada na pasta *C:\Java\Hsqldb* no Windows, o comando equivalente será:

set CLASSPATH=%CLASSPATH%;C:\Java\Hsqldb\lib\hsqldb.jar

Criando e abrindo um banco de dados

Agora é necessário escolher uma pasta para a criação dos arquivos do banco de testes. Para seguir o artigo sem mudanças, crie a pasta *bd* na raiz do seu drive C: no Windows, ou na raiz do sistema de arquivos no Linux. Desta forma podemos usar o caminho */bd* nas aplicações Java, independentemente do sistema operacional (desde que o drive corrente no prompt do MS-DOS seja sempre o *C:*).
O comando a seguir abre um utilitário baseado no Swing para administração do HSQLDB, que permite a digitação de comandos SQL e a exploração da estrutura de tabelas do banco de dados:

java org.hsqldb.util.DatabaseManagerSwing

O utilitário inicia apresentando uma janela de conexão, que deve ser preenchida conforme a **Figura 2**. Observe a escolha da opção "HSQL Database Engine Standalone" no combobox *Type*. O Database Manager fornecido com o HSQLDB não é em nada específico para ele, pois utiliza apenas a API padrão JDBC. Por isso pode ser utilizado da mesma forma com qualquer outro servidor banco de dados, bastando fornecer as informações de conexão corretas.

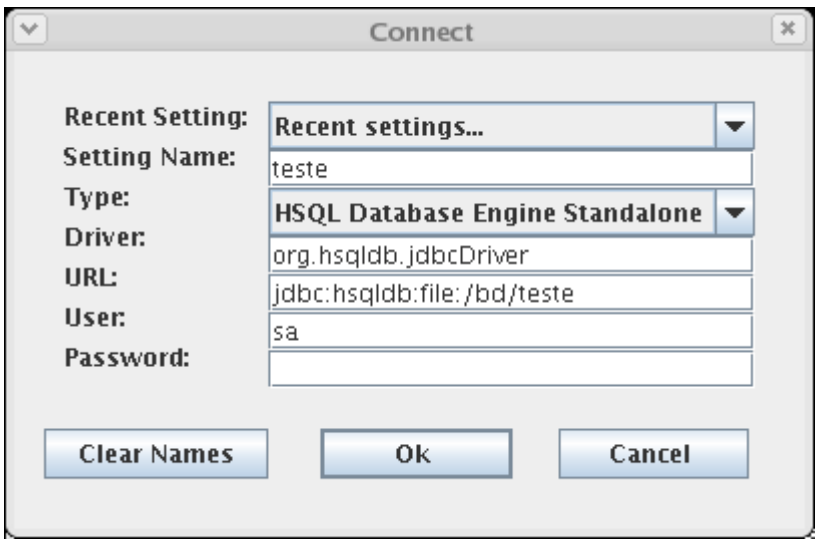

Figura 2. Parâmetros para conexão ao banco de dados do exemplo

O resultado da conexão será a criação de um banco de dados vazio. O leitor poderá confirmar que a pasta */bd* agora contém arquivos como *teste.lck*, *teste.properties* e *teste.log*, criados de acordo com o nome do banco de dados passada para a conexão.
Crie agora uma tabela para experimentação, digitando o comando SQL a seguir na área de texto do Database Manager:

```
create table contato (
    id identity primary key,
    nome varchar(40) not null,
    email varchar(30) not null
)
```

Depois de digitar o comando SQL exatamente como indicado, clique no botão *Execute SQL*. O resultado será como na **Figura 3**.

