

Truques com dados

Mostramos como alguns novos recursos do MySQL 5 ajudarão a melhorar o desenho de programas e melhorarão muito o desempenho dos aplicativos.
por Larkin Cunningham



Alfonso Romero - www.sxc.hu

O sistema de bancos de dados de código aberto MySQL serve muitas das maiores empresas no mundo, como Yahoo e Ticketmaster. Também está presente num grande número de websites com tráfego intenso, como a Wikipédia. Diversas organizações, no entanto, tradicionalmente preterem o MySQL em troca de sistemas de bancos de dados comerciais ricos em recursos, como o Oracle e o DB2. A partir do MySQL 5 [1], os desenvolvedores do MySQL começaram a introduzir uma gama de recursos profissionais que acabarão tornando-o mais competitivo com sistemas de bancos de dados comerciais. Este artigo examina alguns dos recursos profissionais agora presentes no MySQL. Muitos deles foram introduzidos na versão 5.0, e alguns podem vir a sê-lo na versão 5.1, que ainda estava em fase beta durante a confecção deste artigo, mas possivelmente já terá sido lançado quando você o ler. Neste artigo foi usada a versão 5.1.9-beta para confecção dos exemplos.

Três dos novos recursos mais atraentes no MySQL 5.x são os *stored procedures* (procedimentos armazenados), *triggers* (gatilhos), e *views* (visualizações). Esses recursos não são novos na indústria. A Oracle, por exemplo, primeiro apresentou o PL/SQL [2], sua implementação de uma linguagem procedural para SQL, em 1991. O Sybase, PostgreSQL e DB2 estão entre os outros sistemas de gerenciamento de bancos de dados com linguagens procedurais para SQL. Todavia, triggers, views

e stored procedures ainda são uma adição muito bem vinda ao MySQL.

Deve-se notar que alguns desses recursos profissionais do MySQL ainda estão em estágio inicial de desenvolvimento. Muitos desses recursos ainda estão incompletos ou não atingem níveis ótimos de desempenho. A versão 5.1 resolveu algumas dessas questões, e os novos recursos sem dúvida continuarão melhorando nas versões mais novas do MySQL.

Ao longo deste artigo, haverá referências às tabelas *produtos*, *cabecalhos_pedidos*, *linhas_pedidos*, *estoque* e *clientes*, para melhor ilustrar os exemplos. O Exemplo 1 mostra os comandos SQL *create table* que criam as tabelas. Ao dar exemplos de stored procedures, triggers e views, serão feitas referências às tabelas do Exemplo.

Stored procedures

Antes de explicar o que são stored procedures, é importante explicar que esse termo será usado aqui geralmente em referência à combinação de stored procedures e *stored functions* (funções armazenadas). Uma stored procedure aceita múltiplos parâmetros de entrada e saída. Uma stored function também aceita múltiplos parâmetros de entrada, mas retorna somente um único valor a quem a chamou. Essa restrição permite o uso de stored functions dentro de comandos SQL, o que efetivamente proporciona uma extensão das capacidades do SQL.

As stored procedures são uma ferramenta poderosa para um desenvolvedor ter em seu arsenal. Elas podem oferecer grandes benefícios em termos de desempenho e desenho de aplicações. Em relação ao desempenho, é possível reduzir muito tráfego de rede realizando um maior processamento de dados dentro do banco de dados. Ao reduzir o tráfego de rede, pode-se eliminar a latência associada à comunicação do servidor de aplicação com o servidor de banco de dados, principalmente quando eles residem em servidores separados, como é o caso na maioria das aplicações de grande escala.

Com stored procedures, é possível usar uma abordagem de caixa preta para o desenho e desenvolvimento de aplicações. Um desenvolvedor programando em Java, PHP, Ruby ou qualquer outra linguagem com suporte ao MySQL não precisa ter grandes conhecimentos em SQL ou PL/SQL. Em equipes de desenvolvimento com diversos membros, pode-se ter desenvolvedores de stored procedures concentrados no desenvolvimento de stored procedures, enquanto desenvolvedores de Java, PHP ou Ruby se dedicam a suas linguagens de programação específicas. Contudo que cada desenvolvedor esteja ciente das entradas e saídas esperadas, ambos os desenvolvedores podem trabalhar em paralelo. Essa pode ser uma forma de utilizar melhor os conhecimentos de seus desenvolvedores, caso o projeto

seja grande o suficiente para garantir os recursos de desenvolvimento.

A portabilidade também é favorecida por um maior desenvolvimento da sua lógica no banco de dados. Seria possível, por exemplo, desenvolver um programa em lote escrito em C, um aplicativo web usando Ruby on Rails, e um *web service* em Java, e todos utilizarem o mesmo conjunto de stored procedures.

