

Bônus!
Apenas
R\$ 8,90

Desenvolvedores de Software e DBAs

SQL magazine

Edição 31 :: Ano 3 : R\$ 8,90



Agora é briga!

Saiba como utilizar
os novos BDs
GRATUITOS da
Microsoft e Oracle

Tutorial Firebird

Utilizando o Firebird com
Delphi e Visual Studio

Tuning SQL

A importância da otimização
em comandos SQL

Tutorial Stored Procedures

Confira passo a passo como
trabalhar com SPs no MySQL 5

Linguagem SQL: Utilizando Constraints

Confira mais uma forma de deixar
a lógica da aplicação no BD

Resposta do Desafio

Modelando um BD pra controle
de grade horária semanal

Especial:
Concurso BNDES
Confira a segunda parte
do gabarito comentado
deste concurso público

ISSN 1677918-5



9 771677 918004

0 0 0 3 1

Quem nunca desejou trabalhar com o Oracle ou SQL Server e não ter que pagar por isso? Recentemente, as duas principais empresas desenvolvedoras de SGBDs (Microsoft e Oracle) lançaram novas versões de seus SGBDs, e o melhor de tudo, com opções gratuitas. Os novos lançamentos do SQL Server 2005 e do Oracle 10g têm agora versões sem custo, o SQL Server Express e o Oracle XE (Express Edition). Descubra nesta edição da SQL Magazine as principais limitações, características e como proceder com a instalação de ambos os SGBDs.

Destacamos também a coluna Desafio da SQL Magazine. Temos a resposta para o desafio lançado na edição 30 e mais um super desafio de modelagem para você.

Temos ainda nesta edição um artigo onde são apresentadas através de exemplos as novidades em segurança do SQL Server 2005.

Gostaria de destacar também três matérias interessantíssimas sobre SGBDs livres: Constraints no PostgreSQL, Stored Procedures no MySQL e, Instalação, Configuração e Acesso ao Firebird.

Com um total de 10 artigos, esta edição da SQL Magazine aborda temas bastante variados. Desejo a todos vocês uma excelente leitura.

Abraços,

Rodrigo Oliveira Spínola
editor@sqlmagazine.com.br



Segurança no SQL Server 2005

Conheça as novidades6

Carlos Eduardo Selonke de Souza

Microsoft cria nova geração de certificações16

Dalton Gerth

Instalação, Configuração e Acesso ao Firebird26

Luciano Pimenta

Desafio SQL Magazine30

Marco Antonio Pereira Araújo

Resposta do Desafio: Grade Horária Semanal36

Mansueto Gomes de Almeida

Índice

Bancos da Dados Gratuitos40

Cesar Blumm

Tutorial SQL: Constraints43

Bruce Momjian

Stored Procedures no MySQL 52

Ian Gilfillan

Questões de Banco de dados do concurso BNDES58

Mauro Pichiani

Tuning SQL64

Eduardo Gonçalves

Espaço do anunciante

Espaços Publicitários
Divulgue sua marca ou produto entre os melhores desenvolvedores do Brasil. Anuncie nas revistas do Grupo DevMedia.

Reprints Editoriais

Se foi publicado em nossas revistas uma matéria sobre seu produto ou serviço, ou simplesmente uma matéria que poderá alavancar suas vendas, multiplique essa oportunidade! Você pode reimprimir essa matéria junto com a capa da revista e distribuir entre seus clientes.

Essa ferramenta aumenta a visibilidade e a credibilidade de seu produto. Aproveite!

Encarte de CD

Faça como nossos maiores anunciantes encarte um CD com uma mostra de sua tecnologia na capa de nossas revistas. Seu produto chegará às mãos de um público segmentado e formador de opinião. Consulte nossos preços.

Encarte direcionado

Se sua empresa atua somente em uma determinada região do Brasil você pode fazer ações direcionadas atingindo apenas os leitores de seu interesse. Entre em contato com a gente!

Kaline Dolabella
publicidade@devmedia.com.br
(21) 2213-0940

Edições anteriores

Adquira as edições anteriores da revista SQL Magazine ou de qualquer outra revista do Grupo DevMedia através do nosso site de forma rápida, prática e segura. Tem sempre uma boa promoção esperando por você: www.devmedia.com.br/anteriores

Coordenador Editorial

Gladstone Matos
gladstone@clubedelphi.net

Editor Geral

Rodrigo Oliveira Spínola
rodrigo@sqlmagazine.com.br

Equipe Editorial

Paulo Ribeiro
paulo@sqlmagazine.com.br

Ricardo Resende
ricardo@sqlmagazine.com.br

Consultor Editorial

Emerson Facunte
facunte@devmedia.com.br

Gerente de Marketing

Kaline Dolabella
kalined@terra.com.br

Articulistas desta edição

Carlos Eduardo Selonke de Souza, Dalton Gerth, Luciano Pimenta, Marco Antonio Pereira Araújo, Cesar Blumm, Bruce Momjian, Ian Gilfillan, Mauro Pichiani, Eduardo Gonçalves e Mansueto Gomes de Almeida