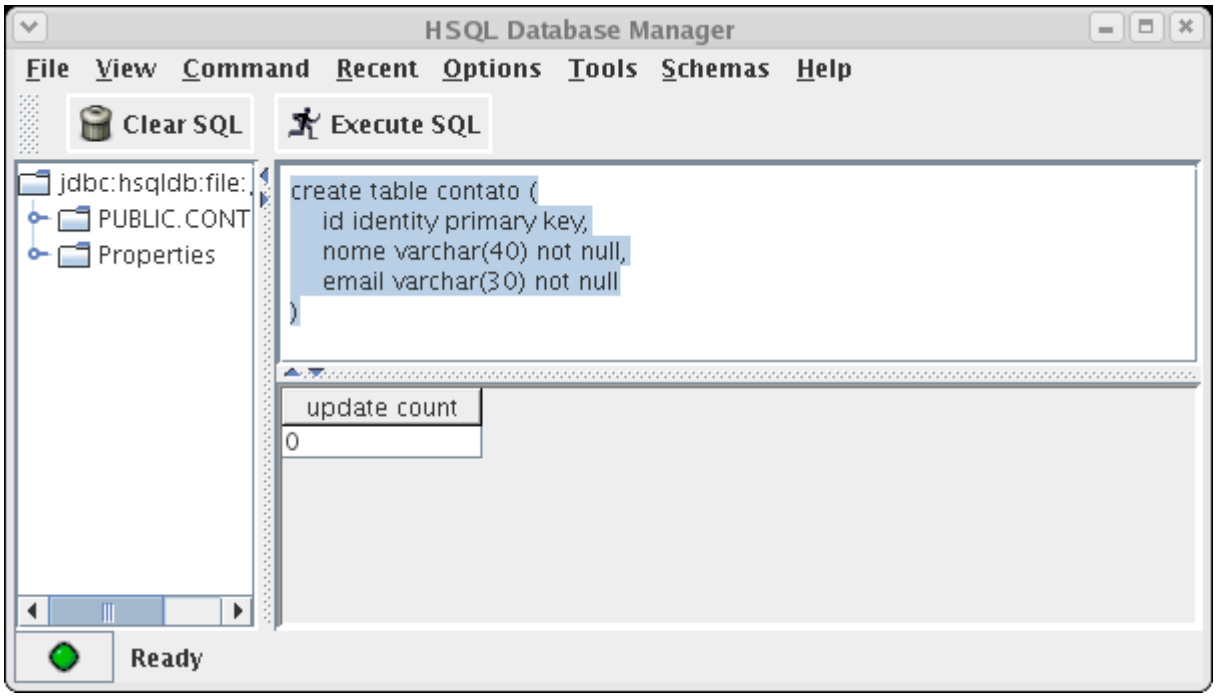

Figura 3. Após a criação de uma tabela

Um esclarecimento sobre o código SQL. O tipo de coluna **identity** é um inteiro auto-incrementado, semelhante aos fornecidos pelo MS Access e MySQL. O HSQLDB também fornece como opção o uso de seqüências (*sequences*) que funcionam do mesmo modo que no Oracle e PostgreSQL.

O resultado esperado é realmente "update count" igual a zero, pois o comando mostrado anteriormente não afeta nenhum registro (ao contrário de comandos como **insert**, **update** e **delete**). Caso haja algum erro de sintaxe na digitação do comando, o erro será exibido na tabela abaixo da caixa de texto, no mesmo local da mensagem "update count".

Observe que a parte esquerda da tela foi atualizada, sendo inserido na árvore um novo nó com o nome *PUBLIC.CONTATO*. "PUBLIC" é o nome do *schema* padrão do HSQLDB, sendo "CONTATO", claro, a tabela recém-criada. É possível criar schemas adicionais para agrupar tabelas, da forma semelhante ao agrupamento de classes Java em pacotes (declarações **package** e **import**), entretanto não é possível criar schemas aninhados.

Experimente expandir o nó da tabela *Contato* para visualizar sua definição. A **Figura 4** ilustra o que o leitor deverá ver. Observe que há um nó *Indices* na árvore: é comum que bancos de dados relacionais criem automaticamente um índice correspondente à chave primária (**primary key**) de uma tabela.