A abordagem de várias pessoas ao desenvolver aplicativos que usem um banco de dados relacional é ou embutir todo o SQL dentro de seu código, ou embutir todo o SQL em stored procedures e simplesmente chamá-las a partir de seu código. Muitos desenvolvedores se baseiam em mapeadores objeto-relacionais como o *Hibernate* [3] (para Java) e o *ActiveRecord* [4] (para Ruby on Rails), onde stored procedures são altamente irrelevantes. Decidir qual abordagem usar para lidar com o processamento dos dados nas suas aplicações dependerá de fatores como desempenho e portabilidade. Se o desempenho não for importante, você pode considerar um mapeador objeto-relacional para gerar seu SQL dinamicamente. Mas, se você se importar com o desempenho e tiver necessidade de um certo número de transações por segundo, ou de um tempo de resposta em determinado número de milissegundos, você provavelmente terá interesse nos méritos do uso de stored procedures. Se você trabalha em um ambiente heterogêneo com muitas plataformas de desenvolvimento, então as stored procedures podem ser uma forma de você desenvolver sua lógica de processamento de dados só uma vez e numa localização central. Afinal, as stored procedures não se importam com a linguagem que faz a chamada.

Triggers

Os triggers têm muitos usos, incluindo tarefas básicas de manutenção, como auditoria e arquivamento. Eles podem ter diversos outros usos também. Um cenário comum é quando um trigger é acionado após uma linha ser criada, como por exemplo a adição de uma linha à tabela *linhas_pedidos*. Um trigger poderia ser acionado depois de a linha ser inserida para atualizar a quantidade do produto em estoque na tabela *estoque*.

Onde o arquivamento é necessário, pode-se ter mais uma tabela de arquivamento para cada tabela, onde se guardará a informação sobre os arquivamentos. Por exemplo, a tabela *produtos* pode ter uma

tabela associada *arquivo_produtos* com as mesmas colunas da tabela de produtos. Para arquivar automaticamente, você criaria triggers na tabela dos produtos para inserir uma linha na tabela *arquivo_produtos* após cada atualização ou remoção. Você não deve criar um trigger que seja acionado após uma inserção, pois ao buscar todo o histórico de um produto, seria retornada a união da linha da tabela *produtos* com as linhas associadas da tabela *arquivo_produtos*.

A abordagem é semelhante quando a auditoria é necessária. Em vez de ter uma tabela de arquivamento associada nos locais que precisam de arquivamento, é possível ter uma única tabela de auditoria. Para as tabelas para as quais você deseja manter uma trilha de atividades de auditoria, você pode ter triggers acionados após qualquer operação de adição, atualização ou remoção. Esses triggers inseririam uma linha na tabela de auditoria contendo a natureza da ação, a tabela afetada, o usuário que realizou a operação, a hora em que isso foi realizado e quaisquer outros dados que se julgar importantes. A abordagem de usar triggers no banco de dados para auditoria e não no código da sua aplicação pode reduzir o trabalho de programação de seus desenvolvedores e encorajar a consistência em ambientes onde diversas aplicações acessam o mesmo banco de dados. Existem muitas boas abordagens para a auditoria que podem ser empregadas no código da sua aplicação, então cada caso deve ser examinado em seu contexto.

Sobre views

Uma view é uma tabela virtual gerada a partir de uma consulta armazenada. A consulta armazenada frequentemente é composta por múltiplas operações de *join* que usam dados de diversas tabelas com algumas condições em comum. Num nível mais simples, pode ser só um subconjunto de uma tabela maior. Um exemplo trivial, novamente usando a tabela *produtos*, é criar uma view chamada *produtos_sem_estoque*, que une a tabela *produtos* à tabela *estoque*, nos itens com estoque zero.

As views podem ajudar a reduzir o código SQL para conjuntos de dados frequentemente acessados. As views também melhoram a eficiência, pois a consulta respectiva pode sofrer cache e ser carregada mais rápido que as diversas versões da mesma consulta rodando a partir de diferentes locais.

Linguagem procedural do MySQL

O MySQL 5 oferece uma linguagem procedural que você pode usar para criar suas stored procedures e triggers. Ao invés de usar uma linguagem procedural baseada em C ou *Python*, os desenvolvedores do MySQL criaram uma linguagem procedural de acordo com o padrão ANSI SQL:2003 [5]. O padrão ANSI é usado em diferentes graus pelos desenvolvedores de outros sistemas gerenciadores de bancos de dados, então, ao seguir o padrão, os conhecimentos adquiridos no desenvolvimento de stored procedures e triggers para o MySQL é transferível a outros bancos de dados como Oracle, DB2 e PostgreSQL, que possuem implementações semelhantes de linguagens procedurais.

