Guia de Programação e Melhores Práticas para MS-SQL Server 2000

Tópicos nesta página

Introdução

O objetivo deste manual é que ele seja utilizado como referência para programação no Microsoft SQL Server. Nele você vai encontrar um guia para programação e utilização das melhores práticas para banco de dados SQL Server.

Este guia foi desenvolvido com base na documentação fornecida pela Microsoft e nos problemas de desempenho encontrados nos diversos sistemas que funcionam em diferentes empresas do mercado. Todos os itens listados aqui, visam uma otimização na utilização do uso dos recursos do SQL Server buscando sempre o melhor desempenho. É importante ressaltar, que existem situações excepcionais em que você pode decidir por não utilizar algum dos itens aqui listados, seja por motivos de regras de negócio ou por decisões estratégicas. O importante é que você tenha ciência de que regra não está sendo seguida e o porquê disso, estando atento a todas as implicações desta decisão.

Chaves primárias (PKs) e índices cluster

Chaves primárias (PKs) permitem identificar um registro em uma tabela de modo único e podem ser clustered ou non-clustered. Mais do que isso, elas garantem o desempenho das suas consultas e a integridade dos dados. Tenha certeza de todas as tabelas do seu banco de dados possuem chave primária. Caso não seja possível a criação de uma chave primária, então um índice cluster deve ser criado.

Comentários

Procure incluir comentários em suas stored procedures, triggers e blocos de comandos SQL sempre que o que estiver programando não for muito obvio. Isto ajudará outros programadores a entender seu código facilmente. Não se preocupe com o tamanho dos comentários pois eles não causam impacto em performance, diferentemente de linguagens interpretadas como ASP 2.0

Consultas simples

Não utilize SELECT * em suas consultas. Sempre escreva o nome das colunas necessárias após o comando SELECT. Exemplo:

```
SELECT CustomerID, CustomerFirstName, City
```

Esta técnica diminui o I/O do disco e melhora o desempenho.

Cursores

Procure não utilizar cursores no servidor sempre que possível. Em alguns casos, cursores podem ser substituídos por comandos SELECT e variáveis do tipo TABLE. Se o cursor é extremamente necessário, tente utilizar um loop com WHILE ao invés do cursor. A execução de um loop com WHILE é sempre mais rápida do que a de um cursor. O único detalhe é que você vai precisar de uma coluna (Primary Key ou Unique Key) para identificar cada linha de modo único.

Tabelas Temporárias x Variáveis do tipo Table: principais vantagens e desvantagens

Dadas as principais vantagens e desvantagens do uso de tabelas temporárias e de variáveis do tipo table listadas a seguir, as variáveis do tipo table devem ser utilizadas sempre que possível. Tabelas temporárias devem ser utilizadas somente em processos que precisem manipular grandes volumes de dados precisando de desempenho e sem a preocupação da concorrência com outros sistemas.

Tabelas temporárias

- Principais vantagens
    - Apresenta melhor desempenho para grandes volumes de dados
    - Permite a troca de informações entre processos e aplicações diferentes (tabelas globais)
- Principais desvantagens
    - Elevado acesso a disco
    - Uso do tempdb
    - Baixa Concorrência

Variáveis do tipo table

- Principais vantagens
    - Toda informação é manipulada em memória
    - Apresenta melhor desempenho para pequenos volumes de dados

    - Concorrência elevada (o processamento de um sistema tem impacto muito reduzido em outro concorrente)
- Principais desvantagens
    - Baixo desempenho com grandes volumes de dados

O exemplo a seguir, ilustra a utilização de variáveis do tipo table.

```
declare @tbAuthors TABLE (
                  au_id  char(11),
                  au_lname varchar(40),
                  au_fname varchar(20),
                  phone  char(12),
                  address varchar(40),
                  city  varchar(20),
                  state char(2),
                  zip char(5),
                  contract bit
                  )

insert @tbAuthors
select *
from pubs..authors where state = 'UT'

select *
from @tbAuthors

update @tbAuthors
set state = 'NY'
where state = 'UT'

select *
from @tbAuthors
```

Operadores curinga e de negação

Procure evitar caracteres "curinga" no início de uma palavra quando realizar uma busca utilizando o operador LIKE, pois isto resulta em um index scan, o que subutiliza o índice criado. O primeiro dos exemplos a seguir resulta em um index scan, enquanto o segundo resulta em um index seek.

```
SELECT au_id FROM authors WHERE au_lname LIKE '%inger'
SELECT au_id FROM authors WHERE au_lname LIKE 'R%r'
```

Evite também realizar buscas utilizando operadores de negação (<> e NOT) já que eles resultam em table e index scan.

