Relatório de consultoria

***<cliente>***

*Consultoria em SQL Server 2008 R2 – análise de ambiente e recomendações de desempenho para carga massiva de dados.*

Maio de 2012

Brasília, 04 de Maio de 2012

Sumário

[Resumo do serviço 3](#_Toc325027799)

[Considerações gerais 3](#_Toc325027800)

[Processo atual 3](#_Toc325027801)

[Resultados finais 4](#_Toc325027802)

[Alterações efetuadas e sugeridas 6](#_Toc325027803)

[Alterações na aplicação 6](#_Toc325027804)

[Paralelismo 6](#_Toc325027805)

[Controle do paralelismo 8](#_Toc325027806)

[Quantidade de linhas, threads e BatchSize 12](#_Toc325027807)

[Configurações servidor SQL Server 13](#_Toc325027808)

[TRACEFLAG 610 e 2371 13](#_Toc325027809)

[Service Pack 1 15](#_Toc325027810)

[Read Commited Snapshot Isolation Level 15](#_Toc325027811)

[Compressão de dados 16](#_Toc325027812)

[Melhoria das consultas 17](#_Toc325027813)

[Particionamento (expurgo de dados da <Tabela>) 20](#_Toc325027814)

[Particionamento (remover particionamento atual) 24](#_Toc325027815)

[Configurações gerais 24](#_Toc325027816)

[Recovery interval para 0 24](#_Toc325027817)

[TEMPDB 25](#_Toc325027818)

[Configurações Hardware 25](#_Toc325027819)

[HyperThreading 25](#_Toc325027820)

# Resumo do serviço

Durante duas semanas a Sr. Nimbus esteve atuando junto ao <cliente> para analisar a situação do ambiente SQL Server 2008 R2 com o objetivo de sugerir melhorias no ambiente a fim de obter máxima performance para inserção de dados e consultas no SQL Server. Este relatório é o resultado deste serviço e contém todas as recomendações para o ambiente, bem como a análise de algumas alternativas propostas pela equipe da Sr. Nimbus, alternativas estas que devem ser testadas e seguidas pela equipe do <cliente>.

Neste projeto a consultoria foi executada pelos profissionais: Fabiano Amorim – responsável pela consultoria - e Luciano Moreira. Foram necessárias um total de 96 horas para execução do serviço.

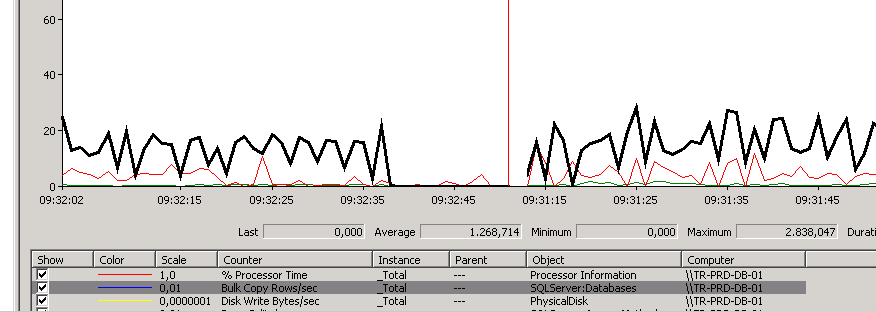
# Considerações gerais

O principal objetivo do projeto consiste em inserir a maior quantidade de linhas possíveis em tabelas que serão utilizadas pela aplicação para consulta de diversas informações. Além de receber uma carga de dados extremamente grande, as tabelas devem continuar acessíveis para consulta.

Existem alguns pontos chaves em relação à inserção de dados massiva no SQL Server, ao longo deste relatório iremos destacar e explicar todas as alterações e testes efetuados no ambiente, a fim de que as técnicas sejam replicadas no ambiente de produção, e as devidas alterações sejam efetuadas na aplicação que envia os dados para o servidor SQL Server.

# Processo atual

Atualmente o processo de inserção de dados consegue inserir uma média de 2 mil linhas por segundo, com picos de ***2.8*** mil linhas. Abaixo podemos visualizar um gráfico capturado no ambiente de testes quando o software enviava os comandos de insert para o banco de dados:



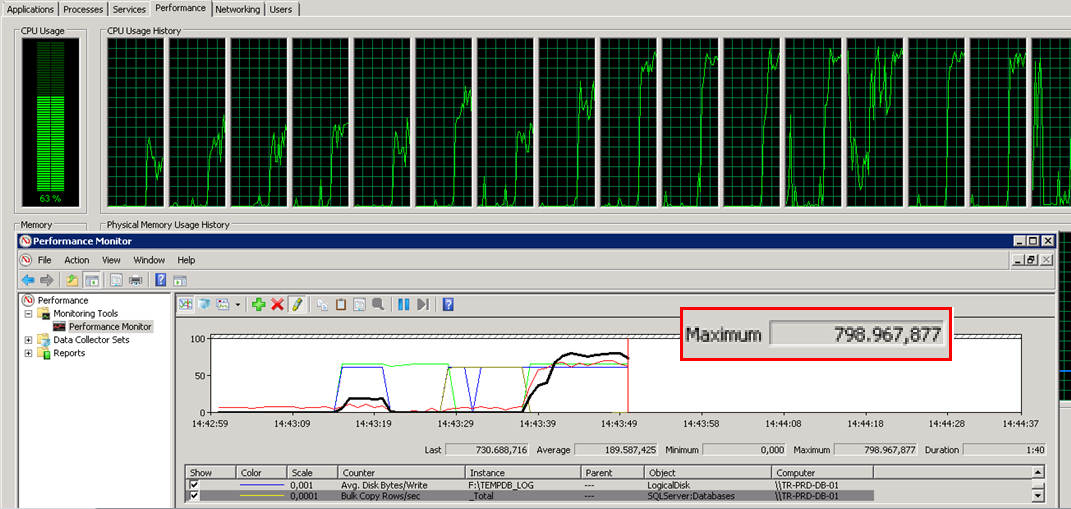
O gráfico acima exibe o número máximo de ***2.838,04*** linhas inseridas por segundo.

Outros testes pontuais utilizando o comando BULK INSERT e efetuados pela equipe do Valor exibiram um número máximo de 8 mil linhas por segundo.