Diagramação, Ilustrações e Capa

Cidadelas Produções
cidadelas@cidadelas.com.br

Vinicius O. Andrade
viniciusoandrade@gmail.com

Revisão

Alfredo Ferreira
alfredo@devmedia.com.br

Distribuição

Fernando Chinaglia Dist. S/A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Realização



Apoio



Atendimento ao leitor

A DevMedia possui uma Central de Atendimento, inteiramente on-line, onde você poderá retirar suas dúvidas sobre nossos serviços; enviar críticas e sugestões e falar diretamente com um de nossos atendentes. Através da Central de Atendimento também é possível alterar seus dados cadastrais, consultar o status de sua assinatura e conferir a data de envio de suas revistas. Acesse agora mesmo: www.devmedia.com.br/central. Se preferir entrar em contato através do telefone: (21) 2283-9012

Para sugerir matérias, enviar artigos, fazer críticas ou sugestões sobre o conteúdo da revista SQL Magazine envie seu e-mail diretamente para nosso editor. É muito importante para nós conhecer sua opinião. Rodrigo Spínola - Editor da SQL Magazine editor@sqlmagazine.com.br

Fale com a redação



Stored procedures no MySQL – Parte 1

Ian Gilfillan

O MySQL finalmente implementou as funcionalidades de stored procedures em sua versão 5.0. Mas o que são exatamente stored procedures? Este é o tipo de pergunta que faz com que profissionais de banco de dados que usam outros SGBDs levantem suas sobrancelhas. Stored procedures foram integrados ao Oracle, PostgreSQL, DB-2, SQL Server, entre outros, já faz anos, e por muito tempo, foi doloroso não tê-las para o MySQL.

Uma stored procedure é um procedimento simples que é armazenado no servidor de banco de dados. Desenvolvedores do MySQL tiveram, desde tempos remotos, que escrever e armazenar estes procedimentos no servidor de aplicação (ou web), simplesmente porque não tinham outra opção. Isto era limitativo. Alguns diziam que existem duas escolas de pensamento: a primeira estipulava que a lógica do negócio deveria residir na aplicação, a segunda que deveria residir no banco de dados. Porém, a maioria dos profissionais não adotou sempre o primeiro ou o segundo ponto de vista. Como sempre, existem situações em que os dois pontos de vista fazem sentido. Infelizmente, alguns dos partidários mais fortes favoráveis ao primeiro ponto de vista (a lógica deve residir na aplicação), só o adotaram porque até o presente momento não tinham nenhuma outra escolha, e era a forma com a qual estavam acostumados. Então, por que desejaríamos colocar lógica no servidor de banco de dados?

Por que utilizar stored procedures?

Porque rodarão em todos os ambientes e sem necessidade de recriar a lógica para cada aplicação desenvolvida. Desde o momento em que estão no servidor de banco de dados, independentemente do ambiente de aplicação usado, a stored procedure permanece consistente. Mesmo que a sua configuração envolva clientes diferentes e linguagens de programação diferentes, a lógica permanece em um só

lugar. Desenvolvedores web normalmente fazem menos uso desta característica, pois o servidor web e o servidor de banco de dados geralmente são estreitamente ligados. Porém, em organizações cliente-servidor complexas, esta é uma grande vantagem. Os clientes estarão sempre automaticamente em sincronismo com a lógica de procedimento assim que a mesma tenha sido atualizada.

Podem também reduzir o tráfego da rede. Se parte da lógica da aplicação for feita no servidor de banco de dados, não haverá necessidade de enviar result sets e novas consultas do servidor da aplicação para o servidor do banco de dados. O tráfego de rede é um gargalo freqüente, que causa problemas de desempenho, e as stored procedures podem ajudar a reduzir este problema. Entretanto, freqüentemente acontece que o gargalo seja provocado pelo próprio servidor de banco de dados, portanto isto pode não ser uma grande vantagem. Caberá ao desenvolvedor, em conjunto com o DBA, identificar cada caso.

Um exemplo simples

Uma stored procedure é, simplesmente, um conjunto de declarações SQL. Quase qualquer SQL válido pode ser usado em uma stored procedure (com algumas exceções que veremos posteriormente). Montemos agora uma stored procedure básica. A **Listagem 1** simplesmente dirá 'Oi' na linguagem.

Listagem 1. Uma stored procedure simples.

```
mysql> CREATE PROCEDURE oi() SELECT 'oi';
Query OK, 0 rows affected (0.00 sec)

mysql> CALL oi()\G
***** 1. row *****

oi: oi
1 row in set (0.00 sec)
```

Certamente essa stored procedure não é de grande utilidade. O importante aqui é entender que CREATE PROCEDURE sp_name() define o procedimento e CALL sp_name() o chama para execução.

Parâmetros

O real benefício de uma stored procedure é obtido, é claro, quando você pode passar valores para ela e receber algum resultado. O conceito de parâmetros deverá ser familiar para qualquer um que tenha experiência com programação procedural. Existem três tipos de parâmetro:

- IN: o padrão. Este é um parâmetro de entrada para o procedimento. Pode ser alterado dentro da procedure, mas permanecerá inalterado fora dele;
- OUT: não é um parâmetro de entrada para o procedimento (será assumido NULL como valor de entrada), mas poderá ser modificado dentro da procedure e estará disponível fora dela após a alteração;
- INOUT: tem as características de ambos os parâmetros IN e OUT. Um valor pode ser passado ao procedimento, modificado no mesmo e também retornado.