Tabelas derivadas

Utilize as tabelas derivadas sempre que possível, pois elas apresentam um melhor desempenho. Veja, por exemplo, a consulta a seguir utilizada para exibir o segundo livro mais caro da tabela titles.

```
SELECT MIN(price)
FROM dbo.titles
WHERE title_id IN
(
```

```
SELECT TOP 2 title_id
FROM dbo.titles
ORDER BY price DESC
)
```

A mesma consulta pode ser escrita utilizando uma tabela derivada, como no exemplo a seguir que é executado bem mais rapidamente do que o exemplo anterior

```
SELECT MIN(price)
FROM
(
SELECT TOP 2 price
FROM dbo.titles
ORDER BY price DESC
) AS A
```

Obs.: Este é apenas um exemplo e seus resultados podem ser outros em cenários diferentes dependendo do desenho do banco, dos índices, do volume de dados, etc. Por isso, teste seu código de todos os modos possíveis a fim de que ele seja escrito do modo mais eficiente.

Desempenho

Quando estiver criando os objetos do seu banco de dados, tenha sempre em mente o desempenho. Em algumas situações não é possível resolver totalmente problemas de desempenho quando o banco de dados já está em produção uma vez que pode ser necessárias a redefinição de tabelas, índices, e a alteração de consultas.

Utilize o EXECUTION PLAN gráfico do Query Analyser ou os comandos SHOWPLAN_TEXT e SHOWPLAN_ALL para analisar suas consultas. Se certifique de que as consultas executam um "Index Seek" ao invés de um "Index Scan" ou "Table Scan". Um index scan ou um table scan degradam demais o desempenho e devem ser evitados sempre que possível.

Referenciando tabelas

Sempre que referenciar uma tabela, utilize o nome do proprietário da tabela como prefixo do objeto (ex.: dbo.authors) uma vez que isso proporciona um acesso mais rápido a tabela e evita confusões. De acordo com o Microsoft SQL Server Books Online, referenciar nomes de tabelas com o nome do seu proprietário, auxilia na reutilização do plano de execução e melhora a performance.

SET NOCOUNT

Utilize SET NOCOUNT ON no inicio dos seus blocos de comandos SQL das suas procedures e triggers em ambiente de produção, pois desta forma você estará evitando mensagens do tipo '(1 row affected)' que são exibidas após a execução de comandos INSERT, UPDATE e DELETE. Isto melhora o desempenho diminuindo o tráfego na rede.

Joins no padrão ANSI

Faça uso da maior flexibilidade dos joins padrão ANSI ao invés de utilizar os joins antigos. Com joins padrão ANSI, a cláusula WHERE é utilizada somente para filtro de dados, enquanto nos joins antigos, a cláusula WHERE é utilizada tanto para condições de join quanto para filtro de dados. Nos exemplos a seguir, o primeiro utiliza a sintaxe de join antigo e o segundo utiliza a sintaxe do novo join padrão ANSI

```
SELECT a.au_id, t.title
FROM dbo.titles t, dbo.authors a, dbo.titleauthor ta
WHERE
```

```
a.au_id = ta.au_id AND
ta.title_id = t.title_id AND
t.title LIKE '%Computer%'

SELECT a.au_id, t.title
FROM dbo.authors a
INNER JOIN
dbo.titleauthor ta
ON
a.au_id = ta.au_id
INNER JOIN
dbo.titles t
ON
ta.title_id = t.title_id
WHERE t.title LIKE '%Computer%'
```

Sub-queries

O query optimizer do SQL 2000 realiza uma conversão de qualquer Sub-Query primeiro para Joins, para só a partir daí , gerar o plano de execução da consulta. Você pode otimizar esse processo escrevendo as rotinas críticas diretamente como Joins. O exemplo a seguir, ilustra o problema.
Sub-Query:

```
SELECT dbo.titles.title, dbo.titles.price
FROM  dbo.titles
WHERE dbo.titles.title_id IN (      SELECT dbo.sales.title_id
                                    FROM dbo.sales
                                    WHERE dbo.sales.qty > 20)
```

Join equivalente a Sub-Query:

```
SELECT dbo.titles.title, dbo.titles.price
FROM   dbo.sales
INNER JOIN
dbo.titles
ON
dbo.sales.title_id = dbo.titles.title_id
WHERE (dbo.sales.qty > 20)
```

Nomes para Stored Procedures

Não utilize como prefixo para nomes de suas stored procedures o "sp_". O prefixo sp_ é reservado para stored procedures de sistema que são fornecidas com o SQL Server. Sempre que o SQL Server encontra um nome de procedure que comece com sp_, ele primeiro tenta localizar a procedure no banco de dados master, depois procura pelos qualificadores (banco de dados, proprietário) fornecidos, em seguida ele tenta utilizar o dbo como proprietário. Portanto, você realmente pode ganhar muito tempo na localização da stored procedure simplesmente não utilizando o "sp_" como prefixo.