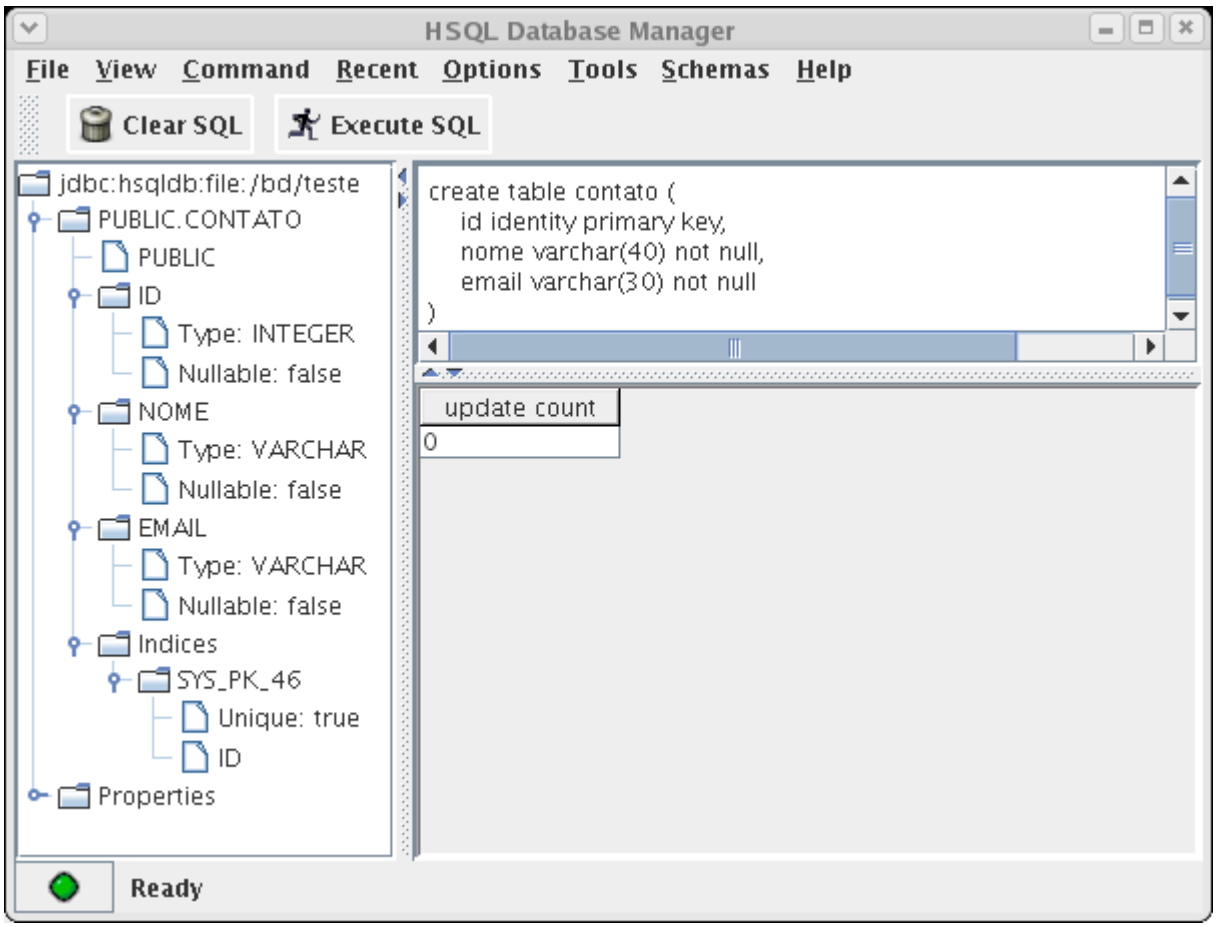

Figura 4. Expandindo na parte esquerda a estrutura da tabela recém-criada

Inserindo e consultado registros no banco de dados

Para inserir dados de teste na tabela, execute comandos SQL como os a seguir, um a um. A cada execução, deverá ser exibido como resultado um "update count" com o valor 1.

```
insert into contato (nome, email) values (
    'Fernando Lozano', 'fernando@lozano.eti.br')

insert into contato (nome, email) values (
    'Osvaldo Pinali Doederlein', 'osvaldo@visionnaire.com.br')

insert into contato (nome, email) values (
    'Java Magazine', 'info@javamagazine.com.br')
```

Em seguida, consultamos os dados da tabela:

```
select * from contato
```

O resultado esperado é apresentado na **Figura 5**.