Assim como as linguagens de programação com as quais você deve ter

Exemplo 1: O esquema do banco de dados

```
01 CREATE TABLE produtos (
02   id          MEDIUMINT NOT NULL AUTO_INCREMENT,
03   nome       CHAR(40) NOT NULL,
04   custo      DOUBLE(9,2) UNSIGNED DEFAULT 0.0,
05   PRIMARY KEY (id)
06 );
07
08 CREATE TABLE estoque (
09   id          MEDIUMINT NOT NULL AUTO_INCREMENT,
10   id_produto  MEDIUMINT NOT NULL,
11   quantidade  MEDIUMINT NOT NULL DEFAULT 0,
12   PRIMARY KEY (id)
13 );
14
15 CREATE TABLE cabecalhos_pedidos (
16   id          MEDIUMINT NOT NULL AUTO_INCREMENT,
17   id_cliente  MEDIUMINT NOT NULL,
18   data_pedido DATETIME NOT NULL,
19   status_pedido CHAR(1) DEFAULT '0',
20   PRIMARY KEY (id)
21 );
22
23 CREATE TABLE linhas_pedidos (
24   id          MEDIUMINT NOT NULL AUTO_INCREMENT,
25   id_pedido  MEDIUMINT NOT NULL,
26   id_produto  MEDIUMINT NOT NULL,
27   quantidade  MEDIUMINT NOT NULL DEFAULT 0,
28   PRIMARY KEY (id)
29 );
30
31 CREATE TABLE clientes (
32   id          MEDIUMINT NOT NULL AUTO_INCREMENT,
33   nome       VARCHAR(70) NOT NULL,
34   endereco   VARCHAR(200) NOT NULL,
35   telefone   VARCHAR(20) NOT NULL,
36   email      VARCHAR(40) NOT NULL,
37   PRIMARY KEY (id)
38 );
```

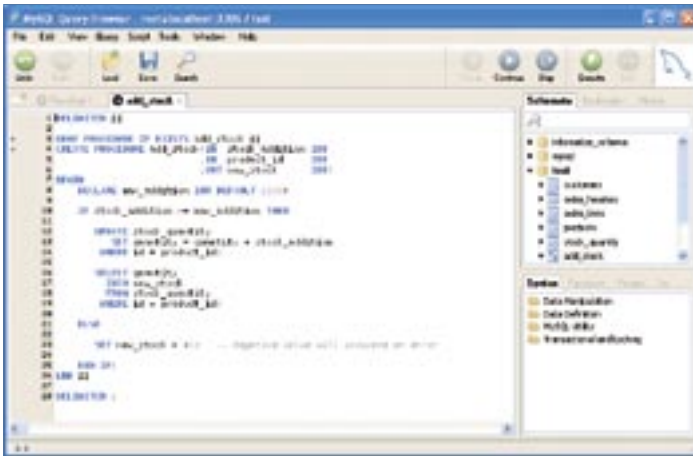


Figura 1 Uma *stored procedure* simples no navegador de consultas do MySQL.

intimidade, como PHP e Java, a linguagem procedural do MySQL possui as estruturas de que você precisa para escrever códigos úteis. Isso inclui comandos condicionais (*IF-THEN-ELSE* e *CASE-WHEN*) e comandos iterativos (*REPEAT-UNTIL* e *WHILE-DO*).

O comprimento deste artigo não nos permite apresentar todos os recursos da linguagem procedural do MySQL. Em vez disso, será explicado como as *stored procedures* e *triggers* do MySQL são estruturados e oferecem alguns exemplos simples que mostram, ainda que basicamente, o que as *stored procedures*, *triggers* e *views* realmente são. Se você for um programador experiente em qualquer linguagem moderna, a linguagem procedural do MySQL parecerá bastante simplista. Ela foi pensada como uma forma de oferecer entrada a comandos SQL e manipular seus resultados, e não como uma linguagem para competir com outras como PHP ou Java.

Estrutura de uma *stored procedure*

As *stored procedures* são escritas de forma a permitir que sejam criadas por qualquer ferramenta que execute SQL. Alguns dos exemplos aqui são mostrados no navegador de consultas do MySQL [6], uma ferramenta livre e muito útil do MySQL. Elas são escritas como scripts SQL, que basicamente informam ao MySQL o nome e os conteúdos da *stored procedure*. Se ela contiver erros, o MySQL informará o desenvolvedor quando este tentar criá-la.