Views

As views normalmente são utilizadas para exibir dados específicos para usuários específicos de acordo com seus interesses. As views também são utilizadas para restringir acesso a tabelas essenciais do sistema uma vez que o usuário só precisa ter acesso a view e não a tabela. Existe ainda uma outra utilização interessante para o uso de views que é a simplificação das suas consultas. Incorpore comandos que você utiliza com freqüência, como joins complicados, e cálculos em uma view de modo que você não terá que repetir estes joins e cálculos em todas as suas consultas. Você só precisará consultá-los na

view.

User Defined Datatypes

Utilize User Defined Datatypes se uma particular coluna se repete com muita freqüência em suas tabelas de modo que o tipo de dado desta coluna fique consistente em todas as suas tabelas.

Tipo de dados TEXT

Procure não utilizar tipos de dados TEXT ou NTEXT para o armazenamento de textos não muito extensos. O tipo de dados TEXT possui alguns problemas inerentes associados a ele. Você, por exemplo, não pode escrever ou atualizar o texto utilizando os comandos INSERT ou UPDATE. Para isso, você precisa utilizar os comandos READTEXT, WRITETEXT e UPDATETEXT. Existem ainda umas séries de problemas associados à replicação de dados contendo colunas do tipo TEXT. Portanto, se você não precisa armazenar mais do que 8KB de texto, utilize os tipos de dados CHAR(8000) ou VARCHAR (8000).

Tipo de dados CHAR

Utilize o tipo de dados CHAR para uma coluna somente se a coluna não permite valores nulos. Se uma coluna do tipo CHAR permite valores nulos, ela é tratada como uma coluna de tamanho fixo a partir do SQL 7. Portanto, uma coluna CHAR(100), quando possuir um valor nulo, utilizará 100 bytes, resultando em perda de espaço no servidor. Utilize o tipo de dados VARCHAR(100) em situações como esta. Obviamente, colunas com tipo de dados de tamanho variável, tem um pouco mais de overhead de processamento em relação a colunas com tamanho fixo. Lembre-se disso sempre que estiver definindo colunas com estes tipos de dados.

Comandos dinâmicos SQL

Evite comandos dinâmicos SQL sempre que possível. Comandos dinâmicos SQL tendem a ser mais lento do que comandos estáticos, já que o SQL precisa gerar o plano de execução sempre que o comando é executado. Comandos IF e CASE são boas alternativas para se evitar comandos SQL dinâmicos. Outra grande desvantagem do uso de comandos SQL dinâmicos, é que você precisa que os usuários tenham permissões de acesso direto a todos os objetos, como por exemplo tabelas e views. Normalmente os usuários têm acesso a procedures que alteram tabelas sem precisar o acesso direto às tabelas. Nesta situação, comandos dinâmicos SQL não funcionarão. O exemplo a seguir, ilustra o problema.

```
sp_addlogin 'dSQLuser'
GO
sp_defaultdb 'dSQLuser', 'pubs'
GO
USE pubs
GO
sp_adduser 'dSQLUser', 'dSQLUser'
GO
CREATE PROC dbo.dSQLProc
AS
BEGIN
SELECT * FROM dbo.titles WHERE title_id = 'BU1032' --Funciona
DECLARE @str CHAR(100)
SET @str = 'SELECT * FROM titles WHERE title_id = ''BU1032'''
EXEC (@str) --Este comando apresenta o erro
END
GO
GRANT EXEC ON dbo.dSQLProc TO dSQLuser
```

```
GO
```

Mensagem de erro exibida com o uso do comando dinâmico:

```
Server: Msg 229, Level 14, State 5, Line 1
SELECT permission denied on object 'titles', database 'pubs', owner 'dbo'.
```

Uma outra alternativa a comandos dinâmicos SQL quando os parâmetros da cláusula WHERE precisam ser montados dinamicamente, é a utilização de uma procedure que contenha todos os parâmetros (com default NULL) e a função ISNULL na cláusula WHERE fornecendo somente os parâmetros desejados. Nos exemplos a seguir, o primeiro utiliza comandos dinâmicos SQL e o segundo, faz uso do ISNULL.

```
CREATE PROCEDURE dbo.dinamicSQL(
@SQL VARCHAR(8000)
)
AS
EXEC (@SQL)
GO
```

O segundo comando é executado com um desempenho muito melhor.

```
CREATE PROCEDURE dbo.staticSQL(
@state char(2) = NULL,
@au_fname varchar(40) = NULL,
@type char(12) = NULL
)
AS
SELECT dbo.authors.au_fname, dbo.titles.title, dbo.titles.type,
dbo.authors.city, dbo.authors.state
FROM dbo.authors INNER JOIN dbo.titleauthor
ON dbo.authors.au_id = dbo.titleauthor.au_id INNER JOIN dbo.titles
ON dbo.titleauthor.title_id = dbo.titles.title_id
WHERE dbo.authors.state = ISNULL(@state, dbo.authors.state)
AND dbo.authors.au_fname = ISNULL(@au_fname,dbo.authors.au_fname)
AND dbo.titles.type = ISNULL(@type,dbo.titles.type)
GO
```