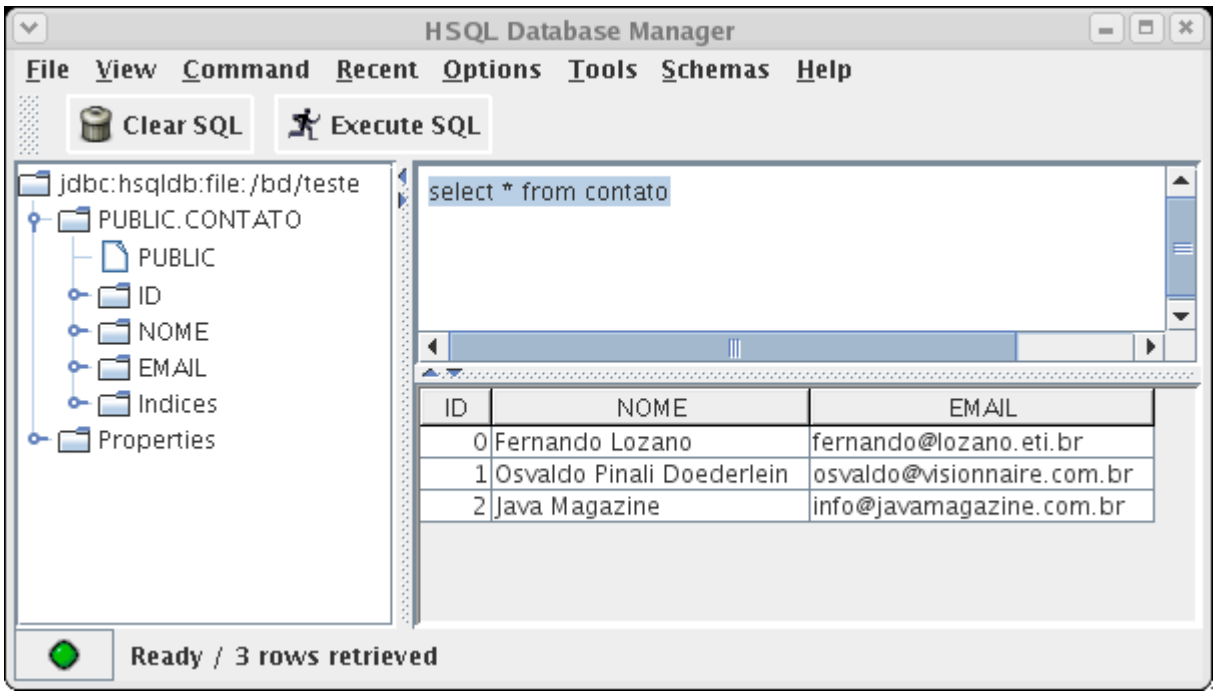

Figura 5. Listando todos os registros da tabela Contato

Fechando o banco de dados

Antes de fechar o utilitário, é necessário finalizar o engine do HSQLDB. Isso fecha o banco de dados e garante a sua consistência. Envie o comando SQL **shutdown** da mesma forma que foram enviados os comandos **create table**, **insert** e **select** nos exemplos anteriores. Observe que, daí em diante, qualquer tentativa de executar novos comandos SQL irá gerar uma exceção (veja a **Figura 6**).

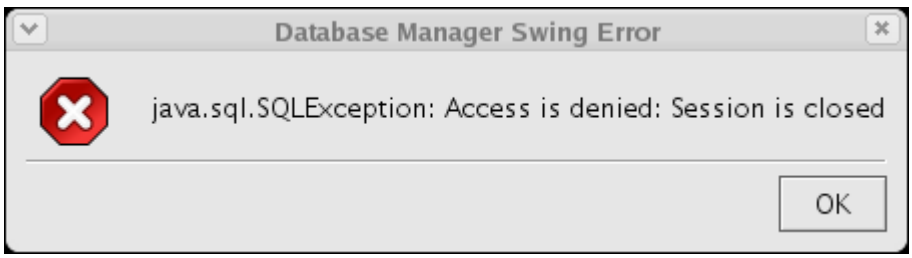

Figura 6. Exceção gerada ao tentar acessar uma banco de dados após um *shutdown*

Depois de encerrado o utilitário Database Manager, a pasta */bd* deverá conter apenas dois arquivos: *teste.properties* e *teste.script*. Se além desses existir o arquivo *teste.log* (e talvez *teste.lck*), isso significa que o engine do HSQLDB foi finalizado "à força", sem antes realizar a finalização do engine com **shutdown**.
Como não iniciamos um servidor HSQLDB em separado, o engine do banco de dados estava rodando na mesma JVM que o utilitário Database Manager. O encerramento do utilitário chama **System.exit(0)**, finalizando a JVM e assim "matando" o engine do HSQLDB, que não tem a oportunidade de fechar o banco de dados.
Felizmente o HSQLDB será capaz de se recuperar desta falha na próxima vez que o banco de dados for aberto. Mesmo assim não se deve deixar de realizar a finalização da forma correta. Em caso de esquecimento, abra novamente o banco de dados com o Database Manager e execute o comando SQL **shutdown**.