# Resultados finais

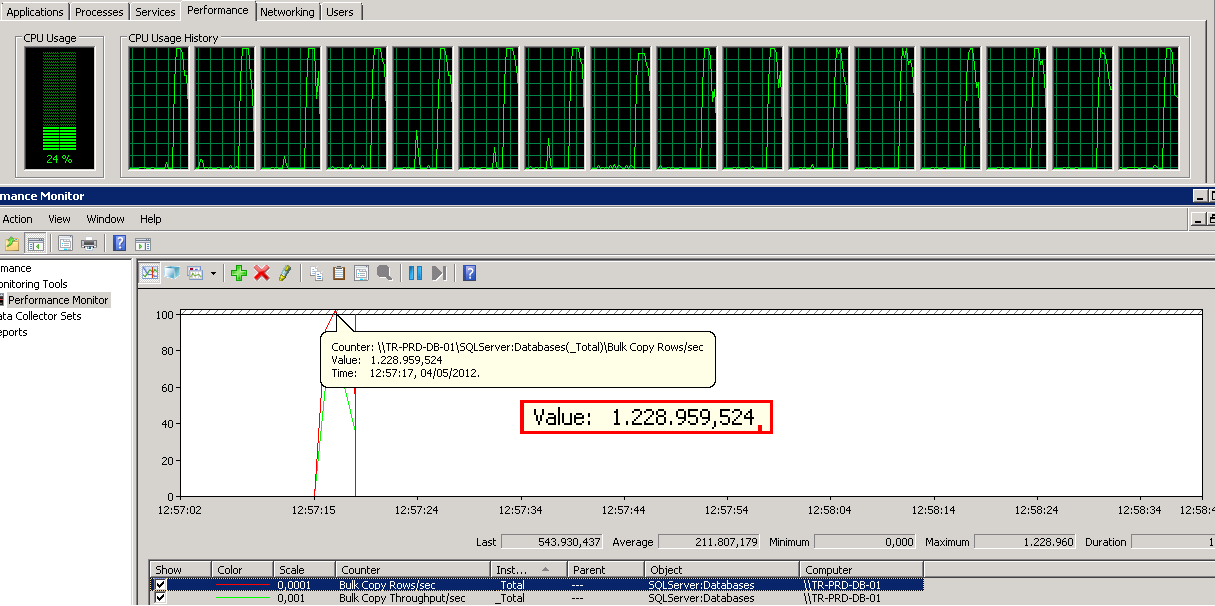
Após efetuar várias alterações no servidor e na aplicação que envia os dados para o servidor SQL Server, conseguimos atingir um pico de ***798.967,88*** linhas inseridas em apenas 1 segundo. Para isso, os recursos de hardware disponíveis são bem utilizados.

O gráfico abaixo exibe o número mencionado acima:



O gráfico acima foi capturado quando a aplicação utilizada para simular os testes estava rodando.

Durante os testes, utilizando o BCP para ler diretamente dos arquivos textos, conseguimos um número ainda maior, conforme podemos visualizar no gráfico abaixo conseguimos importar 3.6 milhões, obtendo picos de ***1.228.959,52*** milhões de inserts por segundo.



A intenção do gráfico acima é exibir que o SQL Server consegue inserir um número ainda maior do que o necessário para o projeto.

# Alterações efetuadas e sugeridas

## Alterações na aplicação

### Paralelismo

Um dos pontos chave em relação à carga massiva de dados no SQL Server é o uso de operações em paralelo.

Para obter máxima performance nas operações de carga, as operações devem ser dividas em vários operações e enviadas para o SQL Server em simultâneo. Somente desta forma conseguimos utilizar todo o poder de processamento e escrita do hardware disponível. Operações sequenciais, não conseguem utilizar todos os recursos que o hardware oferece.

Para confirmar esta hipótese, inicialmente utilizamos uma [API](http://www.codeproject.com/Articles/29356/Asynchronous-T-SQL-Execution-Without-Service-Broke) escrita em CLR (Common Language Runtime) para importar vários arquivos texto para a tabela <Tabela> no banco de dados <BANCO>.

Segue abaixo o script utilizado para efetuar esta tarefa:

-- Seta parâmetros das threads

EXEC <BANCO>.dbo.Parallel\_Declare 'BlocoInserts'

EXEC <BANCO>.dbo.Parallel\_SetOption\_MaxThreads 16

EXEC <BANCO>.dbo.Parallel\_SetOption\_CommandTimeout 30

-- Gera threads

DECLARE @i Int = 1, @SQL VarChar(8000), @ParralelName VarChar(80)

WHILE @i <= 16 -- Existem 16 arquivos na pasta

BEGIN

SET @ParralelName = 'SQL' + CONVERT(VarChar, @i);

SET @SQL = 'bcp "<BANCO>.dbo.<Tabela>" in "F:\SYSDBs\SrNimbus\TMP\<Tabela>TMP'+CONVERT(VarChar, @i)+'.txt" -c -T -b 50000 -q -t, -S ' + @@servername

SET @SQL = 'EXEC xp\_cmdshell ' + '''' + @SQL + ''''

PRINT @SQL

EXEC <BANCO>.dbo.Parallel\_AddSql @ParralelName, @SQL

SET @i += 1;

END

-- Executa threads em simultaneo

DECLARE @RC int

EXEC @RC = <BANCO>.dbo.Parallel\_execute

IF @RC != 0

BEGIN

DECLARE @ErrorMessage varchar(MAX)

SET @ErrorMessage = <BANCO>.dbo.Parallel\_GetErrorMessage()

print @ErrorMessage

END

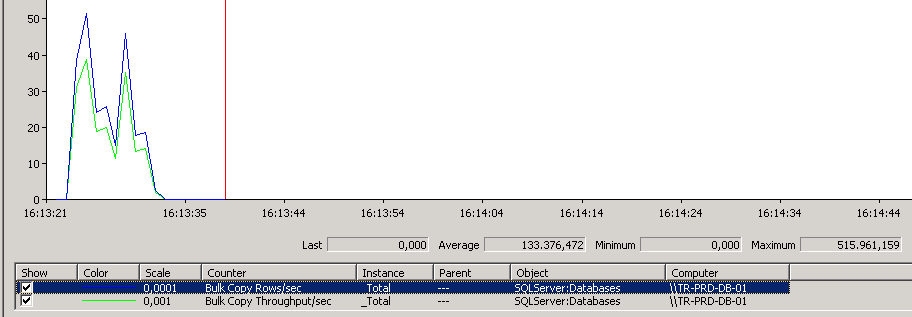
-- Visualiza o resultado das threads

SELECT \* FROM <BANCO>.dbo.parallel\_GetExecutionResult()

GO

O script acima importa 16 arquivos txt com 300 mil linhas cada para a tabela <Tabela>, as operações de insert são executada utilizando o utilitário nativo do SQL Server para carga de dados o BCP. Os arquivos são importados em paralelo, o controle do paralelismo é efetuado pelos objetos em CLR onde threads nativas do SQL Server são utilizadas.