Modos de execução possíveis:

```
EXEC dbo.staticSQL
EXEC dbo.staticSQL @type = 'business'
EXEC dbo.staticSQL @state = 'CA', @au_fname = 'Marjorie'
EXEC dbo.staticSQL @state = 'CA', @au_fname = 'Michael', @type = 'trad_cook'
```

Valores nulos

Minimize o uso de nulos, uma vez que eles confundem o front-end das aplicações, a menos que as aplicações estejam codificadas de modo inteligente para eliminarem os nulos ou convertê-los em alguma outra forma. Qualquer expressão que manipule valores nulos pode gerar um resultado nulo. As funções ISNULL e COALESCE podem ajudar na manipulação de valores nulos. O exemplo abaixo explica o problema.

Considere a tabela de clientes (customers) abaixo, que armazena os nomes dos clientes e o campo middle name pode ser nulo.

```
CREATE TABLE dbo.Customers
(
FirstName varchar(20) null,
MiddleName varchar(20) null,
```

```
LastName varchar(20) null
)
```

Ao inserir um cliente de nome Bill Gates, sem o nome do meio, teremos:

```
INSERT INTO dbo.Customers
(FirstName, MiddleName, LastName)
VALUES ('Bill',NULL,'Gates')
```

O comando SELECT a seguir, retorna nulo, ao invés do nome do cliente

```
SELECT FirstName + ' ' + MiddleName + ' ' + LastName
FROM dbo.Customers
```

Para evitar esse problema, utilize a função ISNULL

```
SELECT FirstName + ' ' + ISNULL(MiddleName + ' ','') + LastName
FROM dbo.Customers
```

Tipo de dados Unicode

Utilize tipo de dados Unicode, como por exemplo NCHAR, NVARCHAR, ou NTEXT, se seu banco de dados não armazena somente caracteres americanos. Utilize estes tipos de dados somente se for extremamente necessário, pois eles utilizam o dobro do espaço utilizado para armazenar tipos de dados não-unicode. O exemplo a seguir ilustra a utilização de tipo de dados unicode.

```
DECLARE @nstring nchar(8)
SET @nstring = N'København'
SELECT UNICODE(SUBSTRING(@nstring, 2, 1)),
   NCHAR(UNICODE(SUBSTRING(@nstring, 2, 1)))
GO
```

Resultado:

```
----------- -
248         ø
```

INSERT

Sempre utilize uma lista de colunas em seus comandos INSERT. Isto ajuda a evitar problemas quando a estrutura da tabela é alterada (no caso da inclusão ou exclusão de colunas). A seguir, um exemplo que ilustra o problema.

```
CREATE TABLE EuropeanCountries
(
CountryID int PRIMARY KEY,
CountryName varchar(25)
)
```

A seguir, o exemplo de comando SELECT que funciona sem problemas:

```
INSERT INTO EuropeanCountries
VALUES (1, 'Ireland')
```

Ao adicionarmos uma coluna a tabela:

```
ALTER TABLE EuropeanCountries
ADD EuroSupport bit
```

Agora, se você executar o INSERT anterior, obterá a mensagem:

```
Server: Msg 213, Level 16, State 4, Line 1
Insert Error: Column name or number of supplied values does not match table
definition.
```

Este problema pode ser evitado especificando-se a lista de colunas no comando INSERT.

```
INSERT INTO EuropeanCountries
(CountryID, CountryName)
VALUES (1, 'England')
```

Constraints

Realize toda a verificação de integridade referencial e validação de dados, utilizando constraints (foreign key e check) ao invés de utilizar triggers, pois as constraints são mais rápidas. Restrinja o uso de triggers somente para auditoria, tarefas específicas e verificações que não podem ser realizadas utilizando-se constraints. As constraints também economizam tempo, já que você não precisa escrever código para estas validações, permitindo que o banco de dados faça todo o trabalho para você.

DeadLocks

Sempre acesse tabelas na mesma ordem em todas as suas stored procedures e triggers de modo consistente. Isto ajuda a evitar DeadLocks. Outras coisas que se deve ter em mente para evitar DeadLocks são as seguintes.

Manter suas transações pelo menor espaço de tempo necessário. Alterar a menor quantidade de dados possível dentro de uma transação.

Nunca, jamais, esperar por uma entrada de dados do usuário no meio de uma transação.

Não utilizar hints de lock ou restringir níveis de isolação de dados, a menos que sejam absolutamente necessários.

