Artigo

Utilizando Stored Procedures, Views e Triggers no MySQL

Este artigo descreve a continuidade do projeto de um banco de dados para um sistema de locação de carros, onde são usados recursos de stored procedures, views e triggers no MySQL para simplificar as operações a serem realizadas por este sistema.

De que se trata o artigo:

Este artigo descreve a continuidade do projeto de um banco de dados para um sistema de locação de carros, onde são usados recursos de *stored procedures*, *views* e *triggers* no MySQL para simplificar as operações a serem realizadas por

do banco de dados e seus objetos de apoio, como *stored procedures*, *views* e *triggers*.

Em que situação o tema é útil:

Stored procedures, *views* e *triggers* são recursos importantes a serem aplicados na implementação de um banco de dados, pois simplificam diversas operações a serem realizadas pelo banco de dados, assim como torna mais simples a codificação do sistema, definindo uma camada intermediária de controle entre o banco de dados físico e o código fonte da aplicação.

Os SGBDs atuais oferecem diversos recursos que facilitam as tarefas de manipulação de dados e, consequentemente, o gerenciamento de um banco de dados. Dentre os principais recursos, que não são tão inovadores, mas que possuem uma importância gigantesca no trabalho de um DBA podemos citar as *stored procedures*, *views* e *triggers*.

Neste artigo, iremos dar continuidade ao desenvolvimento de uma aplicação responsável pelo gerenciamento de uma locadora de carros apresentada no artigo "Modelando um Sistema de Reserva de Carros", publicado também nesta edição da SQL Magazine. Até o momento já possuímos suas tabelas e relacionamentos, nada mais que isso. Na continuidade deste projeto, iremos construir *Stored Procedures* (SPs), *Views* e *Triggers* que nos apóiam na realização das diversas operações realizadas por esta empresa.

Stored Procedures

Por exemplo, uma *stored procedure* pode contar comandos de loop ou condicionais, como *do/while*,*if/then/else*, dentre outros, permitindo a realização de operações mais complexas. Ou seja, uma SP funciona como um procedimento sobre o banco de dados, de forma similar aos procedimentos que usamos durante a codificação usando linguagens procedurais, tais como C ou Pascal.

Possuir códigos SQL em forma de *stored procedures* possibilita à aplicação minimizar a quantidade de código SQL que aparece no código fonte da aplicação e coloca estes códigos sob o controle da camada do banco de dados. Isso faz com que o projeto da aplicação fique mais claro e deixa as páginas de código fonte da aplicação livres de códigos SQL dinâmicos, que seriam desnecessários. Com isso, caso haja a necessidade de mudar alguma instrução em SQL, basta alterarmos o código da *stored procedure*, sem haver a necessidade de alterar o código fonte da aplicação e ter que recompilá-la.

No MySQL, *stored procedures* são rotinas criadas com as instruçõesCREATE PROCEDURE. Sua chamada ocorre através da instruçãoCALLe só pode passar valores de retorno usando variáveis de saída.

Atualmente o MySQL só preserva o contexto para o banco de dados padrão. Uma rotina herda o banco de dados padrão de quem a chama, assim geralmente as rotinas devem utilizar uma instruçãoUSE *nome_bancodedados*, ou então devemos especifique todas as tabelas com uma referência de banco de dados explicita, ex: *nome_bancodedados.nome_tabela*.

O MySQL suporta uma extensão muito útil que permite o uso da instrução

```
CREATE PROCEDURE nome_sp ([parametro[,...]])
[caracteristica ...] corpo_rotina

parameter:
  [ IN | OUT | INOUT ] nome_parametro tipo

tipo:
  Qualquer tipo de dados válidos no MySQL

caracteristica:
    LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | SQL SECURITY {DEFINER | INVOKER}
  | COMMENT string

corpo_rotina:
  Declaração(ões) de procedimento em SQL válida
```

A lista de parâmetros entre parênteses deve estar sempre presente. Se não houver parâmetros, uma lista de parâmetros vazia de“()”deve ser usada. Cada parâmetro é um parâmetroINpor padrão. Para especificar outro tipo de parâmetro, use a palavra chaveOUTouINOUTantes do nome do parâmetro.

Além do CREATE PROCEDURE, existem comandos para alterar (ALTER PROCEDURE), excluir (DROP PROCEDURE) e visualizar o código de criação (SHOW CREATE TABLE) de uma SP. Ao longo do artigo usaremos este comando em exemplos práticos associados ao nosso estudo de caso.