O domínio de stored procedures requer conhecimentos de variáveis de sessão. A maioria de nós provavelmente já sabe usar variáveis de sessão, mas caso não saiba, o conceito é simples. Você pode designar um valor para uma variável, e recuperá-lo depois. A **Listagem 2** apresenta um exemplo em que configuramos a variável x para que a palavra “oi” possa ser repassada para um grupo de pessoas.

Na **Listagem 3**, temos um exemplo de uma stored procedure que mostra o uso de um parâmetro IN. Considerando que IN é o padrão, não há necessidade de especificá-lo.

A variável de sessão @x é configurada dentro do procedimento, com base no parâmetro P, que é passado ao procedimento e permanece inalterado.

Veja agora, na **Listagem 4**, um exemplo utilizando um parâmetro do tipo OUT. Note que foi necessário especificar que a variável é do tipo OUT, uma vez que este não é padrão.

Perceba que reinicializamos a variável @x somente para ter certeza de que o resultado final não seja um legado do procedimento anterior. Desta vez, o parâmetro P é alterado dentro do procedimento, enquanto que a variável de sessão é passada ao procedimento, pronta para receber o resultado.

Para finalizar, a **Listagem 5** apresenta um exemplo que utiliza o parâmetro do tipo INOUT. Neste caso também é necessário especificar o tipo de parâmetro que será utilizado.

Neste exemplo, um parâmetro é passado ao procedimento, usado no cálculo e o resultado é disponibilizado para a variável de sessão @x.

Obtendo informações sobre stored procedures existentes

É interessante que possamos obter mais informações

sobre qualquer stored procedure existente no banco de dados, tal como uma lista de procedimentos disponíveis, e as suas definições. Existem meios específicos para se conseguir isto no MySQL, e a sintaxe já deverá ser familiar para usuários experientes do MySQL.

SHOW PROCEDURE STATUS retorna uma lista de procedimentos armazenados e alguns metadados relacionados. Já SHOW CREATE PROCEDURE retorna a definição de um procedimento em particular (ver **Listagens 6 e 7**).

Listagem 2. Exemplo de utilização de variável.

```
mysql> SET @x='oi';
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @x\G
***** 1. row *****
@x: oi
1 row in set (0.00 sec)
```

Listagem 3. Exemplo de utilização de parâmetro do tipo IN.

```
mysql> CREATE PROCEDURE sp_in(p VARCHAR(10)) SET @x = p;
Query OK, 0 rows affected (0.00 sec)
mysql> CALL sp_in('oi');
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @x\G
***** 1. row *****
@x: oi
1 row in set (0.00 sec)
```

Listagem 4. Exemplo de utilização de parâmetro do tipo OUT.

```
mysql> SET @x='oi';
Query OK, 0 rows affected (0.00 sec)
mysql> CREATE PROCEDURE sp_out(OUT p VARCHAR(10)) SET P='ola';
Query OK, 0 rows affected (0.00 sec)
mysql> CALL sp_out(@x);
Query OK, 0 rows affected (0.00 sec)
```

Listagem 5. Exemplo de utilização de parâmetro do tipo INOUT.

```
mysql> SELECT @x\G
***** 1. row *****
@x: ola
1 row in set (0.00 sec)
mysql> CREATE PROCEDURE sp_inout(INOUT P INT) SET @x=P*2;
Query OK, 0 rows affected (0.00 sec)
mysql> CALL sp_inout(2);
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @x\G
***** 1. row *****
@x: 4
1 row in set (0.00 sec)
```

Também existe um modo ANSI padrão de fazer isto, que será mais familiar para usuários de outros SGBDs (ver **Listagem 8**).

Listagem 6. Verificando quais stored procedures existem no banco de dados.

```
mysql> SHOW PROCEDURE STATUS\G
```

```
***** 1. row *****
```

```
Db: test  
Name: molo  
Type: PROCEDURE  
Definer: ian@localhost  
Modified: 2005-07-29 19:20:27  
Created: 2005-07-29 19:20:27
```

```
Security_type: DEFINER
```

```
Comment:
```

```
***** 2. row *****
```

```
Db: test  
Name: sp_in  
Type: PROCEDURE  
Definer: ian@localhost  
Modified: 2005-08-02 11:58:34  
Created: 2005-08-02 11:58:34
```

```
Security_type: DEFINER
```

```
Comment:
```

```
***** 3. row *****
```

```
Db: test  
Name: sp_inout  
Type: PROCEDURE  
Definer: ian@localhost  
Modified: 2005-08-02 12:16:18  
Created: 2005-08-02 12:16:18
```

```
Security_type: DEFINER
```

```
Comment:
```

```
***** 4. row *****
```

```
Db: test  
Name: sp_out  
Type: PROCEDURE  
Definer: ian@localhost  
Modified: 2005-08-02 12:01:56  
Created: 2005-08-02 12:01:56
```

```
Security_type: DEFINER
```

```
Comment:
```

```
4 rows in set (0.00 sec)
```