Faça o front-end ou o middle tie da sua aplicação inteligente a ocorrência de deadlocks de modo que a aplicação possa re-enviar toda a transação no caso da ocorrência de um DeadLock (erro 1205).

Em suas aplicações, processe todos os resultados retornados pelo SQL Server imediatamente de modo que os locks nas linhas processadas sejam liberados.

Depuração de erros

Sempre inclua um parâmetro @Debug nas suas stored procedures. Ele pode ser do tipo de dados BIT. Quando o valor 1 for especificado para este parâmetro, imprima todos os resultados intermediários e conteúdo de variáveis utilizando os comandos SELECT ou PRINT. Quando o valor 0 for especificado, processe seus comandos normalmente. Isto pode te ajudar bastante a resolver problemas relativos a comportamentos inesperados de procedures, uma vez que você não precisa adicionar ou remover os comandos SELECT/PRINT sempre que estiver identificando um problema.

Chamadas repetidas a funções

Não chame funções repetidamente em suas stored procedures, triggers, functions e blocos de comandos SQL. Por exemplo, você pode precisar calcular o tamanho de uma string em vários lugares em sua procedure. Não faça a chamada da função LEN mais de uma vez para cada string. Chame a função somente uma vez e armazene o valor em uma variável para utilizar sempre que preciso.

Retorno de Stored Procedures

Certifique-se de que suas stored procedures sempre retornam um valor indicado o status da execução. Padronize os valores de retorno das procedures de modo que indiquem o sucesso ou a falha. O comando RETURN existe para a indicação do status de execução somente, não dados. Para o retorno de dados, utilize parâmetros do tipo OUTPUT.

Parâmetros do tipo OUTPUT

Se sua stored procedure sempre retorna uma linha como resultado, altere esta procedure de modo que os dados sejam retornados com parâmetros do tipo OUTPUT, ao invés de utilizar comandos SELECT, já que as interfaces de objetos de dados (ADO, RDO, etc) manipulam parâmetros do tipo OUTPUT de modo mais rápido do que os resultados de comandos SELECT.

@@ERROR, XACT_ABORT e @@ROWCOUNT

Sempre verifique a variável global @@ERROR imediatamente após a execução de comandos de manipulação de dados (INSERT/UPDATE/DELETE), de modo que você possa cancelar sua transação caso ocorra um erro (o valor da variável @@ERROR será maior do que 0 caso ocorra um erro). Isto é muito importante, pois o SQL Server, a principio, não irá cancelar todas as alterações realizadas anteriormente dentro da transação no caso da falha de um dos comandos. Este comportamento pode ser alterado com a execução do comando:

```
SET XACT_ABORT ON
```

A variável @@ROWCOUNT também tem um papel importante neste contexto, pois determina a quantidade de linhas afetadas pelo ultimo comando de manipulação de dados e com base nesta informação, você pode decidir entre confirmar ou cancelar uma determinada transação.

Legibilidade

Escreva seus comandos SQL de forma mais legível possível. Comece cada cláusula em uma nova linha e endente sempre que necessário. A seguir, um exemplo.

```
SELECT title_id, title
FROM   dbo.titles
WHERE  title LIKE '%Computer%' AND
       title LIKE '%cook%'
```

Bug do milênio

Lembre-se de que nós sobrevivemos ao ano 2000 e sempre armazene o ano nas datas com 4 dígitos (especialmente se você estiver utilizando colunas com caracteres do tipo CHAR ou INT) ao invés de 2 dígitos, para evitar problemas e confusões. Este problema não ocorre com colunas do tipo DATETIME, já que o século é armazenado mesmo que você especifique somente 2 dígitos. De qualquer modo, sempre é uma boa prática especificar o ano com 4 dígitos mesmo em colunas do tipo DATETIME.

GOTO

Do mesmo modo que ocorre em outras linguagens de programação, não utilize o GOTO, ou então, utilize-o raramente. O uso excessivo do GOTO pode gerar um código muito difícil de se interpretar.

Constantes

Lembre-se de que o T-SQL não possui o conceito de constantes (do mesmo modo que ocorre em outras linguagens como o Visual Basic e o C, por exemplo). Você pode utilizar variáveis para os mesmos fins. Ao utilizar variáveis ao invés de valores constantes literais em suas consultas, você facilita a compreensão e a manutenção do seu código. O exemplo a seguir, ilustra isso.

```
SELECT OrderID, OrderDate
FROM dbo.Orders
WHERE OrderStatus IN (5,6)
```

A mesma consulta pode ser escrita de um modo bem mais legível como ilustrado a segui.

```
DECLARE @ORDER_DELIVERED, @ORDER_PENDING
SELECT @ORDER_DELIVERED = 5, @ORDER_PENDING = 6

SELECT OrderID, OrderDate
FROM Orders
WHERE OrderStatus IN (@ORDER_DELIVERED, @ORDER_PENDING)
```