Views

computada ou colada a partir de dados existentes no banco de dados. Ao alterar os dados em uma tabela, os dados exibidos por uma *view* serão alterados automaticamente.

Views podem prover diversas vantagens ao ser comparadas com tabelas:

• *Views* podem representar um subconjunto dos dados contidos em uma tabela.

• *Views* podem realizar junções e simplificar múltiplas tabelas em uma simples tabela virtual.

• *Views* podem atuar como tabelas agregadas, onde o banco de dados agrega dados (somas,média, etc) e apresenta os resultados calculados como parte dos dados.

• *Views* podem ocultar a complexidade dos dados; por exemplo, uma *view* pode ser exibida como Locacões2000 ou Locações2001, particionando de forma transparente a tabela atual.

• *Views* usam muito pouco espaço para serem armazenadas; o banco de dados contém apenas a definição de uma *view*, não uma cópia de todos os dados que ela apresenta.

• Dependendo do SGBD adotado, *views* podem prover uma segurança extra;

• *Views* podem limitar o nível de exposição de uma tabela para o "mundo externo" à aplicação.

O MySQL suporta views desde a sua versão 5.x e quase todas as funcionalidades

• O MySQL primeiro combina as consultas e parâmetros de entrada que definem uma view, então o MySQL executa esta consulta.

O MySQL permite ainda criar *views* de *views*. Na declaração SELECT da definição de uma view, podemos referenciar outras *views*.

A instrução CREATE VIEWcria uma nova *view*. Sua sintaxe está apresentada na **Listagem 2**. Para alterar a definição de uma *view* ou excluir uma *view*, use, respectivamente, ALTER VIEWouDROP VIEW.

Listagem 2. Sintaxe para criação de uma *View* no MySQL

```
CREATE
    [OU REPLACE]
    [ALGORITMO = {UNDEFINED | MERGE | TEMPTABLE}]
    [DEFINIDOR = { user | CURRENT_USER }]
    [SEGURANÇA SQL { DEFINER | INVOKER }]
    VIEW nome_view [(lista_colunas)]
    AS declaração_select
    [WITH [CASCADED | LOCAL] CHECK OPTION]
```

Uma view pode ser criada a partir de vários tipos de declarações de SELECT. Ela pode se referenciar a outras tabelas ou views, ela pode usar JOINs, UNION e subconsultas. O SELECT não precisa se referir a qualquer tabela. O exemplo apresentado na **Listagem 3** define uma view que seleciona duas colunas (quantidade e preço) a partir de uma tabela de Vendas, assim como o resultado da multiplicação entre essas colunas.

Listagem 3. Exemplo de *View* no MySQL

```
+------+-------+-------+
|    3 |    50 |   150 |
|    5 |    60 |   300 |
+------+-------+-------+
mysql> SELECT * FROM v WHERE qtde = 5;
+------+-------+-------+
| qtde | preco | valor |
+------+-------+-------+
|    5 |    60 |   300 |
+------+-------+-------+
```

Triggers

Uma *trigger*consiste em um código procedural que é executado automaticamente em resposta a certos eventos em uma tabela ou *view* particular em um banco de dados. *Trigger* é comumente usada para manter a integridade das informações em um banco de dados. Por exemplo, quando um novo registro (representando um novo funcionário) é adicionado à tabela de empregados, novos registros devem ser criados também nas tabelas de taxas, férias e salários.

Triggers são comumente usadas para:

• Prevenir mudanças a ocorrerem em um banco de dados;

• Registrar as mudanças em um arquivo de log (ex: manter uma cópia dos dados antigos);

• Auditar mudanças (ex: manter um log dos usuários e papéis envolvidos em

• Replicar dados (ex: armazenar um registro de todas as mudanças para que ele seja persistido em outro banco de dados posteriormente);

• Garantir desempenho (ex: atualizar o saldo de uma conta corrente após uma transação, para consultas rápidas ao saldo).

O apoio a *triggers* foi incluído no MySQL a partir de sua versão 5.0.2. Uma *trigger* é um objeto de banco de dados associado a uma tabela e que é ativado quando um evento particular ocorre na tabela em questão. Alguns usos para *triggers* são feitos para verificar valores inseridos em uma tabela ou para realizar cálculos sobre valores envolvidos em uma atualização.