Listagem 7. Verificando o metadado de criação de uma stored procedure em particular.

```
mysql> SHOW CREATE PROCEDURE molo\G
```

```
***** 1. row *****
```

```
Procedure: molo  
sql_mode:  
Create Procedure: CREATE PROCEDURE 'test'.'molo'()  
SELECT 'Molo'  
1 row in set (0.00 sec)
```

Listagem 8. Verificando quais stored procedures existem no banco de dados utilizando o padrão ANSI.

```
mysql> SELECT * FROM INFORMATION_SCHEMA.ROUTINES\G
```

```
***** 1. row *****
```

```
SPECIFIC_NAME: molo
```

```
ROUTINE_CATALOG: NULL
```

```
ROUTINE_SCHEMA: test
```

```
ROUTINE_NAME: molo
```

```
ROUTINE_TYPE: PROCEDURE
```

```
DTD_IDENTIFIER:
```

```
ROUTINE_BODY: SQL
```

```
ROUTINE_DEFINITION: SELECT 'Molo'
```

```
EXTERNAL_NAME: NULL
```

```
EXTERNAL_LANGUAGE: NULL
```

```
PARAMETER_STYLE:
```

```
IS_DETERMINISTIC: NO
```

```
SQL_DATA_ACCESS: CONTAINS_SQL
```

```
SQL_PATH: NULL
```

```
SECURITY_TYPE: DEFINER
```

```
CREATED: 2005-07-29 19:20:27
```

```
LAST_ALTERED: 2005-07-29 19:20:27
```

```
SQL_MODE:
```

```
ROUTINE_COMMENT:
```

```
DEFINER: ian@localhost
```

```
***** 2. row *****
```

```
SPECIFIC_NAME: sp_in
```

```
ROUTINE_CATALOG: NULL
```

```
ROUTINE_SCHEMA: test
```

```
ROUTINE_NAME: sp_in
```

```
ROUTINE_TYPE: PROCEDURE
```

```
DTD_IDENTIFIER:
```

```
ROUTINE_BODY: SQL
```

```
ROUTINE_DEFINITION: SET @x = P
```

```
EXTERNAL_NAME: NULL
```

```
EXTERNAL_LANGUAGE: NULL
```

```
PARAMETER_STYLE:
```

```
IS_DETERMINISTIC: NO
```

```
SQL_DATA_ACCESS: CONTAINS_SQL
```

```
SQL_PATH: NULL
```

```
SECURITY_TYPE: DEFINER
```

```
CREATED: 2005-08-02 11:58:34
```

```
LAST_ALTERED: 2005-08-02 11:58:34
```

```
SQL_MODE:
```

```
ROUTINE_COMMENT:
```

```
DEFINER: ian@localhost
```

Vamos a partir de agora avançar com exemplos mais complexos. Primeiro, criaremos uma tabela de demonstração (**Listagem 9**)

Listagem 9. Criação de uma tabela.

```
mysql> CREATE table sp1 (id INT, txt VARCHAR(10),
```

```
PRIMARY KEY(id));
```

```
Query OK, 0 rows affected (0.11 sec)
```

Delimitadores e procedimentos com mais de um statement

As stored procedures não seriam tão úteis caso consistissem apenas de uma declaração. Os procedimentos analisados até aqui poderiam ter seus efeitos produzidos por uma única declaração SQL. Procedimentos úteis são mais do que isso.

Vocês poderiam estar imaginando as complicações decorrentes. Como poderíamos diferenciar declarações múltiplas dentro do procedimento e o fim do mesmo? Para isso temos que criar um delimitador diferente para finalizar a declaração CREATE PROCEDURE. Vejamos como fazer isto na **Listagem 10**.

Listagem 10. Criando um delimitador.

```
mysql> DELIMITER |
```

Note que não há ponto-e-vírgula após o símbolo ‘|’ (pipe) que será usado como o delimitador para nossos propósitos. Você tem que escolher um delimitador que não apareça no procedimento e o mesmo poderá ser representado por mais de um caractere. Veja um exemplo de uso do delimitador definido na **Listagem 10** na **Listagem 11**.

Listagem 11. Utilização do delimitador criado na **Listagem 10**.

```
mysql> CREATE PROCEDURE sp_ins (P VARCHAR(10))
      -> BEGIN
      ->   SET @x=CHAR_LENGTH(P);
      ->   SET @y = HEX(P);
      ->   INSERT INTO sp1(id,txt) VALUES(@x,@y);
      -> END|
```

Query OK, 0 rows affected (0.05 sec)

```
mysql> CALL sp_ins('ABC');
      -> |
Query OK, 1 row affected (0.00 sec)
```

```
mysql> DELIMITER ;
mysql> SELECT * FROM sp1\G
***** 1. row *****
id: 3
txt: 414243
1 row in set (0.00 sec)
```

Note o que aconteceu quando tentamos chamar o procedimento. O MySQL ainda estava usando o símbolo ‘|’ como o delimitador e não o ponto-e-vírgula. Por esta razão, a declaração não roda após o ponto-e-vírgula. Precisamos antes fechá-la com o símbolo ‘|’. Depois, restauramos o delimitador para o normal, e verificamos que os registros foram adicionados corretamente à tabela sp1.