O roteiro que acabamos de realizar criou um banco de dados, que foi acessada pelo engine do HSQLDB no modo *Standalone*, utilizando tabelas em memória. Estes e outros conceitos do HSQLDB serão apresentados mais adiante neste artigo.

Dicas diversas

O Database Manager fornece uma série de opções de linha de comando que podem facilitar o seu uso, Para ver todas as opções reconhecidas, use a opção **--help** da linha de comando (ou **-?**). Por exemplo:

```
java org.hsqldb.util.DatabaseManagerSwing --help
```

As opções **--help** e **-?** têm efeito análogo para outros utilitários inclusos no HSQLDB.
Para abrir o banco de dados recém-criado, sem passar pelo diálogo de conexão, utilize:

```
java org.hsqldb.util.DatabaseManagerSwing ⏎
    --url jdbc:hsqldb:file:/bd/teste
```

Note que há espaços em branco antes e depois da opção **--url**. É possível ainda passar opções ao engine do HSQLDB inserindo-as ao final da URL de conexão, separadas por ponto-e-vírgula. Por exemplo, a opção **shutdown=true** faz com que o engine seja finalizado quando a última (ou a única) conexão for fechada, da mesma forma que seria feito pelo envio do comando SQL **shutdown**. O comando então ficaria:

```
java org.hsqldb.util.DatabaseManagerSwing ⏎
     --url jdbc:hsqldb:file:/bd/teste;shutdown=true
```

Usuários de Linux (e outros sistemas cuja linha de comando seja baseada no shell do Unix, como o Mac OS X) devem inserir uma URL de conexão, com parâmetros, toda entre aspas. Caso contrário o sinal de ponto-e-vírgula será interpretado como o separador de comandos. Então em sistemas Unix o comando anterior ficaria:

```
java org.hsqldb.util.DatabaseManagerSwing ⏎
    --url "jdbc:hsqldb:file:/bd/teste;shutdown=true"
```

Iniciando e conectando a um servidor HSQLDB

Em vez de iniciar o engine do HSQLDB dentro do utilitário de administração (ou de outra aplicação qualquer, usando uma URL de conexão contendo o subprotocolo "file:") é possível rodar um servidor HSQLDB independente, que aceita conexões de outros computadores em rede. Os arquivos do banco de dados são os mesmos, com ou sem servidor. Dessa forma, o comando a seguir inicializa um servidor que permite acesso remoto ao banco de dados de teste criado anteriormente:

```
java org.hsqldb.Server --database /bd/teste
```

Em outro terminal ou janela de prompt no Windows, com o ambiente devidamente configurado para o Java 2 SDK e o classpath também configurado para incluir o arquivo *hsqldb.jar*, execute agora um Database Manager que conecta ao servidor HSQLDB:

```
java org.hsqldb.util.DatabaseManagerSwing ⏎
    --url jdbc:hsqldb:hsql://127.0.0.1
```

A URL de conexão é diferente – especificando o subprotocolo "hsql:" e indicando o endereço IP do servidor; no caso o endereço 127.0.0.1 é o próprio computador – mas o resultado deverá ser igual ao retornado em conexões anteriores.
Caso o leitor prefira usar o diálogo de conexão em vez de especificar a URL na linha de comando, siga o modelo apresentado na **Figura 7**. Observe que dessa vez foi escolhida a opção "HSQL Database Engine Server" no combobox *Type*.