Uma *trigger* é definida para ser ativada quando uma instrução de INSERT,DELETE, ouUPDATEexecuta na tabela associada. Uma *trigger* pode ser configurada para ser ativada antes ou após a instrução. Por exemplo, podemos ter uma *trigger* que será ativada antes que cada linha de uma tabela seja inserida ou após a atualização de uma linha da tabela.

> **Importante:** Triggers no MySQL são ativadas apenas por instruções SQL. Elas não são ativadas por mudanças em tabelas realizadas por APIs que não transmitem instruções SQL ao servidor MySQL.

Para criar ou excluir uma *trigger*, use, respectivamente, a instrução CREATE TRIGGERou DROP TRIGGER. A sintaxe da instrução de criação está apresentada

```
4 |     ON nome_tabela FOR EACH ROW declaração_trigger
```

Esta declaração cria uma nova *trigger*. Esta *trigger* estaria associada à tabela *nome_tabela*, que deve se referir a uma tabela permanente.

Vejamos na **Listagem 5** um exemplo simples que associa uma *trigger* a uma tabela para instruções INSERT. A *trigger* atua como um acumulador, somando os valores inseridos em uma coluna da tabela.

Listagem 5. Exemplo de uma *Trigger* no MySQL

```
1 | mysql> CREATE TABLE conta (numero_conta INT, quantidade DECIMAL(10,2
2 | Query OK, 0 rows affected (0.03 sec)
3 |
4 | mysql> CREATE TRIGGER inserir_soma BEFORE INSERT ON conta
5 |     -> FOR EACH ROW SET @soma = @soma + NEW.quantidade;
6 | Query OK, 0 rows affected (0.06 sec)
```

A instrução CREATE TRIGGERcria uma *trigger* chamada *inserir_soma*que está associada à tabela *conta*. Ela também inclui uma cláusula que especifica o tempo de ativação da *trigger*, o evento de disparo e o que ela deve fazer ao ser ativada (somar o valor inserido).

A partir de agora iremos aplicar estes três conceitos em nosso estudo de caso da ACDN *Rental Car*.

Recapitulando o Estudo de Caso: ACDN Rental

de carros e vários preços de locação.

Entre as principais regras de negócio da empresa, podemos citar:

• A locação de um veículo pode ser feita em uma sede e a devolução na outra sede, sem restrição. Caso o ponto de devolução seja diferente do ponto de locação, ocorrerá uma multa.

• Os carros são agrupados em classes. O preço de locação varia por classe do carro e tempo de locação.

• Um cliente não pode realizar uma segunda reserva caso ainda possua uma locação aberta (cujo carro ainda não foi retornado).

O modelo físico (ver **Nota DevMan 1**) do nosso banco de dados *AcdnRentalCar* está apresentado na **Figura 1**, onde usamos a ferramenta DBDesigner. Observe nesta figura que são apresentadas as tabelas, seus atributos e relacionamentos, explicitando as chaves estrangeiras de cada tabela que instanciam os relacionamentos.

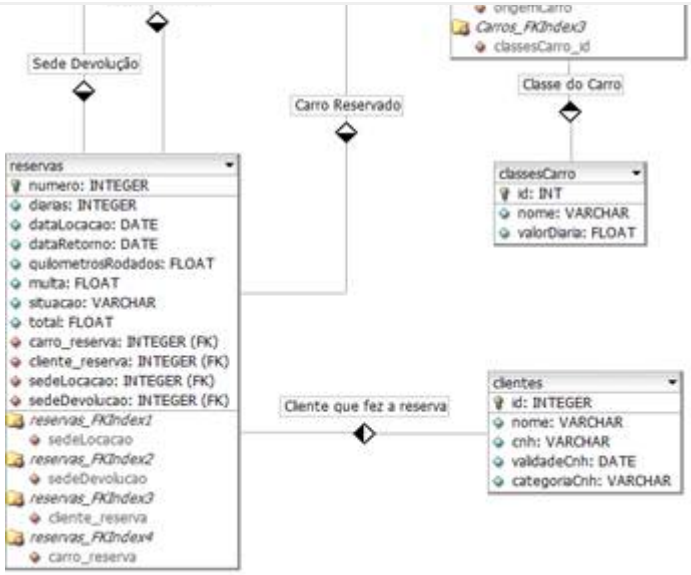

Figura 1. Modelo Físico do Sistema da *ACDN Rental Car*

Nota DevMan 1. Modelo Físico

Um modelo de dados físico é uma representação de um projeto de dados que leva em consideração as facilidades e restrições de um certo sistema gerenciador de banco de dados (SGBD). No ciclo de vida de um projeto, ele é tipicamente derivado a partir do modelo lógico. Um modelo de dados físico completo incluirá todos os artefatos de banco de dados necessários para criar relacionamentos entre tabelas ou obter os objetivos de desempenho, tais como índices, restrições, views, particionamentos, etc.