A figura 1 mostra uma *stored procedure* que aceita um valor inteiro para

uma quantidade a ser adicionada ao estoque. Como o delimitador padrão no MySQL é um ponto-e-vírgula, e a linguagem procedural do MySQL utiliza esse caractere para terminar cada comando, precisamos mandar o MySQL mudar o delimitador ao tentarmos criar nossa *procedure*. A convenção normal é usar como delimitador dois cifrões seguidos, com o comando *DELIMITER \$\$* (linha 1). O

próximo comando (linha 3) manda o MySQL destruir a *stored procedure* de mesmo nome, caso exista uma. Caso contrário, esse comando é ignorado e o parser MySQL continua processando os comandos. A linha 4 faz o MySQL criar uma nova *stored procedure* com o nome e parâmetros fornecidos. Toda a lógica das *stored procedures* começa com o comando *BEGIN* (linha 7). Vários comandos declarativos, sequenciais, condicionais e iterativos podem ser usados antes de terminar a lógica da *stored procedure* com o comando *END* (linha 26). Note como o comando *END* é seguido de nossos delimitadores temporários, os dois cifrões. Isso acontece porque agora deixamos a *stored procedure* e voltamos a parsear o SQL do MySQL normalmente. Nesse ponto, vamos voltar ao delimitador padrão de ponto-e-vírgula (linha 28).

Variáveis, parâmetros e tipos de dados

Na figura 1, declaramos uma variável *adicao_maxima* (linha 8) e três parâmetros *adicao_estoque*, *id_produto* e *novo_estoque* (linhas 4 a 6). As palavras-chave *IN* e *OUT* informam ao MySQL que o parâmetro pode receber um valor de entrada, retornar um valor a quem o chamou, ou ambos (declarando que um parâmetro é *IN OUT*). Os parâmetros podem ser usados como variáveis normais, mas somente os *OUT* devem ter seus valores mudados na *procedure*.

As variáveis têm que ser explicitamente declaradas e receber um tipo e um valor padrão opcional. Os tipos a escolher pertencem aos tipos de dados SQL padrão que o MySQL suporta para colunas de tabelas. Todos os tipos de dados são escalares, ou seja, só podem guardar um único valor discreto. Isso elimina tipos de dados como vetores, que podem ser frustrantes para os desenvolvedores acostumados com linguagens como PHP e Java. Entretanto, há formas de contornar isso, como tabelas temporárias que usam um mecanismo de armazenamento em memória. Alguns dos tipos de dados típicos incluem *CHAR* e *VARCHAR* (para caracteres e strings), *DATE*, *DATETIME*, *INT* (incluindo *TINYINT*, *SMALLINT*, *MEDIUMINT* e *BIGINT*), *DECIMAL*, *FLOAT*, *DOUBLE* e outros. Grandes quantidades de dados podem ser armazenadas usando-se outros tipos de dados, como *TEXT* (até 64 KB) e *BLOB* (grande objeto binário – teoricamente é possível armazenar até 4 TB em um *LONGLOB*).

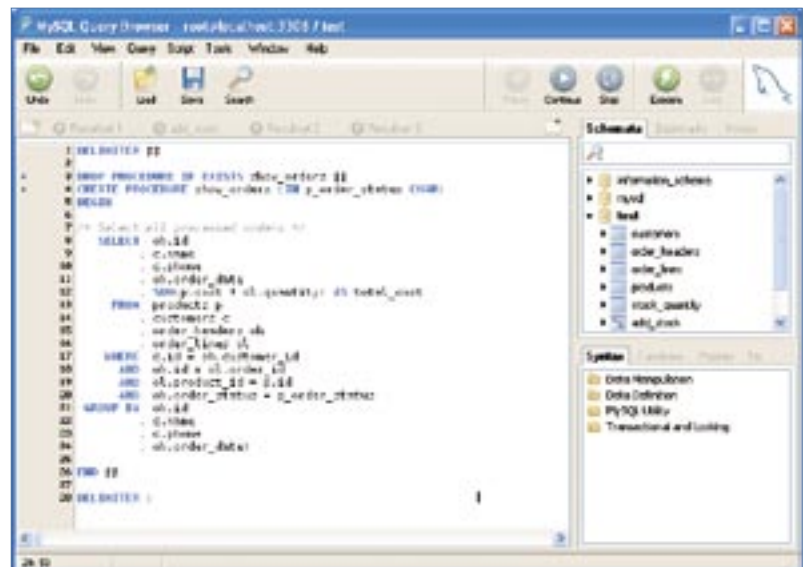


Figura 2 Uma *procedure* feita para retornar um conjunto de resultados a quem a chamou.