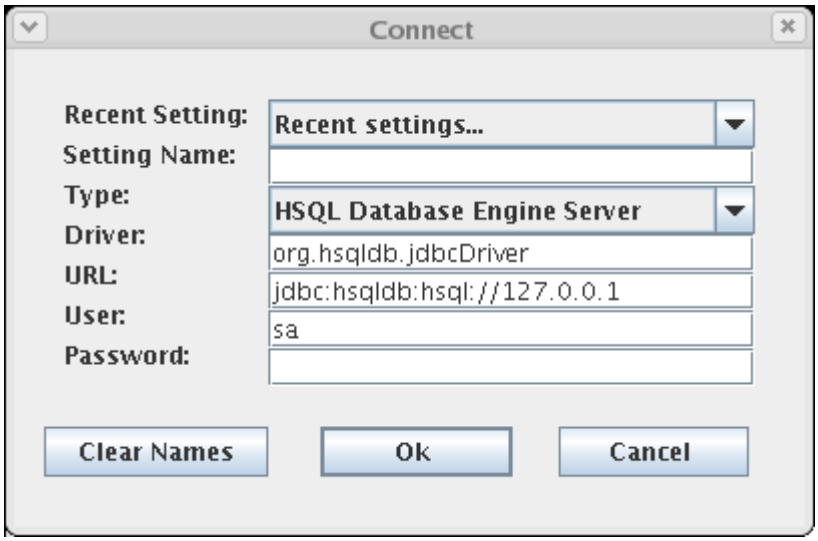

Figura 7. Diálogo de conexão para o servidor HSQLDB

Note ainda que incluir opções como **shutdown=true** na URL de conexão não tem efeito algum sobre o servidor remoto. É necessário conectar-se e enviar ao servidor o comando SQL **shutdown** para finalizá-lo corretamente.

Uma aplicação para acesso ao banco

A **Listagem 1** apresenta um programa Java simples, criado para ser executado diretamente pela linha de comando. O programa abre a nosso banco de teste, executa uma consulta SQL e exibe os resultados na saída padrão.

Observe que não há nada diferente de qualquer outro programa Java que utilize a API JDBC. A URL de conexão, assim como o usuário padrão "sa", sem senha, são os mesmos que utilizamos com o Database Manager nos exemplos anteriores. O nome da classe do driver JDBC também é o mesmo – embora este parâmetro possa ter passado desapercebido pelo leitor, pois o Database Manager já fornece o valor correto uma vez selecionado o tipo de banco de dados no combobox *Type*.

Então compile a classe e depois a execute com **java TesteHsqldb**. Tendo executado o programa com sucesso, experimente modificar a URL de conexão para acessar um servidor HSQLDB, em vez de embutir o engine na própria JVM da aplicação.

Com esse roteiro para iniciar no uso do HSQLDB e a escrita de aplicações que fazem uso desse banco de banco de dados, o leitor que começou por aqui pode passar ao corpo do artigo, onde será examinado mais a fundo a arquitetura e outras características do HSQLDB.

Listagem 1. *TesteHsqldb.java:* programa que conecta ao banco de dados de teste e realiza uma consulta

```
import java.sql.*;

public class TesteHsqldb {
  private static final String jdbcDriver =
      "org.hsqldb.jdbcDriver";

  private static final String jdbcUrl =
      "jdbc:hsqldb:file:/bd/teste;shutdown=true";

  private static final String jdbcUser = "sa";
  private static final String jdbcPasswd = "";

  public static void main(String[] args) throws Exception {
    Class.forName(jdbcDriver);
    Connection con = DriverManager.getConnection(
       jdbcUrl,jdbcUser, jdbcPasswd);
    String sql = "select nome, email from contato order by nome";
    Statement st = con.createStatement();
    ResultSet rs = st.executeQuery(sql);
    while (rs.next()) {
      System.out.println(rs.getString(1) + "\t<"
        + rs.getString(2) + ">");
    }
    rs.close();
```

```
    st.close();
    con.close();
  }
}
```

Listagem 2. Script para inicializar o banco de dados de teste usando o SqlTool (*teste.sql*)

```
create table contato (
  id identity primary key,
  nome varchar(40) not null,
  email varchar(30) not null
);

insert into contato (nome, email) values (
  'Fernando Lozano', 'fernando@lozano.eti.br');
insert into contato (nome, email) values (
  'Osvaldo Pinali Doederlein', 'osvaldo@visionnaire.com.br');
insert into contato (nome, email) values (
  'Java Magazine', 'info@javamagazine.com.br');

commit;
shutdown;
```