Por diversas vezes as pessoas tendem a confundir modelo de dados lógico e físico. A Tabela abaixo tenta esclarecer um pouco a diferença entre eles.

Usa nome de elementos do domínio para seus atributos	Usa nomes de colunas abreviados limitados pelo SGBD
É independente de tecnologia (ex: plataforma ou SGBD)	Inclui chaves primárias e índices para acesso rápido aos dados
É normalizado até a quarta forma normal (ver artigo “Aplicando Normalização a um Modelo de Dados” publicado na edição 58 da SQL Magazine)	Pode ser desnormalizado para atingir requisites de desempenho baseado na natureza do banco de dados

Tabela 1. Comparação entre Modelo Lógico x Modelo Físico

Para atender à demanda de operações do sistema, o proprietário da empresa fez algumas solicitações aos desenvolvedores. São elas:

• Ao alugar um carro, este deve ficar inacessível aos demais clientes que desejam realizar uma reserva. Este só passa a ficar acessível novamente após a devolução do veículo ao seu ponto de origem.

• O proprietário da empresa deseja obter constantemente um relatório que apresenta qual o veículo que foi alugado pela empresa, exibindo os dados do veículo, total da quilometragem rodada nas locações do período em questão, a quantidade de locações realizadas e, por fim, uma média de quilômetros rodados

realizando os ajustes necessários em nosso banco de dados.

Construindo Stored Procedures

Iniciaremos atendo à primeira demanda, onde foi solicitado que ao realizar uma venda, o carro fica impossibilitado de fazer uma nova reserva até que este seja devolvido ao seu ponto de origem.

Observem que nesta situação a operação de LOCAÇÃO possui duas tarefas: (1) registrar a locação do veículo na tabela de *reserva* e (2) atualizar a situação do carro para *alugado* em sua tabela. Da mesma forma, a operação de DEVOLUÇÃO possuirá duas tarefas: (1) registrar a devolução do veículo na tabela de reserva e (2) atualizar a situação do carro para *disponível*, caso esteja sendo devolvido em seu ponto de origem, ou *fora do ponto de origem*, caso esteja sendo devolvido em um ponto diferente. Essas operações podem ser implementadas com mais facilidade se usarmos os recursos disponíveis pelas *stored procedures*. As **Listagens 6** e **7** apresentam os códigos referentes a essas duas operações.

Listagem 6. Stored procedure para a operação de LOCAÇÃO

```
1  CREATE PROCEDURE sp_registrarLocacao (v_diarias int(10), v_dataLocac
2    BEGIN
3    // Inserindo registro de nova reserva
4  INSERT INTO reservas (diarias, dataLocacao, carro_reserva, sedeLocac
5
6
7  // Atualizando a situação do carro
```

multa a ser paga pelo cliente e que já foram obtidas as informações sobre a sede de locação.

Listagem 7. Stored procedure para a operação de DEVOLUÇÃO

```
CREATE PROCEDURE sp_registrarDevolucao (v_reserva int(10), v_dataRet
  BEGIN
  // Atualizando registro de nova reserva
UPDATE reservas SET dataRetorno = v_dataRetorno, quilometrosRodados


// Atualizando a situação do carro
IF (v_sedeLocalcao != v_sedeDevolucao) THEN
  UPDATE carros WHERE id = v_carro SET situação = “fora do ponto de
ELSE
  UPDATE carros WHERE id = v_carro SET situação = “disponível”;
END IF
  END;
```

Construindo Views

Vamos agora à segunda solicitação: um relatório com o carro mais alugado em um certo período de tempo.