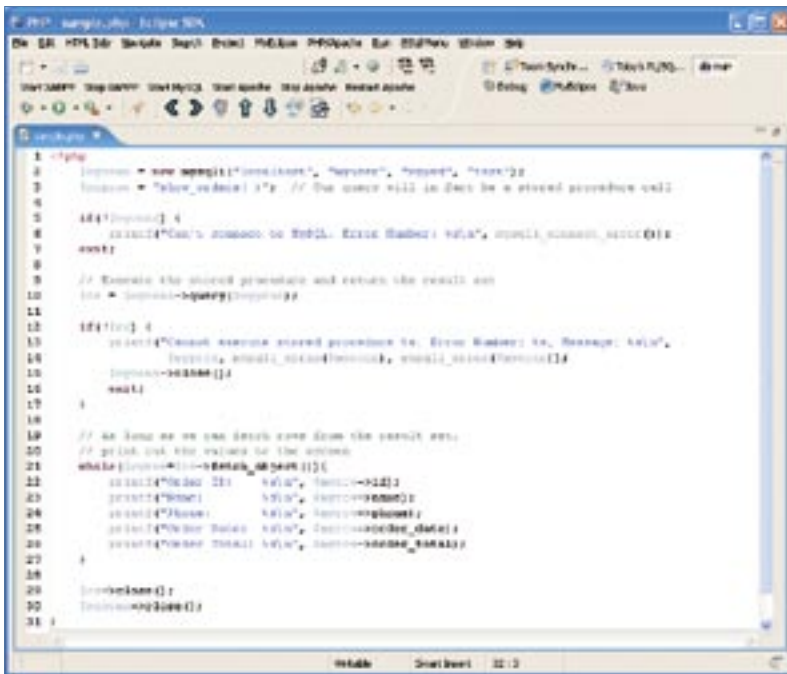


Figura 3 Exemplo de chamada de *stored procedure* em PHP usando *mysql*.

SQL em stored procedures

Diferentemente de outras linguagens de programação, como PHP e Java, não existem drivers com os quais se preocupar, nem funções ou métodos especiais para executar seu SQL. No lugar disso, os comandos SQL podem ser executados dinamicamente e os resultados inseridos diretamente das variáveis. Os comandos `UPDATE` e `INSERT` podem ler valores diretamente das variáveis e parâmetros.

Na [figura 1](#), um comando `UPDATE` ([linha 12](#)) mistura nomes de tabelas, de colunas e parâmetros. No comando `SELECT` seguinte ([linha 16](#)), um valor é selecionado e enviado diretamente para um parâmetro `OUT`. Como já foi dito, a linguagem procedural do MySQL é, no final das contas, uma forma de entrar dados em SQL e processar os resultados.

No comando `SELECT` ([linha 16](#)), na [figura 1](#), um valor foi selecionado e enviado para um parâmetro `OUT`. Supondo que a coluna `id` garanta que cada dado é único, isso não deve ter problemas. Mas e se o comando SQL retornar vários valores? Você só deve selecionar algo e enviar para uma variável se tiver certeza absoluta de que somente um valor será retornado. Esse será o caso quando o discriminador (os argumentos após a palavra-chave `WHERE`) usar uma chave única, como uma coluna de `id` usando um `auto_increment`, ou quando você selecionar enviar o valor de uma função

para uma variável, com `SUM()` ou `MAX()`, por exemplo. Como as variáveis são escalares e só conseguem guardar valores unitários, é eliminada a possibilidade de se retornar uma lista de valores diretamente para uma variável. É aí que o conceito de cursores chega para ajudar.

Cursores

Um cursor é um ponteiro para um conjunto de resultados retornados por uma consulta `SELECT`. Embora seja usado principalmente quando esse tipo de consulta retorna mais de uma linha, também é possível usar um cursor mesmo quando somente uma linha é retornada. Mesmo que nenhuma linha seja retornada, isso não gerará um erro. Porém, se tentarmos buscar uma linha de um resultado nulo ou se tentarmos buscar além da última linha dos resultados, um erro do MySQL será relatado.

O [exemplo 2](#) mostra uma forma possível para se utilizar cursores com `REPEAT-UNTIL`. Começamos a *procedure* declarando algumas variáveis padrão, incluindo uma com o nome de `nao_encontrado`. Essa variável é usada junto com um `HANDLER` para a condição `NOT FOUND` (NÃO ENCONTRADO). Ou seja, quando uma condição `NOT FOUND` for encontrada, como por exemplo quando o cursor ultrapassa seus limites, o valor de `nao_encontrado` será definido como 1 ou `TRUE`.

Nosso cursor `resumo_pedido_atual`, não é nada mais que um valor dado a uma variável de mesmo nome até que consigamos abri-la