Conforme podemos visualizar abaixo, utilizando o processo acima conseguimos comprovar que era possível chegar próximo do número de inserções desejadas para execução do projeto, as 4.800.000,00 de linhas foram inseridas na tabela em apenas 10 segundos, obtendo uma média de 480.000,00 inserts por segundo e um pico de ***515.961,15*** inserts por segundo.



Após obter a confirmação de que o processo de inserção em paralelo conseguiria inserir a quantidade de linhas desejada, utilizamos o mesmo conceito de paralelismo na aplicação que envia os inserts para o banco de dados.

Depois de analisar o código fonte do FeedHandler (aplicativo que efetua leitura da fila e envia inserts para o banco de dados), identificamos que o processo de leitura dos dados que serão enviados para o SQL Server, são inseridos em um objeto DataTable em uma aplicação .NET e enviados para o SQL Server utilizando a classe de SqlBulkCopy.

A fim de gerar o mínimo de impacto na aplicação, e obter melhor performance possível, a Sr.Nimbus criou um aplicativo para simular o mesmo comportamento do FeedHandler (porém, lendo dados de arquivos .txt) e enviando os inserts para o banco de dados em paralelo utilizando recursos da classe Task Parallel Library disponível no .NET 4.0. O aplicativo foi disponibilizado para a equipe do <CLIENTE>, para que o mesmo conceito fosse aplicado no FeedHandler.

Abaixo podemos visualizar os métodos para execução das operações em paralelo:

Parallel.ForEach(DataTables, table =>

{

BulkLoadData2(table);

}

);

public void BulkLoadData2(DataTable dt)

{

using (SqlConnection conn = new SqlConnection(textBox1.Text.ToString()))

{

conn.Open();

using (SqlBulkCopy bulkCopy = new SqlBulkCopy(conn))

{

int anInteger;

anInteger = Convert.ToInt32(txtBatchSize.Text);

bulkCopy.BatchSize = anInteger;

bulkCopy.DestinationTableName = "dbo.<Tabela>";

bulkCopy.BulkCopyTimeout = 60;

bulkCopy.WriteToServer(dt);

bulkCopy.Close();

}

}

}

A ideia por traz do aplicativo é simples, popular vários objetos do tipo DataTable com as linhas lidas nos arquivos .txt e disparar os comandos de BulkInsert em paralelo.

O código fonte completo do aplicativo será entregue junto com este relatório.

### Controle do paralelismo

Um dos pontos levantado pela equipe do Valor, é que a ordem dos dados que estão sendo inseridos em paralelo deve ser mantida, pois o processo de consulta dos dados só pode visualizar uma informação caso todo o processo que esta rodando em paralelo tenha finalizado.

Para garantir que os usuários somente visualizem as linhas inseridas no banco quando todas as linhas de uma operação em paralelo terminarem, foram criadas tabelas auxiliarem que serão utilizadas para controlar quando uma operação finalizou.

Cada operação de inserção em paralelo receberá um ID e este ID será gravado na tabela destino. Ao consultar a tabela, apenas os registros que possuem IDs com status “finalizado” poderão ser visualizados.

Abaixo podemos visualizar o script criado para controle do processo.

Incluir coluna ID na tabela <Tabela>

IF NOT EXISTS(SELECT \* FROM syscolumns WHERE id = OBJECT\_ID('<Tabela>') AND name = 'ID')

BEGIN

ALTER TABLE <Tabela> ADD ID BigInt

END

GO

Criar uma tabela que será utilizada para controle dos inserts:

IF OBJECT\_ID('TabIdentity<Tabela>') IS NOT NULL

DROP TABLE dbo.TabIdentity<Tabela>

GO

CREATE TABLE dbo.TabIdentity<Tabela>

(

ID bigint NOT NULL IDENTITY(-9223372036854775808, 1) PRIMARY KEY,

StartDate DateTime NOT NULL DEFAULT GetDate(),

EndDate DateTime NULL

)

CREATE INDEX ixEndDate ON TabIdentity<Tabela>(EndDate) where EndDate IS NULL

GO

Criar uma view que deverá ser utilizada para ***TODAS*** as leituras efetuadas na tabela.

-- View to select <Tabela> table

IF OBJECT\_ID('vw\_<Tabela>') IS NOT NULL

DROP VIEW dbo.vw\_<Tabela>

GO

CREATE VIEW dbo.vw\_<Tabela>

WITH SCHEMABINDING

AS

SELECT TickId,

OriginId,

SymbolCode,

Timepoint,

TradeId,

SessionLast,

SessionLastSize,

BuyerId,

SellerId,

EraseTimestamp,

UpdateTimestamp

FROM dbo.<Tabela>

WHERE ID <= (SELECT MAX(ID)

FROM dbo.TabIdentity<Tabela>

WHERE EndDate IS NOT NULL)

AND ID NOT EXISTS(SELECT ID

FROM dbo.TabIdentity<Tabela>

WHERE EndDate IS NOT NULL)

GO

Criar procedure que será utilizada para retorna o um ID para um processo de insert em paralelo. A procedure abaixo possui um controle para identificar quando um processo de inserção iniciou porém não foi finalizado, possivelmente por causa de um erro durante os inserts, neste caso, as linhas inseridas na tabela <Tabela> devem ser removidas, antes de uma nova carga iniciar.

IF OBJECT\_ID('st\_Start<Tabela>ID') IS NOT NULL

DROP PROC dbo.st\_Start<Tabela>ID

GO

CREATE PROC dbo.st\_Start<Tabela>ID (@ID BigInt OUTPUT)

AS

BEGIN

-- Manage "uncommited rows"

DECLARE @TempID BigInt

-- Make sure this is indexed

-- create index ixEndDate on TabIdentity<Tabela>(EndDate) where EndDate IS NULL

SELECT @TempID = ID

FROM TabIdentity<Tabela>

WHERE EndDate IS NULL

IF @@ROWCOUNT > 0

BEGIN

-- Make sure this is indexed

-- create index ixID on <Tabela>(ID)

DELETE FROM <Tabela> WITH(ROWLOCK)

WHERE ID = @TempID

DELETE FROM TabIdentity<Tabela>

WHERE ID = @TempID

END

INSERT INTO TabIdentity<Tabela> DEFAULT VALUES

SET @ID = SCOPE\_IDENTITY();

END

GO

Nota: O processo acima requer que as execuções dos inserts em paralelo sejam efetuadas por apenas um aplicativo por vez, caso mais de um aplicativo utilize este controle, o controle atual deverá ser alterado para suportar esse tipo de processo. Uma alternativa simples pode ser a seguinte, o código utilizado acima para excluir os dados que não foram “comitados” pode ser executado por um job diariamente.