Observe que para esta operação precisamos de diversas informações disponíveis em diferentes tabelas (como a tabela de *carros* e a de *reservas*). Para implementar esta necessidade, optamos pela criação de uma visão que possibilite exibir os dados desejados pelo cliente. A **Listagem 8** apresenta o código referente à criação desta *view*.

```
4 FROM carros AS c
5 JOIN reservas AS r ON c.id = r.carro_reserva
6 GROUP BY r.carro_reserva
```

Se executarmos essa view para algum carro específico informado como parâmetro (ex: ID = 1), observamos que esta retorna exatamente o que o usuário deseja. A **Listagem 9** apresenta um exemplo de utilização da *view* criada e seu resultado.

Listagem 9. Utilizando a *view* consultarReservasCarro

```
1 SELECT * FROM consultarReservasCarro where id =1;
2 +---+--------+--------------+---------+--------+----------+--------
3 | id| PLACA  | MODELO       | ANO     | COR    |SITUACAO  |DESCRICA
4 +---+--------+--------------+---------+--------+----------+--------
5 | 1 |JWL-2981|Sandero Stepway|2009/2010|Vermelho|disponivel| Lindo
6 +---+--------+--------------+---------+--------+----------+--------
```

Construindo Triggers

Vamos agora à terceira solicitação: manter atualizada a quilometragem rodada por um carro após a sua devolução.

Observe nessa situação que o ideal é que assim que um carro for devolvido seja feita a atualização da quilometragem do veículo. No entanto, fazer isso de forma manual seria muito custoso, e caso algum operador do sistema esquecesse ou colocasse um valor diferente daquele registrado na operação de devolução

na locação que está sendo finalizada.

A **Listagem 10** apresenta o código da *trigger* responsável por esta operação em nosso banco de dados.

Listagem 10. Triggers para manter consistente a quilometragem rodada por um carro

```
1 CREATE TRIGGER tr_kmRodados after update on reservas
2 FOR EACH ROW
3 UPDATE carros SET quilometragem = quilometragem + NEW.quilometrosRoc
```

Na **Listagem 11** podemos ver a *trigger* em funcionamento. Observe que antes de atualizarmos o registro da tabela reservas a quilometragem do carro com ID = 1 era 600 kms, e após a atualização deste registro ela passou a 750 kms, pois foi somado a ela o total de quilômetros rodados na locação finalizada, que foi de 150 quilômetros.

Listagem 11. Exibindo o funcionamento da trigger *tr_kmRodados*

```
1 SEELCT id, placa, modelo, quilometragem FROM carros WHERE id =1;
2 +----+----------+-----------------+---------------+
3 | id | placa    | modelo          | quilometragem |
4 +----+----------+-----------------+---------------+
5 |  1 | JWL-2981 | Sandero Stepway |        600.00 |
6 +----+----------+-----------------+---------------+
7
8 UPDATE reservas SET quilometrosRodados = 150 WHERE  numero = 1;
9
```

```
+----+----------+-----------------+---------------+
|  1 | JWL-2981 | Sandero Stepway |        600.00 |
+----+----------+-----------------+---------------+
```

Com isso, atendemos às três demandas de nosso cliente de forma simples e eficiente, utilizando os recursos providos pelo nosso SGBD (MySQL). Os passos seguintes a serem realizados seriam a implementação de novas *SPs*, *Views* e *Triggers*, assim como passar a olhar a questão de desempenho do banco de dados através da aplicação de técnicas de *tuning*, evitando um decaimento no desempenho do nosso banco de dados à medida que um grande volume de registros seja inserido.

Conclusões

Neste artigo vimos de uma forma prática como utilizar alguns dos principais recursos de banco de dados: *SPs*, *Views* e *Triggers*. Estes recursos fazem parte do dia-a-dia de qualquer DBA, pois constantemente nos deparamos com situações onde eles nos auxiliam de forma satisfatória.

Vimos que a aplicação destes recursos é de certa forma simples, quando utilizamos o MySQL. Para demonstrar tal simplicidade, evoluímos o projeto do banco de dados de uma locadora de carros, já publicado em edições anteriores da SQL Magazine. Dessa forma, podemos ver uma possível forma de evolução do modelo e projeto de um banco de dados.

Abraços e até a próxima!

Por **Arilo**

Em 2010

Falar com professor - Tire a sua dúvida.

Poste aqui a sua dúvida, nessa seção só você e o consultor podem ver os seus comentários.

Enviar dúvida

Você está em

Guia de MySQL » **Stored Procedures**