Exemplo 2: Uma stored procedure com uso de um cursor

```

01 DELIMITER $$
02
03 DROP PROCEDURE IF EXISTS mostrar_pedidos_processados $$
04 CREATE PROCEDURE mostrar_pedidos_processados ()
05 BEGIN
06
07 DECLARE v_p_id          MEDIUMINT;
08 DECLARE v_c_nome        VARCHAR(70);
09 DECLARE v_c_telefone    VARCHAR(20);
10 DECLARE v_p_data        DATETIME;
11 DECLARE v_p_total       DOUBLE(9,2);
12 DECLARE nao_encontrado TINYINT;
13
14 /* Selecionar todos os pedidos processados */
15 DECLARE resumo_pedido_atual CURSOR FOR
16 SELECT oh.id
17        , c.nome
18        , c.telefone
19        , oh.data_pedido
20        , SUM(p.custo * ol.quantidade) AS custo_total
21 FROM   produtos p
22        , clientes c
23        , cabecalhos_pedidos oh
24        , linhas_pedidos ol
25 WHERE  c.id = oh.id_cliente
26        AND oh.id = ol.id_pedido
27        AND ol.id_produto = p.id
28        AND oh.status_pedido = 'P'
29 GROUP BY oh.id
30        , c.nome
31        , c.telefone
32        , oh.data_pedido;
33
34 DECLARE CONTINUE HANDLER FOR
35 NOT FOUND
36 SET nao_encontrado = 1;
37
38 SET nao_encontrado = 0;
39
40 OPEN resumo_pedido_atual;
41
42 laco_resumo_pedido:REPEAT
43
44 FETCH resumo_pedido_atual
45 INTO v_p_id
46        , v_c_nome
47        , v_c_telefone
48        , v_p_data
49        , v_p_total;
50
51 IF nao_encontrado THEN
52 LEAVE laco_resumo_pedido;
53 END IF;
54
55 SELECT CONCAT('ID do pedido: ', v_p_id, ', Nome: ',
56             v_c_nome,
57             ', Telefone: ', v_c_telefone, ',
58             Data do pedido: ', v_p_data,
59             ', Total do pedido: ', v_p_total);
60 UNTIL nao_encontrado
61 END REPEAT laco_resumo_pedido;
62 CLOSE resumo_pedido_atual;
63
64 END $$
65
66 DELIMITER ;

```

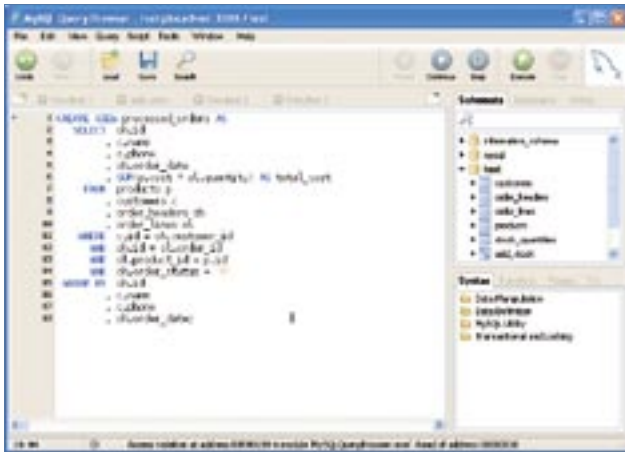


Figura 4 O SQL para criar a view.

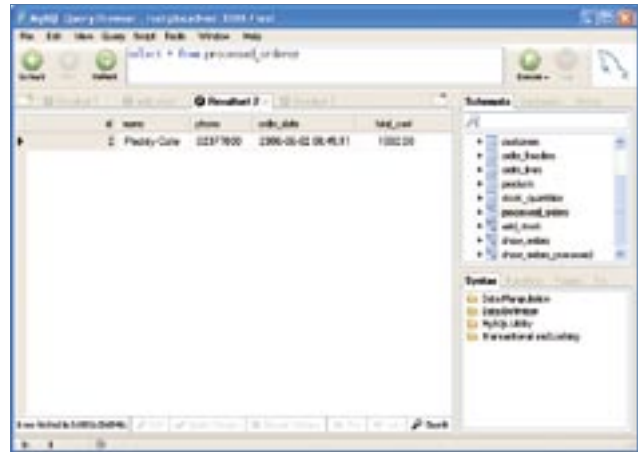


Figura 5 Uso de uma view como uma consulta normal.

com `OPEN`. Após a abertura, podemos começar buscando nosso cursor e enviando-o a variáveis na mesma ordem das colunas no comando `SELECT` do cursor. Para buscar todas as linhas retornadas pela nossa consulta de seleção, precisamos usar um comando iterativo. Existem muitas formas de fazer isso, e uma delas é `REPEAT-UNTIL`. Apesar de o nosso comando `REPEAT` continuar **ATÉ** (`UNTIL`) uma condição específica ser verdadeira (a variável `nao_encontrado`, no caso), nós temos a opção de sair (`LEAVE`) da iteração antes que a condição `UNTIL` seja atingida. Para isso, usamos um rótulo para a iteração, `taco_resumo_pedido`. Isso nos permite sair da iteração antes de usar alguma das variáveis buscadas, as quais, no caso de buscar além da última linha, resultará num erro do MySQL.