Nota 2: A coluna ID precisa fazer parte da PK das tabelas <Tabela> e <Tabela>.

Código para finalizar um ID:

IF OBJECT\_ID('st\_End<Tabela>ID') IS NOT NULL

DROP PROC dbo.st\_End<Tabela>ID

GO

CREATE PROC dbo.st\_End<Tabela>ID (@ID BigInt)

AS

BEGIN

UPDATE TabIdentity<Tabela> SET EndDate = GetDate()

WHERE ID = @ID

END

Teste do processo:

-- Inicia um ID

DECLARE @ID BigInt

EXEC st\_Start<Tabela>ID @ID = @ID OUT;

-- Executa inserts na tabela <Tabela> utilizando o ID recebido acima

INSERT [dbo].[<Tabela>] ([TickId], [OriginId], [SymbolCode], [UpdateTimestamp], [BestBid], [BestBidSize], [BestBidPartCode], [BestAsk], [BestAskSize], [BestAskPartCode], ID) VALUES (74956, 2, N'BBAS3', CAST(0x07006EE644B677350B AS DateTime2), CAST(27.260000000 AS Numeric(28, 9)), 400, N'23', NULL, NULL, NULL, @ID)

INSERT [dbo].[<Tabela>] ([TickId], [OriginId], [SymbolCode], [UpdateTimestamp], [BestBid], [BestBidSize], [BestBidPartCode], [BestAsk], [BestAskSize], [BestAskPartCode], ID) VALUES (74957, 2, N'BBAS3', CAST(0x0780D49158B677350B AS DateTime2), CAST(27.270000000 AS Numeric(28, 9)), 100, N'21', NULL, NULL, NULL, @ID)

INSERT [dbo].[<Tabela>] ([TickId], [OriginId], [SymbolCode], [UpdateTimestamp], [BestBid], [BestBidSize], [BestBidPartCode], [BestAsk], [BestAskSize], [BestAskPartCode], ID) VALUES (74961, 2, N'BBAS3', CAST(0x0780D49158B677350B AS DateTime2), NULL, NULL, NULL, CAST(27.280000000 AS Numeric(28, 9)), 400, N'10', @ID)

INSERT [dbo].[<Tabela>] ([TickId], [OriginId], [SymbolCode], [UpdateTimestamp], [BestBid], [BestBidSize], [BestBidPartCode], [BestAsk], [BestAskSize], [BestAskPartCode], ID) VALUES (74965, 2, N'BBAS3', CAST(0x07006B2A59B677350B AS DateTime2), CAST(27.260000000 AS Numeric(28, 9)), 400, N'23', NULL, NULL, NULL, @ID)

INSERT [dbo].[<Tabela>] ([TickId], [OriginId], [SymbolCode], [UpdateTimestamp], [BestBid], [BestBidSize], [BestBidPartCode], [BestAsk], [BestAskSize], [BestAskPartCode], ID) VALUES (74966, 2, N'BBAS3', CAST(0x0780B5875EB677350B AS DateTime2), CAST(27.270000000 AS Numeric(28, 9)), 100, N'386', NULL, NULL, NULL, @ID)

INSERT [dbo].[<Tabela>] ([TickId], [OriginId], [SymbolCode], [UpdateTimestamp], [BestBid], [BestBidSize], [BestBidPartCode], [BestAsk], [BestAskSize], [BestAskPartCode], ID) VALUES (74970, 2, N'BBAS3', CAST(0x0780C229D2B677350B AS DateTime2), NULL, NULL, NULL, CAST(27.290000000 AS Numeric(28, 9)), 500, N'82', @ID)

INSERT [dbo].[<Tabela>] ([TickId], [OriginId], [SymbolCode], [UpdateTimestamp], [BestBid], [BestBidSize], [BestBidPartCode], [BestAsk], [BestAskSize], [BestAskPartCode], ID) VALUES (74974, 2, N'BBAS3', CAST(0x0780EF5AD3B677350B AS DateTime2), NULL, NULL, NULL, CAST(27.290000000 AS Numeric(28, 9)), 100, N'82', @ID)

INSERT [dbo].[<Tabela>] ([TickId], [OriginId], [SymbolCode], [UpdateTimestamp], [BestBid], [BestBidSize], [BestBidPartCode], [BestAsk], [BestAskSize], [BestAskPartCode], ID) VALUES (74978, 2, N'BBAS3', CAST(0x0780EF5AD3B677350B AS DateTime2), CAST(27.260000000 AS Numeric(28, 9)), 400, N'23', NULL, NULL, NULL, @ID)

INSERT [dbo].[<Tabela>] ([TickId], [OriginId], [SymbolCode], [UpdateTimestamp], [BestBid], [BestBidSize], [BestBidPartCode], [BestAsk], [BestAskSize], [BestAskPartCode], ID) VALUES (74982, 2, N'BBAS3', CAST(0x0780EF5AD3B677350B AS DateTime2), CAST(27.260000000 AS Numeric(28, 9)), 1000, N'23', NULL, NULL, NULL, @ID)

INSERT [dbo].[<Tabela>] ([TickId], [OriginId], [SymbolCode], [UpdateTimestamp], [BestBid], [BestBidSize], [BestBidPartCode], [BestAsk], [BestAskSize], [BestAskPartCode], ID) VALUES (74986, 2, N'BBAS3', CAST(0x0780EF5AD3B677350B AS DateTime2), CAST(27.250000000 AS Numeric(28, 9)), 100, N'72', NULL, NULL, NULL, @ID)

INSERT [dbo].[<Tabela>] ([TickId], [OriginId], [SymbolCode], [UpdateTimestamp], [BestBid], [BestBidSize], [BestBidPartCode], [BestAsk], [BestAskSize], [BestAskPartCode], ID) VALUES (74987, 2, N'BBAS3', CAST(0x07008EA203B777350B AS DateTime2), CAST(27.280000000 AS Numeric(28, 9)), 200, N'386', NULL, NULL, NULL, @ID)

INSERT [dbo].[<Tabela>] ([TickId], [OriginId], [SymbolCode], [UpdateTimestamp], [BestBid], [BestBidSize], [BestBidPartCode], [BestAsk], [BestAskSize], [BestAskPartCode], ID) VALUES (74991, 2, N'BBAS3', CAST(0x0700D46527B777350B AS DateTime2), NULL, NULL, NULL, CAST(27.290000000 AS Numeric(28, 9)), 1000, N'23', @ID)

INSERT [dbo].[<Tabela>] ([TickId], [OriginId], [SymbolCode], [UpdateTimestamp], [BestBid], [BestBidSize], [BestBidPartCode], [BestAsk], [BestAskSize], [BestAskPartCode], ID) VALUES (74995, 2, N'BBAS3', CAST(0x0700D46527B777350B AS DateTime2), NULL, NULL, NULL, CAST(27.290000000 AS Numeric(28, 9)), 900, N'23', @ID)

INSERT [dbo].[<Tabela>] ([TickId], [OriginId], [SymbolCode], [UpdateTimestamp], [BestBid], [BestBidSize], [BestBidPartCode], [BestAsk], [BestAskSize], [BestAskPartCode], ID) VALUES (74999, 2, N'BBAS3', CAST(0x0780292E48B777350B AS DateTime2), NULL, NULL, NULL, CAST(27.290000000 AS Numeric(28, 9)), 500, N'23', @ID)

-- Finaliza ID

EXEC st\_EndID @ID = @ID;

GO

Caso durante o inserts alguma sessão tente ler os dados da tabela utilizando a view vw\_<Tabela>, apenas os dados com IDs com EndDate NOT NULL serão visualizados.