Variáveis de procedimento

As stored procedures não fazem uso apenas das declarações SQL padrão. Você também pode usar o DECLARE para declarar variáveis que só existam dentro do procedimento, assim como designar valores com a declaração SET sem usar o símbolo ‘@’ exigido para variáveis de sessão (ver **Listagem 12**).

Listagem 12. Declarações na stored procedure.

```
mysql> DELIMITER |
mysql> CREATE PROCEDURE sp_declare (P INT)
      -> BEGIN
      ->   DECLARE x INT;
      ->   DECLARE y INT DEFAULT 10;
      ->   SET x = P*y;
      ->   INSERT INTO sp1(id,txt) VALUES(x,HEX('DEF'));
      -> END|
```

Query OK, 0 rows affected (0.00 sec)

```
mysql> DELIMITER ;
mysql> CALL sp_declare(4);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT * FROM sp1\G
```

```
***** 1. row *****
id: 3
txt: 414243
```

```
***** 2. row *****
id: 40
txt: 444546
```

2 rows in set (0.00 sec)

Perceba também que nesta stored procedure vemos como efetuar o insert de valores no banco de dados.

Populando variáveis a partir de uma tabela existente

Agora que já vimos como efetuar um INSERT dentro da stored procedure para inserir registros em uma tabela, vamos saber como obtemos valores de uma tabela existente. Veja o exemplo na **Listagem 13**. Perceba que neste caso fizemos uso do LIMIT 1 para limitar o resultado ao primeiro registro encontrado.

Características

Vamos agora analisar todas as características que você pode definir ao criar um stored procedure. A **Listagem 14** mostra um exemplo da declaração CREATE PROCEDURE utilizando todas as cláusulas de sua sintaxe.

Listagem 13. Recuperando informações de tabela através de stored procedure.

```
mysql> DELIMITER |
mysql> CREATE PROCEDURE sp_select ()
-> BEGIN
->   DECLARE x INT;
->   DECLARE y VARCHAR(10);
->   SELECT id,txt INTO x,y FROM sp1 LIMIT 1;
->   SELECT x,y;
-> END|
Query OK, 0 rows affected (0.00 sec)

mysql> DELIMITER ;
mysql> CALL sp_select()\G
***** 1. row *****
x: 3
y: 414243
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

Listagem 14. Criando uma stored procedure utilizando todas as cláusulas da sintaxe.

```
CREATE PROCEDURE sp_full()
LANGUAGE SQL
NOT DETERMINISTIC
MODIFIES SQL DATA
SQL SECURITY DEFINER
COMMENT 'Returns a random number'
SELECT RAND()
```

O LANGUAGE SQL simplesmente indica que a linguagem utilizada para escrever o procedimento é o SQL. O MySQL por enquanto não consegue usar procedimentos em qualquer outra linguagem, mas provavelmente poderá no futuro.

O NOT DETERMINISTIC quer dizer que o procedimento pode produzir resultados diferentes, dadas as mesmas entradas. Ou seja, o resultado da execução da procedure leva em consideração aspectos variáveis em seu cálculo interno independentemente dos valores de entrada. A alternativa a esta característica é o valor DETERMINISTIC, caso o procedimento sempre produza os mesmos resultados dadas as mesmas entradas. O valor padrão é NOT DETERMINISTIC.

O MODIFIES SQL DATA indica que os dados podem ser modificados pelo procedimento. As alternativas são CONTAINS SQL (a procedure não deve conter qualquer cláusula para manipulação de dados, todas as demais cláusulas SQL são permitidas. Isso evita que a procedure execute operações de leitura e escrita nos dados do banco), NO SQL ou READS SQL DATA. O padrão é CONTAINS

SQL. Vale ressaltar aqui um ponto muito importante: estas características são apenas boas práticas. O servidor não as utiliza para restringir qual tipo de cláusula de uma rotina pode ser executada.

O SQL SECURITY DEFINER indica que o MySQL deverá utilizar os privilégios do usuário que definiu o procedimento antes de executar o mesmo. Isso é interessante quando você deseja permitir que um dado usuário tenha acesso a dados de uma tabela de sistema e não quer que este acesso seja feito de qualquer forma. Neste caso, você pode, por exemplo, permitir que ele acesse uma dada tabela de sistema utilizando apenas uma stored procedure. Assim, embora ele não tenha necessariamente acesso à tabela do sistema, ele poderá visualizar suas informações. A alternativa é SQL SECURITY INVOKER, que faz com que o MySQL use os privilégios do usuário que chama o procedimento. O padrão é SQL SECURITY DEFINER.

O COMMENT é auto-explicativo e pode ser usado para descrever o procedimento.

Declarações SQL exclusivas de stored procedures

Um procedimento pode precisar usar condições ou iterações, e o SQL padrão não é suficiente para estes propósitos. Por isso, o padrão SQL inclui várias declarações que só acontecem dentro de stored procedures. Veremos condições e iterações. Os conceitos deverão ser familiares para qualquer um com experiência em programação.

Veremos inicialmente as condições.

IF THEN ELSE

A **Listagem 15** apresenta a sintaxe do desvio condicional (if then else).

Listagem 15. Sintaxe do desvio condicional em stored procedures.