ORDER BY

Não utilize o numero das colunas na cláusula ORDER BY. Veja o exemplo abaixo, onde a segunda consulta é mais fácil de se entender do que a primeira.

```
SELECT OrderID, OrderDate
FROM Orders
ORDER BY 2

SELECT OrderID, OrderDate
FROM Orders
ORDER BY OrderDate
```

UNION

O operador UNION realiza um comando SELECT DISTINCT entre todos os conjuntos de resultados. Para a realização desta atividade, tabelas temporárias são automaticamente criadas e um table scan ocorre entre estas tabelas para garantir a unicidade dos dados. Você pode otimizar este processo utilizando o operador UNION ALL. A diferença é que o UNION ALL não garante a unicidade dos dados, assim, não é necessária a execução do table scan.

Triggers

No SQL Server 6.5 você só podia definir 3 triggers por tabela, uma para o INSERT, outra para o UPDATE e outra para o DELETE. A partir do SQL Server 7.0 esta restrição foi eliminada, e você passou a poder criar várias triggers por ação. Mas no SQL Server 7.0 você não pode controlar a ordem em que as triggers são acionadas. No SQL Server 2000 você pode definir que trigger deve ser acionada antes e que trigger deve ser acionada depois, utilizando a procedure sp_settriggerorder. Até o SQL Server 7.0, as triggers são executadas somente após a modificação dos dados. Por este motivo, elas são chamadas de post triggers. Mas no SQL Server 2000 você pode criar triggers que sejam acionadas antes da modificação dos dados. Elas são chamadas de INSTEAD OF triggers. Você pode utilizar triggers para implementar regras de negócios e auditoria. As triggers também podem ser utilizadas como uma extensão da verificação de regras referenciais, mas sempre que possível, utilize constraints para este fim.

Criação de Instância de objetos a partir do SQL

A criação de instância de objetos a partir de triggers e stored procedures, é um processo que consume muito tempo e

atrapalha do desempenho como um todo. O mesmo ocorre com o envio de e-mails a partir de triggers e stored procedures. Em situações como esta, é mais interessante a implementação de uma solução que registre todos os dados necessários em uma tabela a parte e a criação de uma rotina a ser executada com certa periodicidade (Job) para a consulta da tabela e tomada de ações necessárias.

Transações e níveis de isolamento

Entender o conceito de transação e de como ela pode ser parametrizada é fundamental para o desenvolvimento de aplicações eficazes em ambientes complexos e de grande concorrência.

Transação é uma seqüência de operações executadas como uma unidade lógica simples de trabalho e deve possuir quatro propriedades chamadas de ACID (Atomicity, Consistency, Isolation, Durability). Em cada transação no MS-SQL Server você pode utilizar quatro níveis de isolamento diferentes:

- READ COMMITTED, que é o padrão do SQL e utiliza locks do tipo Shared enquanto a informação está sendo processada;
- READ UNCOMMITTED, que faz uso de leituras do tipo "dirty reads" (os locks exclusivos não são respeitados), e é equivalente ao uso do NOLOCK na consulta;
- REPEATABLE READ, que implementa lock em todos os registros solicitados pela consulta mas não impede que novos registros sejam incluídos;
- SERIALIZABLE, que é o nível de isolação mais restritivo e impede que alterações ou novas inclusões sejam realizadas, além de não ser recomendado para situações em que se têm muitos usuários concorrentes;

Procure sempre trabalhar com um nível de isolamento menos restritivo possível de modo a privilegiar o desempenho e a concorrência em sua aplicação. O nível de isolamento é uma característica da conexão e pode ser alterado através do comando:

```
SET TRANSACTION ISOLATION LEVEL
    { READ COMMITTED
        | READ UNCOMMITTED
        | REPEATABLE READ
        | SERIALIZABLE
    }
```

NOLOCK

Uma forma de se reduzir a quantidade de locks e permitir um maior número de processos concorrentes em sua aplicação melhorando desempenho, é fazer uso do hint NOLOCK.

Normalmente o SQL Server aplica um lock do tipo Shared em quaisquer linhas que sejam consultadas por uma procedure ou por uma aplicação. Em alguns casos estes locks de linha do tipo Shared podem ser escalados para locks de página e de tabela, diminuindo a concorrência e degradando o desempenho em seu banco de dados. Se uma consulta tem um tempo de execução muito elevado, ela pode impedir que outros usuários tenham acesso a estes dados para alterações com comandos INSERT ou UPDATE.