Para visualizar como utilizar este controle no aplicativo, basta analisar o código do aplicativo enviado junto com este relatório.

### Quantidade de linhas, threads e BatchSize

Identificar o número ideal de operações que irão rodar em paralelo, qual a quantidade de linhas que cada thread irá trabalhar, e o tamanho dos BatchSize enviado para o SQL Server é primordial para que os recursos da máquina sejam utilizados da melhor maneira possível, consequentemente reduzindo o tempo de resposta do processo.

* Quantidade de threads: considerando que o servidor tem 16 cores, definimos que a quantidade de operações em paralelo (threads) será 16. Testes comprovaram que esse é o melhor número a ser utilizado.
* Número de linhas por thread: Quanto maior o número de linhas enviado por thread, melhor o desempenho das operações de inserção. Recomendamos que cada thread (datatable) trabalhe com no mínimo 30 mil linhas, preferencialmente utilizem o número de 50 mil linhas, ou mais.
* BatchSize: Batchsize é uma propriedade definida no objeto SqlBulkCopy. Este número diz para o SQL Server quantas linhas são controladas por transação de bulk insert.

É extremamente importante definir o valor correto para o BatchSize a fim de evitar que grandes volumes de dados sejam controlados e uma única transação no SQL Server. Além disso, cada operação de bulk insert executada no servidor irá gerar uma operação de SORT, a operação de SORT é executada para fazer com que os dados sejam inseridos na ordem do índice cluster da tabela. Esta operação de SORT pode ser executada em memória, ou no TEMPDB (banco de dados temporário do SQL Server). Como operações de SORT executadas em memória são muito mais eficientes do que operações efetuadas no TEMPDB, é necessário especificar um número de linhas no BatchSize que caiba em memória.

Durante os testes efetuados no servidor, identificamos que o número ideal do BatchSize para a tabela <Tabela> é de 30.000 (trinta mil) e para a tabela <Tabela> é de 25.000 (vinte cinco mil).

## Configurações servidor SQL Server

### TRACEFLAG 610 e 2371

Outro ponto extremamente importante em relação à carga de dados massiva no SQL Server, são operações minimamente logadas. Quando essas operações são executadas, o SQL Server mantém apenas o registro das informações necessárias para um restore da operação completa, e não um restore de uma operação em um ponto do tempo (point in time restore). Sendo assim elas são muito mais rápidas (por isso um TRUNCATE é mais rápido que um DELETE...). Truncate é um exemplo de operação minimalmente logada.

Operações de BULK INSERT podem ser minimamente logadas ou não dependendo de uma série de fatores. Para mais detalhes visualizar o seguinte [artigo](http://msdn.microsoft.com/en-us/library/dd425070%28v=sql.100%29.aspx).

No nosso caso em específico, para conseguirmos obter operações minimamente logadas em tabelas que possuem dados e contêm índices, o TraceFlag 610 deve ser habilitado. E o recovery model do banco de dados deve ser configurado para SIMPLE ou BULK-LOGGED.

Para conseguir utilizar o recurso de backup de LOG, recomendamos utilizar o recovery model BULK-LOGGED. Para fazer isso, basta rodar o script abaixo:

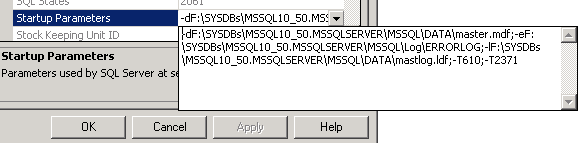
USE master

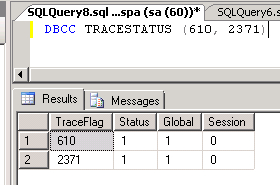
GO

ALTER DATABASE <BANCO> SET RECOVERY BULK\_LOGGED

GO

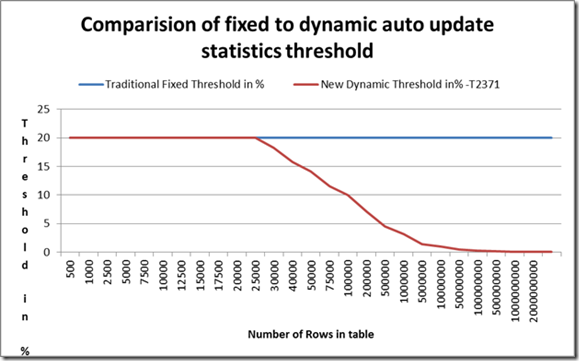
Além do revorery model bulk-logged, o traceflag 610 foi habilitado no servidor conforme podemos visualizar abaixo:





O traceflag 2371 também foi habilitado para mudar o comportamento da atualização de estatísticas das tabelas. Este traceflag foi incluído no SQL Server 2008 R2 service pack 1 e faz com que as estatísticas das tabelas sejam atualizadas automaticamente com mais frequência. Para mais detalhes em relação a este traceflag, leia o seguinte [artigo](http://blogs.msdn.com/b/saponsqlserver/archive/2011/09/07/changes-to-automatic-update-statistics-in-sql-server-traceflag-2371.aspx).

Abaixo podemos visualizar um gráfico que mostra o a diferença de comportamento do SQL Server quando este traceflag está habilitado.



Conforme podemos visualizar no gráfico acima o número de modificações na tabela necessárias para disparar um auto update statistics diminuir conforme o tamanho de linhas da tabela. Como iremos trabalhar com tabelas com muito mais de 25 mil linhas, este traceflag será útil.

Como as estatísticas serão atualizadas automaticamente com mais frequência, as consultas efetuadas nas tabelas irão trabalhar com estatísticas mais precisas e consequentemente tem maior chance de utilizar um plano de execução mais eficiente.