```
IF condition
THEN statement/s
ELSE statement/s
END IF
```

A lógica é simples. Se a condição for verdadeira, então um conjunto de declarações será executado. Caso contrário (ELSE), será executado outro conjunto de declarações. Na **Listagem 16** temos um exemplo desta utilização.

Eu presumo que você consiga seguir facilmente a lógica. Os valores NULL complicam um pouco a coisa - argumentos contra o seu uso sustentam que os mesmos enfraquecem a lógica booleana padrão. Não obstante, você poderá se deparar com exemplos onde são largamente usados. Vejamos, na **Listagem 17**, o que acontece se passarmos um NULL para o procedimento recém criado.

Listagem 16. Exemplo de utilização do desvio condicional.

```
mysql> CREATE PROCEDURE sp_condition(IN var1 INT)
BEGIN
  IF (var1 > 10)
    THEN SELECT 'greater';
  ELSE SELECT 'less than or equal';
  END IF;
END|  
Query OK, 0 rows affected (0.06 sec)  
mysql> CALL sp_condition(5)\G  
***** 1. row *****  
less than or equal: less than or equal  
1 row in set (0.00 sec)  
Query OK, 0 rows affected (0.00 sec)  
  
mysql> CALL sp_condition(15)\G  
***** 1. row *****  
greater: greater  
1 row in set (0.00 sec)  
Query OK, 0 rows affected (0.00 sec)
```

Listagem 17. Tratamento de NULL no desvio condiciona.

```
mysql> CALL sp_condition(NULL)\G  
***** 1. row *****  
less than or equal: less than or equal  
1 row in set (0.00 sec)  
Query OK, 0 rows affected (0.00 sec)
```

NULL, por definição, representa a ausência de valor, portanto NULL > 10 é avaliado como falso, pois o interpretador não tem como “ter certeza” da sua validade e a declaração ELSE será rodada. Façamos a modificação apresentada na **Listagem 18**.

Listagem 18. Tratamento de NULL no desvio condiciona.

```
mysql> CREATE PROCEDURE sp_condition2(IN var1 INT)
BEGIN
  IF (var1 <= 10)
    THEN SELECT 'less than or equal';
  ELSE SELECT 'greater';
  END IF;
END|  
Query OK, 0 rows affected (0.00 sec)  
mysql> CALL sp_condition2(NULL)\G  
***** 1. row *****  
*****  
greater: greater  
1 row in set (0.00 sec)  
Query OK, 0 rows affected (0.00 sec)
```

Usando o que em lógica booleana correta constituiria um teste idêntico, obteremos um resultado discrepante. NULL <= 10 também será avaliado como falso, comprovando a “falta de certeza” quanto à validade da condição.

CASE

A outra construção usada com condições é o CASE. Veja a sintaxe na **Listagem 19**.

Listagem 19. Sintaxe do CASE em stored procedures.

```
CASE variable
WHEN condition1 statement/s
WHEN condition2 statement/s
ELSE statement/s
END CASE
```

Esta construção é usada quando a mesma variável está sendo testada com múltiplas condições. Em lugar do longo aninhamento de declarações IF, o uso da declaração CASE torna mais compacto e legível o código do procedimento. Veja um exemplo na **Listagem 20**.

Listagem 20. Exemplo de utilização da instrução CASE.

```
mysql> CREATE PROCEDURE sp_case(IN var1 INT)
BEGIN
  CASE var1
    WHEN 1 THEN SELECT 'One';
    WHEN 2 THEN SELECT 'Two';
    ELSE SELECT 'Something else';
  END CASE;
END|  
Query OK, 0 rows affected (0.00 sec)  
mysql> CALL sp_case(1)\G  
***** 1. row *****  
One: One  
1 row in set (0.00 sec)  
Query OK, 0 rows affected (0.00 sec)  
mysql> CALL sp_case(2)\G  
***** 1. row *****  
Two: Two  
1 row in set (0.00 sec)  
Query OK, 0 rows affected (0.00 sec)  
mysql> CALL sp_case(3)\G  
***** 1. row *****  
Something else: Something else  
1 row in set (0.00 sec)  
Query OK, 0 rows affected (0.00 sec)
```

Vejamos a partir de agora os exemplos das várias utilizações de iterações em stored procedures no MySQL.

WHILE

Iterações são componentes vitais dos procedimentos. Permite repetir o mesmo trecho de código várias vezes. Iterações WHILE repetem um bloco continuamente até que uma condição particular seja verdadeira. Veja a sintaxe na **Listagem 21**.

Listagem 21. Sintaxe da iteração do tipo WHILE em stored procedures.

```
WHILE condition DO
statement/s
END WHILE
```

A **Listagem 22** mostra um exemplo.

Listagem 22. Exemplo de utilização da iteração do tipo WHILE.