Estes locks do tipo Shared são aplicados para prevenir que os dados utilizados em uma transação sejam alterados enquanto a transação ainda está ativa. Só que ele normalmente não é necessário em aplicações que geram relatórios ou que somente consultam dados. Se este for o seu caso, o uso do hint NOLOCK melhorará significativamente o desempenho das suas consultas. A seguir um exemplo:

```
SELECT au_lname
FROM authors WITH (NOLOCK)
GO
```

O exemplo acima permite, por exemplo, que esta consulta seja processada e retorne resultados mesmo que um ou mais registros da tabela authors estejam sendo alterados por um outro usuário.

ROWLOCK

Em algumas situações, é comum utilizar números seqüências gerados a partir de uma tabela com um campo do tipo IDENTITY. Só que dependendo da lógica de programação utilizada na solução, podem existir situações de contenção onde um usuário não consegue obter um novo numero seqüencial enquanto a transação de um outro usuário para o mesmo fim, estiver ativa. Esta é uma das situações em que o uso do hint ROWLOCK pode trazer um grande ganho de desempenho, permitindo um número maior de processos concorrentes no banco de dados.

Suponha que seu processo para obtenção de números seqüências utilize uma tabela com um campo identity. Em um determinado momento da sua aplicação, uma transação é iniciada para a obtenção de um novo número seqüencial e a atualização de algumas outras tabelas no seu banco de dados. O Exemplo a seguir ilustra esta situação:

```
-- Cria a tabela sequencial e insere um valor inicial
create table seq(num int)
go
insert seq values(1)
go
-- Rotina que obtem o numero sequencial e continua o processamento
begin tran
declare @num int
select @num = max(num) from seq
insert seq select @num + 1
...
-- Atualização de outras tabelas necessarias para a transação
...
commit tran
```

Neste exemplo, enquanto a transação estiver ativa processando "Atualização de outras tabelas necessárias para a transação" todos os outros usuários que tentarem obter um outro numero seqüencial ficarão travados aguardando o termino desta transação. Isso ocorre devido ao fato de que quando um novo registro é incluído na tabela que gera os números seqüenciais, o SQL Server aplica um lock de página ou de tabela ao invés de utilizar um lock de linha, que tem um custo maior para o gerenciador de banco de dados.

Esta situação pode ser contornada com o uso dos hints NOLOCK e ROWLOCK, conforme o exemplo a seguir:

```
-- rotina que obtem o numero sequencial de modo concorrente sem travamentos
begin tran
declare @num int
select @num = max(num) from seq (nolock)
insert seq with (rowlock) select @num + 1
...
-- Atualização de outras tabelas necessarias para a transação
...
commit tran
```

Neste exemplo, vários usuários poderão obter os números seqüenciais de modo concorrente, independente do tempo necessário para o processamento da "Atualização de outras tabelas necessárias para a transação". A única ressalva é que com este tipo de implementação, existe a possibilidade de faltarem alguns números seqüenciais em sua tabela. Isso devido ao fato de que se um primeiro usuário inicia uma transação e obtém, por exemplo, o numero 60, um segundo usuário que iniciar outra transação, obterá o número 61 (note que nenhuma das transações foi concluída ainda). Caso ocorra algum problema durante o processamento da transação do primeiro usuário (alguma regra de negócio não foi obedecida, por exemplo), esta transação será cancelada (sofrerá um rollback) enquanto a transação do segundo usuário foi processada sem problemas. Deste modo, a tabela seqüencial não possuirá o numero 60, visto que a transação do primeiro usuário não foi concluída com sucesso, mas possuirá o numero 61 da transação processada corretamente do segundo usuário.

Linked Server

O uso de Linked Servers é um recurso que possibilita muito mais flexibilidade na programação e troca de dados entre banco de dados de diferentes servidores SQL Server, entretanto, possui algumas limitações. Uma delas é não permitir o uso de hints em consultas remotas, o que impossibilita a utilização tanto do NOLOCK quanto do ROWLOCK anteriormente mencionados. Com isso, você perde desempenho e diminui a concorrência no banco de dados.

Para contornar este problema, você pode utilizar três abordagens diferentes. A primeira delas, é fazer uso de views, a segunda de procedures, e a terceira o uso do OPENQUERY.

Na primeira, fazendo uso de views, a mudança a ser realizada é a seguinte. Ao invés de acessar diretamente a tabela no servidor remoto em seu comando através do linked Server, acesse uma view da tabela no servidor remoto que contenha o hint NOLOCK. Deste modo você conseguirá acesso ao objeto através do Linked Server e no servidor remoto será processada a consulta com o uso do NOLOCK.

Na segunda, utilizando procedures, crie uma procedure no servidor remoto fazendo uso do NOLOCK e do ROWLOCK quando necessário. A equipe do Corporativo adotou esta solução nos seus serviços (fazendo uso do NOLOCK) e a equipe do Caixa Global que faz uso desta solução obteve um ganho de desempenho bastante significativo. A seguir um exemplo da solução implantada.