*Nota: O traceflag 2371 não remove a necessidade de uma rotina de manutenção para atualização das estatísticas.*

### Service Pack 1

Identificamos que a instancia do SQL Server instalada no servidor não estava atualizada. Para evitar problemas com bugs já corridos no SQL Server, ***instalamos o service pack 1*** que á a última versão disponível.

Recomendamos uma analise mais detalhada em relação a todas as correções lançadas após o service pack 1, e caso necessário, instalar a correção.

Segue abaixo o link para todas os CUs (cumulative updates) já lançados após o service pack:

|  |  |
| --- | --- |
| **Patch Level** | **Link** |
| 2008 R2 SP1 + Cumulative Update 1 | <http://support.microsoft.com/kb/2567616> |
| 2008 R2 SP1 + Cumulative Update 2 | <http://support.microsoft.com/kb/2567714> |
| 2008 R2 SP1 + Cumulative Update 3 | <http://support.microsoft.com/kb/2591748> |
| 2008 R2 SP1 + Cumulative Update 4 | <http://support.microsoft.com/kb/2633146> |
| 2008 R2 SP1 + Cumulative Update 5 | <http://support.microsoft.com/kb/2659694> |

### Read Commited Snapshot Isolation Level

Uma forma comum de visualizar os dados de uma tabela que potencialmente esta bloqueada por operações de carga, é utilizar o hint NOLOCK. Também conhecidas como “leituras sujas”, NOLOCK faz com que as consultas rodem no nível de isolamento Read Uncommited Isolation Level, ou seja, dados que ainda não foram “comitados” são lidos, o que pode representar resultados inconsistentes ou incompletos. Porém como as consultas com NOLOCK não obtém LOCKS, elas não são bloqueadas pelas escritas.

Desde o a versão 2005 o SQL Server fornece um mecanismo que impede que consultas lendo dados, gerem bloqueios ou fiquem bloqueadas. Para o nosso cenário, habilitar o RCSI (Read Commited Snapshot Isolation) é muito mais vantajoso do que utilizar o NOLOCK. Leituras feitas sob o nível de isolamento snaphot não leem “dados sujos”, ou seja, os resultados são consistentes pois ele utilizada um controle de versionamento para ler a ultima versão “comitada” do registro.

Para bancos que efetuam muitas operações de update e delete (o que não é o nosso caso), RCSI pode gerar contenção no tempdb (banco de dados utilizado para controlar o versionamento dos dados). O impacto gerado pelos inserts com RCSI habilitado é de 17 bytes adicionais que são incluídos em cada linha inserida na tabela.

Para o cenário analisado, recomendamos remover ***TODOS***os hints nolock dos códigos e habilitar o RCSI. Segue o comando para isso:

USE [master]

GO

ALTER DATABASE <BANCO> SET ALLOW\_SNAPSHOT\_ISOLATION ON

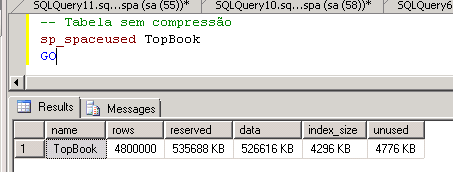
GO

ALTER DATABASE <BANCO> SET READ\_COMMITTED\_SNAPSHOT ON

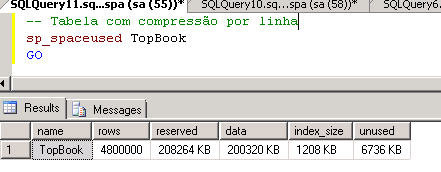
### Compressão de dados

Quando os dados que estão sendo inseridos na tabela são compactados, a compactação é efetuada durante o processamento de carga. Isso pode diminuir a performance dos inserts, porém o ganho em relação ao espaço utilizado pelas tabelas vale a pena para nosso cenário.

Abaixo podemos visualizar que a tabela <Tabela> com 4.8 milhões e sem compressão de dados utiliza aproximadamente ***535mb*** de espaço:

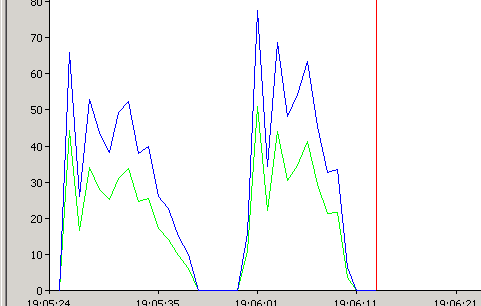


Com a compressão de dados por linha habilitado, o espaço utilizado é reduzido pra menos da metade, e a tabela ocupa aproximadamente ***208mb***.



O impacto no número de inserts por linha é mínimo comparado ao ganho de espaço que o uso da compressão representa.

Abaixo podemos visualizar o gráfico de inserts por segundo ao efetuar a importação das 4.8 milhões de linhas com compressão e sem compressão respectivamente nessa ordem, a diferença é mínima (temos até a impressão de que mais linhas foram inseridas com a compressão habilitada, porém maiores testes provaram que isso não é verdade).



Vale a pena mencionar que o fato de os dados estarem compactados não representa apenas ganho de espaço. Quanto menor a tabela, menos páginas, isso significa que mais linhas são incluídas por página. Quanto menor o número de páginas, menos trafego na rede, menos páginas no buffer cache (memória) e etc...

### Melhoria das consultas

Um dos itens que estava causando um péssimo desempenho das consultas estava relacionado a uma funcionalidade do SQL Server chamada Resource Governor.

Antes de falar sobre o Resource Governor, vamos entender melhor como o SQL Server utiliza memória na execução de uma consulta.

Sempre que uma consulta é executada um mínimo de memória é atribuído para a consulta, esse processo funciona para cada consulta, o padrão de memória que uma consulta pode utilizar é de 1 MB. Quando uma consulta é executada, ao gerar um plano de execução o otimizador de consultas do SQL Server analisa quanto de memória a consulta precisa para rodar, este processo é conhecido como “Memory Grant”.

Algumas operações executadas pelo SQL Server necessitam de memória, exemplos clássicos são as operações de ordenação (operador de SORT) e hash join (operador de HASH MATCH), durante a fase de criação de um plano de execução o otimizador estima quanto de memória será necessário para executar estas operações. Durante a fase de execução da consulta, caso a estimativa feita pelo SQL Server estiver errada, ou seja, ele estimou que precisaria de 1mb para executar uma ordenação, e na verdade ele precisa de 15MB, o processo de ordenação é efetuado em disco (e os dados são gravados no tempdb). Quando isso ocorre a performance da consulta é severamente penalizada.