```
mysql> CREATE PROCEDURE sp_while(IN var1 INT)
BEGIN
  WHILE (var1 < 20) DO
    SELECT var1;
    SET var1=var1+1;
  END WHILE;
END|
Query OK, 0 rows affected (0.00 sec)
mysql> CALL sp_while(18)\G
***** 1. row *****
var1: 18
1 row in set (0.00 sec)
***** 1. row *****
var1: 19
1 row in set (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
mysql> CALL sp_while(22)\G
Query OK, 0 rows affected (0.00 sec)
```

Note que quando chamados o procedimento passando o valor 22 como parâmetro a iteração não é executada, pois a condição falhará de imediato.

REPEAT UNTIL

A outra iteração bastante utilizada é a construção REPEAT UNTIL. Veja sua sintaxe na **Listagem 23**.

Listagem 23. Sintaxe da iteração do tipo REPEAT UNTIL em stored procedures.

```
REPEAT
statement/s
UNTIL condition
END REPEAT
```

Com REPEAT UNTIL, as declarações serão executadas

repetidamente até que a condição seja encontrada. Uma diferença para a iteração WHILE é que a condição só será testada depois que as declarações forem executadas, portanto sempre será rodada pelo menos uma instância das declarações. É o que podemos conceituar como “estrutura de repetição com o teste lógico no início do loop” (WHILE) ou “estrutura de repetição com o teste lógico no final do loop” (REPEAT UNTIL). A **Listagem 24** apresenta um exemplo.

Listagem 24. Exemplo de utilização da iteração do tipo REPEAT UNTIL.

```
mysql> CREATE PROCEDURE sp_repeat(IN VAR1 INT)
BEGIN
  REPEAT
    SELECT var1;
    SET var1=var1+1;
  UNTIL var1>5
  END REPEAT;
END|
Query OK, 0 rows affected (0.09 sec)
mysql> CALL sp_repeat(3)\G
***** 1. row *****
var1: 3
1 row in set (0.00 sec)
***** 1. row *****
var1: 4
1 row in set (0.00 sec)
***** 1. row *****
var1: 5
1 row in set (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
mysql> CALL sp_repeat(8)\G
***** 1. row *****
var1: 8
1 row in set (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
```

Note que mesmo quando chamamos o procedimento com o valor 8 (isto implica que a condição será avaliada como verdadeira (e a iteração será interrompida) da primeira vez que for encontrada), a declaração SELECT ainda será executada uma vez, pois a condição somente é testada no final.

LABELs, LEAVEs e LOOPs

LABELs (rótulos) simplesmente são strings de texto usadas para marcar parte da procedure. Podem fazer simplesmente o papel de comentários, ou fazer parte da lógica, como veremos abaixo na construção LOOP.

Na **Listagem 25** temos um exemplo de LABELs usados para comentar o início de um procedimento. O LABEL é begin1. Isto poderá não parecer útil neste momento, mas declarações complexas podem ser aninhadas em profundidade,

e neste caso, LABELs claramente definidos tornarão mais fácil a análise da lógica. Eles também têm um papel lógico vital, como veremos mais adiante.

Listagem 25. Utilização de LABELs como comentários na stored procedure.

```
mysql> CREATE PROCEDURE sp_label()
begin1: BEGIN
END|
Query OK, 0 rows affected (0.00 sec)
mysql> CREATE PROCEDURE sp_label2()
begin1: BEGIN
END begin1|
Query OK, 0 rows affected (0.00 sec)
```

No segundo exemplo da **Listagem 25**, o fim da procedure também está rotulado. Se um LABEL de fim existir, tem que casar com um LABEL de início do mesmo nome. Você pode usar LABELS antes de declarações BEGIN, WHILE, REPEAT e LOOP (que veremos adiante), bem como com as respectivas declarações END, e também como alvos para declarações ITERATE (que também veremos adiante).

Um terceiro tipo de iteração é a construção LOOP. Ele não testa nenhuma condição nem no início nem no fim da declaração LOOP. Continuará iterando explicitamente até sair com uma declaração LEAVE, entrando facilmente em uma iteração infinita.

A declaração LEAVE encerra um bloco (que pode incluir o próprio procedimento). Uma vez que possam existir construções aninhadas, deve-se também ser acompanhado por um nome de LABEL para poder determinar de qual bloco sair (**Listagem 26**).

Listagem 26. Sintaxe da iteração do tipo LOOP e seu finalizador LEAVE em stored procedures.

```
label LOOP
statement/s
LEAVE label
statement/s
END LOOP
```

Veja agora um exemplo na **Listagem 27**.

O nome de LABEL que usamos foi loop1, e a declaração LEAVE explicitamente interrompeu a iteração. Não usamos um LABEL de fim.

Declarações ITERATE

O ITERATE somente pode aparecer dentro de declarações LOOP, REPEAT e WHILE. É seguido de um nome de LABEL, e efetivamente direciona o controle de volta para aquele LABEL. Portanto, se aparecem no meio de um LOOP, e direciona o controle de volta para o topo do

Listagem 27. Utilização do LOOP em stored procedure.