O comando `SELECT CONCAT` pode parecer um pouco estranho, mas é como exibimos os valores retornados pela consulta do nosso cursor.

Retornar conjuntos de resultados

Se você for um programador experiente que usa algo como PHP, Java, Python ou Ruby, pode estar se perguntando qual o objetivo de uma stored procedure como a do exemplo 2, já que ela só mostra o resultado no console ou no navegador de consultas

do MySQL. Ela não é muito útil se você quiser manipular os dados do conjunto de resultados do cursor. Porém, é possível retornar um conjunto de resultados ao programa que o gerou sem o cursor ou o código de manipulação.

Um comando SQL simples pode ser colocado numa stored procedure sem declarar um cursor e sem realizar um `SELECT` para alguma variável. Ele é escrito exatamente como você o escreveria, caso fosse executar seu SQL no navegador de consultas do MySQL ou no *phpMyAdmin* [7].

No exemplo 2, você pode simplesmente abandonar todos os comandos entre `BEGIN` e `END` que não sejam a consulta SQL. A figura 2 mostra essa pequena stored procedure.

Note como agora a stored procedure foi mudada para receber um parâmetro para selecionar pedidos de um status específico. Essa procedure ordenada pode potencialmente retornar o conjunto de resultados da consulta para um programa, supondo que a linguagem de programação desse programa suporte a busca desses conjuntos de resultados sem limites fixos.

É possível ter múltiplas consultas SQL como a da figura 2. Isso pode ser útil para conjuntos de dados relacionados, porém, é melhor ficar longe dessa abordagem e usar consultas únicas que retornem conjuntos de dados únicos.

Chamada de uma stored procedure

A figura 2 mostra um exemplo de código de chamada de uma stored procedure em PHP. No MySQL 5, deve-se ter instalada a extensão *mysqli*, para orientação a objetos, carregada ou compilada no PHP. A figura 3 mostra as chamadas de métodos usando o *mysqli*. A linha 10 demonstra a chamada à stored procedure. Note que ela não parece ser diferente de uma chamada SQL comum. A diretiva `while` na linha 21 itera através das linhas no conjunto de resultados retornado, assim como um conjunto de dados retornado a partir da execução de uma consulta SQL normal.

Usando triggers

Enquanto as stored procedures são iniciadas por chamadas diretas de acordo com sua necessidade, os triggers, por outro lado, são iniciados por eventos que causam o acionamento dos triggers. Esses eventos são inserções, atualizações e remoções.

De volta às tabelas do exemplo 1, podemos imaginar uma situação em que o cliente A pede o produto B. Uma saída para registrar isso em nosso sistema é executar uma stored procedure que insere o cabeçalho de um pedido e uma linha de pedido, e depois atualiza a quantidade estocada do produto. Outra abordagem seria que, sempre que uma linha de pedido fosse criada, a quantidade estocada correspondente ao tamanho do pedido fosse subtraída. Podemos considerar essa uma das nossas regras. Em vez de gravar a atualização a cada vez que um pedido é feito, podemos simplesmente criar um trigger que seja executado a cada vez que uma linha de pedido for inserida. Isso nos permite ter regras de negócios concretas embutidas no banco de dados.

Exemplo 3: Um trigger simples pós-atualização

```
01 DELIMITER $$
02 CREATE TRIGGER trg_ins_linhas_pedidos
03 AFTER UPDATE ON linhas_pedidos FOR EACH ROW
04 BEGIN
05     UPDATE estoque
06         SET quantidade = quantidade - NEW.quantidade
07         WHERE id_produto = NEW.id_produto;
08 END $$
09 DELIMITER ;
```

O exemplo 3 mostra um trigger acionado após cada atualização. A palavra-chave **NEW** se refere aos novos valores da linha, da forma como estão após o término da atualização. A palavra-chave **OLD** também está disponível, e contém os valores da linha da forma como estavam antes de uma atualização ou remoção. Para o arquivamento, por exemplo, você pode inserir os valores antigos da linha numa tabela de arquivamentos para acesso a dados de histórico. Para auditoria, você pode inserir tanto valores antigos quanto novos numa tabela de trilhas de auditoria.

Também é possível acionar um trigger antes de uma atualização, inserção ou remoção. Isso pode ser útil quando você quiser modificar os valores **NEW** da coluna, por exemplo, para validação.

Usando views

O exemplo 2, mostrou um atraente trecho de SQL que consultava resumos de pedidos processados. Em diversas aplicações, isso pode ser um útil conjunto de resultados para ter por perto e reutilizar em diversos pontos da aplicação. Uma view é um mecanismo para armazenarmos e reutilizarmos essas consultas úteis como se fossem tabelas normais. Assim, a complexidade subjacente da consulta fica escondida de nós, e podemos simplesmente selecionar as colunas dessa tabela virtual.