Listagem 3. Classe que define funções / procedimetnos armazenados para o HSQLDB (*ProcedimentoArmazenado.java*)

```
import java.sql.*;
import java.util.*;

public class ProcedimentoArmazenado {
  public static int maiusculas(Connection con,
     String tabela, String campo) throws Exception
 {
    String sql = "update " + tabela
      + " set " + campo + " = ucase(" + campo + ")";
    Statement st = con.createStatement();
    int nr = st.executeUpdate(sql);
    st.close();
    return nr;
```

```
  }

  public static boolean terminaEm(String texto, String sufixo) {
    return texto.toUpperCase().endsWith(sufixo.toUpperCase());
  }

  public static Object dominioEmail(Connection con, String sufixo)
      throws Exception
  {
    String sql = "select id, nome, email "
      + " from contato where "
      + " terminaem (email, '" + sufixo + "')";
    Statement st = con.createStatement();
    ResultSet rs = st.executeQuery(sql);
    List resultado = new ArrayList();
    while (rs.next()) {
      resultado.add(new Object[] {
        rs.getInt(1),
        rs.getString(2),
        rs.getString(3)
      });
    }
    rs.close();
    st.close();
    return resultado;
  }
}
```

Listagem 4. Classe Java que chama o procedimento armazenado dominioEmail (*TestaProc.java*)

```
//... imports omitidos

public class TesteProc {
  //... parâmetros de conexão iguais aos da Listagem 1

  public static void main(String[] args) throws Exception {
    Class.forName(jdbcDriver);
    Connection con = DriverManager.getConnection(
        jdbcUrl,jdbcUser, jdbcPasswd);
```

```
    String sql =
        "call dominioEmail('.com.br')";
    Statement st = con.createStatement();
    ResultSet rs = st.executeQuery(sql);
    while (rs.next()) {
      List resultado = (List)rs.getObject(1);
      Iterator it = resultado.iterator();
      while (it.hasNext()) {
        Object[] linha = (Object[])it.next();
        for (int i = 0; i < linha.length; i++)
          System.out.print(linha[i] + ", ");
        System.out.println();
      }
    }
    rs.close();  st.close();  con.close();
  }
}
```

Listagem 5. Simulando o retorno de ResultSets por um procedimento armazenado usando tabelas temporárias

```
import java.sql.*;
import java.util.*;

public class ProcedimentoArmazenadoTemp {
  public static int dominioEmailTemp(Connection con,
      String sufixo, String tabela) throws Exception
  {
    String sql = "create temp table " + tabela + "("
      + "id integer primary key, "
      + "nome varchar(40), "
      + "email varchar(30)) ";
    Statement st = con.createStatement();
    int nr = st.executeUpdate(sql);
    st.close();

    sql = "select id, nome, email "
      + "from contato where "
      + "terminaem (email, '" + sufixo + "')";
    st = con.createStatement();
```

```
    ResultSet rs = st.executeQuery(sql);

    while (rs.next()) {
      String sqlins = "insert into " + tabela + " values ("
        + rs.getInt(1) + ", "
        + "'" + rs.getString(2) + "', "
        + "'" + rs.getString(3) + "')";
      Statement stins = con.createStatement();
      nr = st.executeUpdate(sqlins);
      stins.close();
    }
    nr = rs.getRow();
    rs.close();
    st.close();
    return nr;
  }

  public static int dominioEmailTemp2(Connection con,
      String sufixo, String tabela) throws Exception
  {
    String sql = "select id, nome, email "
      + "into temp " + tabela + " "
      + "from contato where "
      + "terminaem (email, '" + sufixo + "')";
    System.out.println(sql);
    Statement st = con.createStatement();
    int nr = st.executeUpdate(sql);
    st.close();
    return nr;
  }
}
```

---

[1]Sandbox (caixa de areia) é o ambiente restrito onde uma JVM executa código que foi baixado da rede em vez de lido de arquivos locais. Ele impede que Applets Java, por exemplo, atuem como vírus.

[2] Quando se usa isolamento de transações, uma consulta deve retornar os dados como estavam quando a transação foi iniciada, sem "enxergar" as modificações feitas sobre os mesmos dados por outras transações. A maioria dos bancos então obtém, se necessário, os dados originais que são salvos no log de transações para o caso de um rollback.

por Fernando Lozano

Expert em Java e programação Web