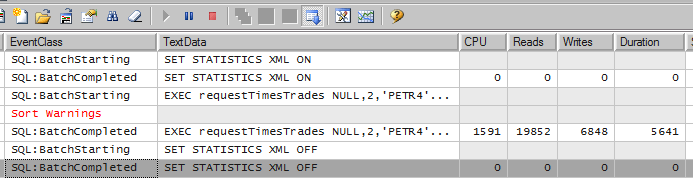
Em poucas palavras, o Resource Governor é utilizado para controlar e limitar quanto de memória e CPU um determinado usuário pode utilizar. O problema é que ao ser habilitado com as configurações padrão, a quantidade de memória que os usuários podem utilizar é de apenas 25%. Isso estava impactando diretamente no processo de memory grant e causando assim problemas de performance.

Para exemplificar o problema, vamos executar a procedure requestTimesTrades com o Resource Governor habilitado.

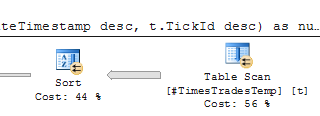
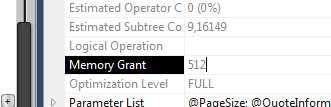
EXEC requestTimesTrades NULL,2,'PETR4','2012-03-22 00:00:00.0000000','2012-10-23 23:59:59.0000000',500,1,null

GO

Abaixo podemos visualizar que no profiler o evento de Sort Warning foi disparado e a consulta demorou 5 segundos para rodar.



Se analisarmos o plano de execução da consulta, poderemos ver que o SQL Server estimou que apenas 512KB seriam necessários para efetuar a operação de SORT envolvendo uma query dentro da procedure, segue parte do plano:

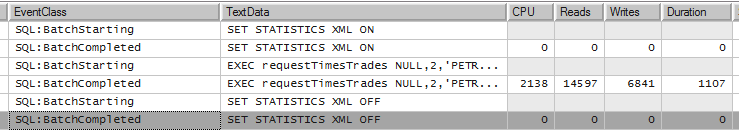
 

Agora vamos rodar a mesma consulta, com o Resource Governor desabilitado.

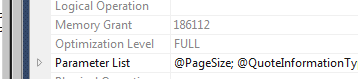
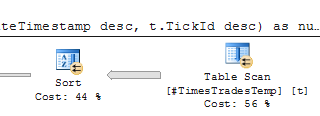
EXEC requestTimesTrades NULL,2,'PETR4','2012-03-22 00:00:00.0000000','2012-10-23 23:59:59.0000000',500,1,null

GO

Desta vez podemos visualizar no resultado do profiler que o sort warning não foi gerado e a quantidade de leitura de páginas necessárias para retornar a consulta foi bem menor, e o tempo foi de apenas 1 segundo.



Também conseguimos visualizar que no plano de execução que desta vez o memory grant foi correto e o otimizador estimou que seriam necessários 186 MBs para executar o processo de ordenação.



Abaixo podemos visualizar outro item relacionado a performance.

Reordenar colunas dos índices cluster das tabelas <Tabela> e <Tabela> para:

ALTER TABLE <Tabela> DROP CONSTRAINT PK

GO

ALTER TABLE <Tabela> ADD CONSTRAINT PK PRIMARY KEY CLUSTERED (SymbolCode, OriginId, TickId, UpdateTimestamp)

GO

ALTER TABLE <Tabela> DROP CONSTRAINT PK\_<Tabela>

GO

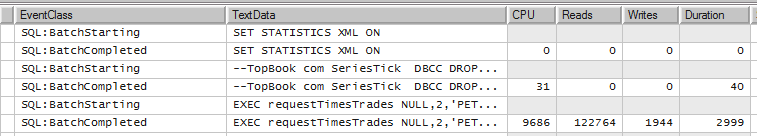
ALTER TABLE <Tabela> ADD CONSTRAINT PK\_<Tabela> PRIMARY KEY(SymbolCode, OriginId, TimePoint, TickId, PartitionCol)

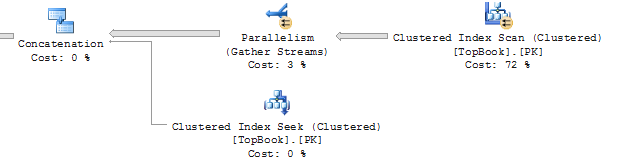
WITH (DATA\_COMPRESSION = ROW) ON spPerDay(PartitionCol)

Alterando a ordem das colunas conseguimos um seek nas consultas pois parâmetro TickID que nem sempre é especificado não é a primeira coluna do índice.

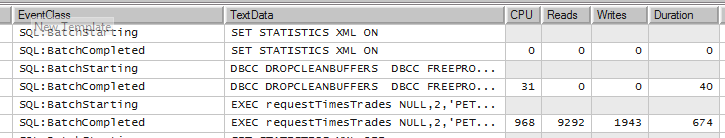
Segue um exemplo da diferença de performance e plano de execução da execução da procedure RequestTimeTrades

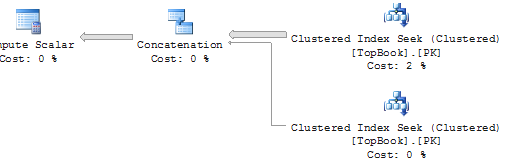
Antes:





Com o índice alterado:





O número de leituras e tempo de resposta diminuiu consideravelmente, e o plano de execução não esta mais fazendo o scan na <Tabela>, que pode ser extremamente traumático quando a tabela conter todas as linhas que estimamos que terá.

### Particionamento (expurgo de dados da <Tabela>)

Atualmente o processo de expurgo dos dados das tabelas <Tabela> e <Tabela> é efetuado com deletes nas tabelas. Isso pode causar alguns problemas como:

* Lock esclation: Durante o delete o SQL Server pode obter um lock exclusivo na tabela toda que esta sendo excluída caso o número de linhas excluídas seja muito grande (aproximadamente mais de 3 mil).
* LOG: O arquivo de log pode ficar muito grande podendo ocupar todo o espaço em disco disponível.
* LOG 2: Escrita massiva no arquivo de log, pois todas as linhas removidas são logadas.

Para evitar os problemas mencionados acima e agilizar o processo de exclusão dos dados sugerimos criar um particionamento por dia nas tabelas e utilizar o comando de SWITCH para mover os dados para outra tabela e posteriormente dropar, ou efetuar um truncate na tabela.