Views não podem aceitar parâmetros. Se você precisar que sua consulta aceite parâmetros, tem que criar uma stored procedure para ela. A figura 4 mostra a consulta do exemplo 3 criada como uma view. A figura 5 exhibe os resultados da execução de uma consulta usando a view. Note como ela se parece com uma consulta a qualquer tabela normal.

Próximos passos

Esse artigo apresentou alguns dos recursos das stored procedures, triggers e views do MySQL 5. Os exemplos tentam passar uma idéia sobre a utilidade (ou não) desses recursos para os seus esforços de desenvolvimento de software. Não se esqueça que, afinal de contas, as stored procedures são somente consultas SQL. Se você escrever SQL ineficiente em seu programa, provavelmente ainda terá consultas SQL ineficientes em suas stored procedures.

Os recursos apresentados são completamente novos no MySQL, mas quem já trabalhou com stored procedures em outros bancos de dados, como Oracle, DB2 e PostgreSQL, provavelmente se interessará mais pelas diferenças entre a implementação no MySQL e as outras. A linguagem procedu-

ral do MySQL ainda não está finalizada. As próximas versões do MySQL 5 (a 5.1 estava próxima da liberação geral quando este artigo foi escrito) e talvez o MySQL 6 devem aumentar o conjunto de recursos consideravelmente, e também melhorar algumas áreas onde a implementação do MySQL está pior que a dos adversários.

Uma avaliação gentil da documentação [9] dos novos recursos do MySQL no site diria que ela é, no máximo, adequada. Porém, diversos livros estão sendo publicados pela MySQL Press e outras editoras, mostrando mais detalhes dos novos recursos. ■

O autor

Larkin Cunningham ama programas de código aberto. Começando em breve um doutorado, Larkin atualmente trabalha com Oracle PL/SQL e Java, mas encontra tempo para se aventurar em tudo relacionado a Linux. Ele pode ser contactado em larkin.cunningham@gmail.com.

Mais Informações

- [1] MySQL 5.0 Community Edition: <http://www.mysql.com/products/database/mysql-community-edition.html>
- [2] Centro de tecnologia PL/SQL da Oracle: http://www.oracle.com/technology/tech/pl_sql/index.html
- [3] Mapeador Hibernate Objeto-relacional para Java e .NET: <http://www.hibernate.org/>
- [4] Mapeador ActiveRecord Objeto-relacional para Ruby: <http://rubyforge.org/projects/activerecord/>
- [5] Uma publicação sobre o ANSI SQL: <http://www.sigmod.org/sigmod/record/issues/0403/E.JimAndrew-standard.pdf>
- [6] Navegador de consultas do MySQL: <http://www.mysql.com/products/tools/query-browser/>
- [7] phpMyAdmin: <http://sourceforge.net/projects/phpmyadmin>
- [8] Extensão mysqli para PHP: <http://www.php.net/mysqli>
- [9] Documentação online escassa sobre stored procedures do MySQL: <http://dev.mysql.com/doc/refman/5.0/en/stored-procedures.html>



A Tecnologia da Informação na direção certa.

Soluções OPEN SOURCE

- SnapVIEW** Gerência e Monitoramento de Rede
- SnapFIREWALL** Sistema Avançado de Segurança com IDS/IPS/VPN
- SnapMAIL** Sistema Integrado de Correio Eletrônico com Anti-virus/Spam
- SnapHA** Sistema de Alta Disponibilidade com Cluster / Load Balance / QoS
- SnapCALL** Sistema de Telefonia / PBX IP / VoIP
- SnapFILE** Sistema de Compartilhamento de Arquivos/Impressoras com LDAP
- SnapSTATION** Estação de Trabalho LINUX Customizada
- SnapWEB** Gerência de Navegação Internet com Política de Permissões
- SnapHELP** Sistema de Help-Desk e Gerência de Projetos
- SnapCHAT** Sistema WEB de Bate-Papo
- SnapLOOK** Sistema WEB de Câmeras
- SnapTRACK** Sistema de Inventário de Software e Hardware Multiplataforma

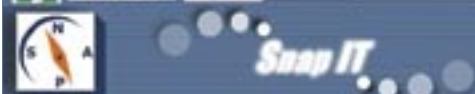
Caso de Sucesso



Alex Armengol - Gerente de TI da TOTAL ELF Lubrificantes

Alex >> "As soluções da SnapIT foram implementadas na TOTAL ELF Lubrificantes do Brasil há vários atrás, nos beneficiando com a redução de custos com licenças de Softwares e tomando o nosso ambiente de TI muito mais estável, seguro e performático. O comprometimento e a experiência da SnapIT é o resultado desta parceria de sucesso".

Outros Casos de Sucesso



www.snapit.com.br
Tel.: +55.11.3731-8008