Supomos que tenhamos dois servidores de SQL distintos S1 e S2.

Se existir uma rotina no S1 que faz uso, diretamente, de dados de tabelas do S2. Como o exemplo abaixo:

```
S1:
...
SELECT      A.NM_AUTOR,
            A.NM_NACIONALIDADE_AUTOR,
            B.NM_LIVRO,
            B.NM_EDITORA
FROM        S2.BIBLIOTECA.DBO.LPT_AUTOR   A
            INNER JOIN
            S2.BIBLIOTECA.DBO.LPT_LIVRO   B
            ON    A.ID_AUTOR = B.ID_AUTOR
WHERE A.NM_AUTOR = @AUTOR
...
```

Devido ao fato do SQL não permitir que coloquemos o NOLOCK em selects de servidores remotos é preferível que criemos procedures no servidor remoto (S2) e que elas sejam apenas executadas do Servidor local (S1). Além de a procedure manter o plano de acesso a tabela pré-compilado, podemos colocar o NOLOCK no Select, melhorando o desempenho em tabelas onde a concorrência é grande. Como ficaria:

```
S1:
...
EXEC S2.BIBLIOTECA.DBO.LPP_OBTER_LIVRO_AUTOR(@AUTOR)
...
S2:
...
CREATE PROCEDURE DBO. LPP_OBTER_LIVRO_AUTOR @AUTOR VARCHAR (50)
AS
SELECT      A.NM_AUTOR,
            A.NM_NACIONALIDADE_AUTOR,
            B.NM_LIVRO,
            B.NM_EDITORA
FROM        BIBLIOTECA.DBO.LPT_AUTOR      A     (NOLOCK)
            INNER JOIN
            BIBLIOTECA.DBO.LPT_LIVRO      B     (NOLOCK)
            ON    A.ID_AUTOR = B.ID_AUTOR
WHERE A.NM_AUTOR = @AUTOR
GO
```

Na terceira alternativa, você pode fazer uso da função OPENQUERY. Esta função pode ser especificada na cláusula FROM da

consulta a ser executada com a mesma funcionalidade do nome de uma tabela. A função OPENQUERY pode ainda ser utilizada em comandos INSERT, UPDATE, e DELETE. Ela recebe como argumentos, o nome do linked server a ser acessado, e um string com o comando a ser executado. Suas principais vantagens são o fato da função executar uma consulta do tipo "pass-through" (o que significa que o comando será enviado diretamente ao servidor de destino para ser processado), e o fato de permitir o uso de query hints (NOLOCK, ROWLOCK, etc). Como principais desvantagens se destacam o fato da função OPENQUERY não aceitar variáveis como argumento, e o limite de 8000 caracteres para cada argumento. A seguir um exemplo.

```
SELECT      *
FROM        OPENQUERY(SQL04, 'SELECT * FROM NORTHWIN.DBO.ORDERS (NOLOCK)')
```

SQL Injection

SQL Injection é uma classe de ataque onde o invasor pode inserir ou manipular consultas criadas pela aplicação web que são enviadas para o backend (SQL). Este tipo de ataque pode pode ser realizado principalmente pelos seguintes motivos:

- A aplicação aceita dados arbitrários fornecidos pelo usuário
    - Confia que o usuário fornecerá dados corretos
- A aplicação faz uso de query dinâmica
- As conexões são feitas no contexto de um usuário com privilégios altos

Em aplicações web com estas caracteristicas, o atacante pode alterar o comando submetido ao banco de dados e em alguns casos, executar praticamente qualquer comando no servidor SQL.

Como prevenir:

- Nunca se conecte com um usuário dbo
    - Utilize um usuário que possua somente os acessos realmente necessários
- Construa comandos SQL seguros
    - Valide os dados digitados pelo usuário: procure pelos dados corretos; o restante deve ser considerado erro (Expressões regulares)
    - Não retorne mensagens de erro do banco de dados diretamente para o usuário. Estas mensagens podem revelar informações sobre seu servidor
    - Utilize entrada de dados parametrizada; nunca concatene strings (query dinâmicas)
    - Valide os dados em todas as camadas (Aplicação, componente, banco de dados, etc.)

Mais informações sobre este assunto podem ser encontradas na edição 23 (Setembro/2005) da revista SQLMagazine (http://www.devmedia.com.br/sqlmagazine) no artigo "SQL Injection: o que é, por que funciona e como previnir"

Objetos criptografados (WITH ENCRYPTION)

O uso de objetos criptografados gera overhead de processamento além de inviabilizar a depuração de problemas quando as rotinas estão em execução, e ainda não garantirem que o código fonte do objeto seja acessado. Por estes motivos não devem ser utilizados.

Última revisão: 12/Setembro/2005