Segue abaixo um exemplo do processo de exclusão dos dados utilizando particionamento:

|  |  |
| --- | --- |
| **Passo 1**  ***Passo opcional***: Aumentar a quantidade de partições para 15 mil para suportar o controle de particionamento por dia.  Outra opção é criar as partições conforme necessidade, isso requer um job para manutenção das partições. Preferencialmente opte pela opção de sliding window, ou seja, crie as partições conforme demanda.  -- Aumentar suporte para número de partições para 15k  EXEC sp\_db\_increased\_partitions '<BANCO>', 'TRUE' | |
| **Passo 2**  Criação da partition scheme e partition function.  O código abaixo supõe que uma coluna do tipo Integer irá retornar as datas no formato yyyymmdd. Opcionalmente pode se utilizar uma coluna do tipo Date() ou então DateTime, porém no caso do DateTime os ranges devem ser definidos corretamente de acordo com a hora (ex: de ‘20100101 00:00:00.000’ até ‘20100101 23:59:59.997’)  -- Partição inicia em (2012-01-01) e vai até (2030-01-01)  -- Total de 6578 partições  /\*  DROP PARTITION SCHEME spPerDay  DROP PARTITION FUNCTION pfPerDay  \*/  DECLARE @IntegerPartitionFunction nvarchar(max)  SET @IntegerPartitionFunction = N'CREATE PARTITION FUNCTION pfPerDay (Int) AS RANGE LEFT FOR VALUES (';  DECLARE @i Date = '20120101';  WHILE @i <= '20300101'  BEGIN  PRINT CONVERT(NVarChar, @i, 112)  SET @IntegerPartitionFunction += CONVERT(NVarChar, @i, 112) + N', ';  SET @i = DATEADD(day, 1, @i);  END  PRINT @IntegerPartitionFunction  SET @IntegerPartitionFunction += CAST(@i as nvarchar(10)) + N');';  EXEC sp\_executesql @IntegerPartitionFunction;  GO  CREATE PARTITION SCHEME spPerDay AS PARTITION pfPerDay ALL TO ("PRIMARY")  GO | |
| **Passo 4**  Marcar a tabela para utilizar o particionamento.  ALTER TABLE <Tabela> ADD CONSTRAINT PK\_<Tabela> PRIMARY KEY(SymbolCode, OriginId, TimePoint, TickId, PartitionCol)  WITH (DATA\_COMPRESSION = ROW) ON spPerDay(PartitionCol)  GO  Obs: Todos os índices devem estar alinhados com a partição, ou seja, todos os índices devem ser re-criados para utilizar a mesma partition scheme. | |
| **Passo 5**    SeriesTickTMP | **Passo 5**  Criar uma tabela com o mesmo esquema da tabela <Tabela> para receber os dados da partição que será excluída.  IF OBJECT\_ID('<Tabela>TMP') IS NOT NULL  DROP TABLE <Tabela>TMP  GO  SELECT \*  INTO <Tabela>TMP  FROM <Tabela>  WHERE 1=0 |

|  |  |
| --- | --- |
| **Passo 6**  SeriesTickTMP  SWITCH  SeriesTickTMP | **Passo 6**  Faz o SWITCH da partição que será excluída para a tabela temporária e apaga a tabela  ALTER TABLE <Tabela> SWITCH PARTITION 1 TO <Tabela>TMP PARTITION 1  GO  -- Apaga a tabela com os dados  IF OBJECT\_ID('<Tabela>TMP') IS NOT NULL  DROP TABLE <Tabela>TMP  Opcionalmente, pode ser feito um loop para identificar todas as partições que devem ser excluídas e o código para o SWITCH pode ser dinâmico. Por ex:  DECLARE @Tab TABLE (ID Int PRIMARY KEY)  DECLARE @i Int = 0, @Str NVarChar(MAX) = ''  DECLARE @DtBase DateTime = '20120402'  -- CREATE INDEX ixTimePoint ON <Tabela> (TimePoint) WITH(DATA\_COMPRESSION = ROW) ON spPerDay(PartitionCol)  INSERT INTO @Tab(ID)  SELECT DISTINCT $PARTITION.pfPerDay(PartitionCol)  FROM <Tabela>  WHERE TimePoint < CONVERT(Date, DateAdd(day, -8, @DtBase)) -- ultimos 7 dias  SELECT TOP 1 @i = ID  FROM @Tab  WHERE ID > @i  ORDER BY ID  WHILE @@RowCount > 0  BEGIN  SET @Str = 'ALTER TABLE <Tabela> SWITCH PARTITION ' + CONVERT(VarChar, @i) + ' TO <Tabela>TMP PARTITION ' + CONVERT(VarChar, @i)  BEGIN TRY  PRINT @Str  --EXEC sp\_executesql @Str;  END TRY  BEGIN CATCH  SELECT ERROR\_NUMBER() AS ErrorNumber,  ERROR\_SEVERITY() AS ErrorSeverity,  ERROR\_STATE() AS ErrorState,  ERROR\_PROCEDURE() AS ErrorProcedure,  ERROR\_LINE() AS ErrorLine,  ERROR\_MESSAGE() AS ErrorMessage;  BREAK;  END CATCH  SELECT TOP 1 @i = ID  FROM @Tab  WHERE ID > @i  ORDER BY ID  END  -- Apaga a tabela com os dados  IF OBJECT\_ID('<Tabela>TMP') IS NOT NULL  DROP TABLE <Tabela>TMP |

Utilizando o conceito exibido acima o processo de exclusão dos dados será muito mais rápido, além de não causar nenhum dos problemas mencionados acima.

### Particionamento (remover particionamento atual)

Atualmente as tabelas que sofrerão carga massiva estão particionadas por uma coluna calculada. A intenção deste particionamento era separar os dados das tabelas entre os vários discos disponíveis e com isso ganhar em performance, tanto na escrita quanto na leitura dos dados.

Para obter o ganho desejado, não é necessário o uso de particionamento, basta criar vários arquivos e aponta-los para o mesmo filegroup, porém os arquivos estarão em discos diferentes. Os arquivos devem ser criados com o mesmo tamanho inicial. Desta forma o SQL Server divide os dados entre todos os arquivos e controla o crescimento dos dados de forma distribuída.

Recomendamos remover o particionamento atual e utilizar o conceito mencionado acima.

## Configurações gerais

### Recovery interval para 0

Recomendamos utilizar o valor padrão para as seguintes configurações:

* Recovery interval para 0
* Cost Threshold for Parallelism para 5
* Network Packet Size 4096
* Min memory per query 1024

### TEMPDB

Criar mais de um arquivo no tempdb para evitar contenção nas páginas de controle de alocação.

## Configurações Hardware

### HyperThreading

Como o processo de inserção dos dados será em paralelo, CPU tende a ser um gargalo, por este motivo recomendamos deixar o HT habilitado. Durante os testes efetuados no ambiente obtivemos um ganho de aproximadamente 50% no número de linhas inseridas quanto HT habilitado.