```
mysql> CREATE PROCEDURE sp_loop(IN var1 INT)
BEGIN
loop1: LOOP
    IF (var1 > 5) THEN
        LEAVE loop1;
    END IF;
    SET var1=var1+1;
    SELECT var1;
END LOOP;
END |
Query OK, 0 rows affected (0.00 sec)
mysql> CALL sp_loop(2)\G
***** 1. row *****
var1: 3
1 row in set (0.00 sec)
***** 1. row *****
var1: 4
1 row in set (0.00 sec)
***** 1. row *****
var1: 5
1 row in set (0.00 sec)
***** 1. row *****
var1: 6
1 row in set (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
```

LOOP, o resto da iteração não será executado (tenha em mente esta lógica ao testar, pois abre a possibilidade de iterações infinitas). Podemos comparar seu comportamento com os famosos GOTO para os que estão acostumados com linguagens de programação.

No exemplo da **Listagem 28** usaremos uma iteração WHILE, e iteraremos de volta para o início da iteração caso var1 ainda seja menor que 3.

Na primeira vez em que o LOOP é percorrido, somente “starting the loop” (iniciando a iteração) será exibido, pois var1 tem o valor 2. Quando for avaliada a condição do IF, a condição será avaliada como verdadeira.

O ITERATE retorna o controle novamente para o início de LOOP. A segunda vez em que o LOOP é percorrido, var1 começa com o valor 2, mas é incrementado em seguida para 3 e a condição IF será falsa. A iteração completa será executada.

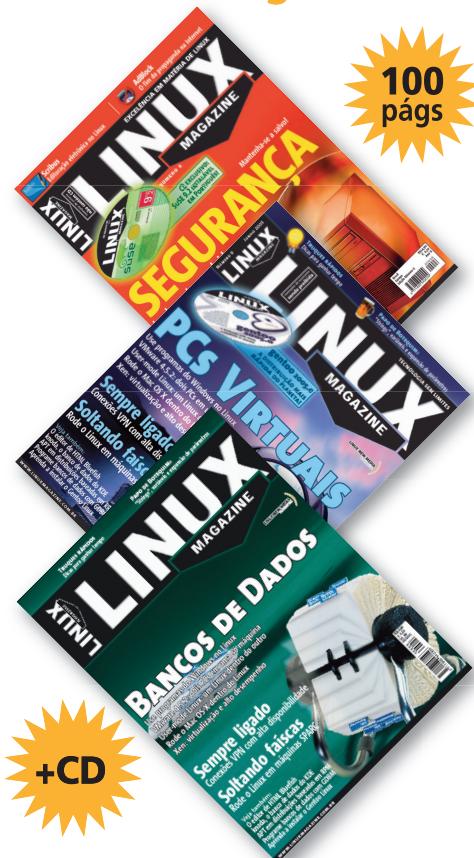
Conclusão

Espero que você tenha se sentido confortável com a sintaxe e possa criar suas próprias stored procedures. O MySQL 5.0 veio satisfazer a todos aqueles que sentiam falta desta poderosa ferramenta para armazenar a “regra de negócio” no próprio banco de dados. A partir de agora, não há mais motivos para “reclamações”.

TECNOLOGIA SEM LIMITES



Análises e comparações de ferramentas de bancos de dados, servidores, técnicas de administração e segurança de redes, todos os meses, só na revista Linux Magazine.



SGBDs Livres

Listagem 28. Utilização do ITERATE em stored procedure.

```
mysql> CREATE PROCEDURE sp_while_iterate(IN var1 INT)
-> BEGIN
->   while_label: WHILE (var1 < 5) DO
->     SELECT CONCAT('starting the loop: var1 is: ',var1);
->     SET var1=var1+1;
->     IF (var1<3) THEN
->       ITERATE while_label;
->     END IF;
->     SELECT CONCAT('ending the loop: var1 is: ',var1);
->   END WHILE;
-> END|
Query OK, 0 rows affected (0.00 sec)
mysql> CALL sp_while_iterate(1)\G
***** 1. row *****
CONCAT('starting the loop: var1 is: ',var1):
      starting the loop: var1 is: 1
1 row in set (0.00 sec)
***** 1. row *****
CONCAT('starting the loop: var1 is: ',var1):
      starting the loop: var1 is: 2
1 row in set (0.00 sec)
***** 1. row *****
CONCAT('ending the loop: var1 is: ',var1):
      ending the loop: var1 is: 3
1 row in set (0.00 sec)
***** 1. row *****
CONCAT('starting the loop: var1 is: ',var1):
      starting the loop: var1 is: 3
1 row in set (0.00 sec)
***** 1. row *****
CONCAT('ending the loop: var1 is: ',var1):
      ending the loop: var1 is: 4
1 row in set (0.00 sec)
***** 1. row *****
CONCAT('starting the loop: var1 is: ',var1):
      starting the loop: var1 is: 4
1 row in set (0.00 sec)
***** 1. row *****
CONCAT('ending the loop: var1 is: ',var1):
      ending the loop: var1 is: 5
1 row in set (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
```

No próximo artigo daremos continuidade à discussão sobre stored procedures no MySQL. Até lá.



Ian Gilfillan

é especialista em MySQL.

Assine agora mesmo!
0800 702 54 01

easyLINUX linuxUSER LINUXPARK

LINUX NEW MEDIA
The Pulse of Linux

www.linuxnewmedia.com.br
info@linuxnewmedia.com.br
SQL Magazine . 25