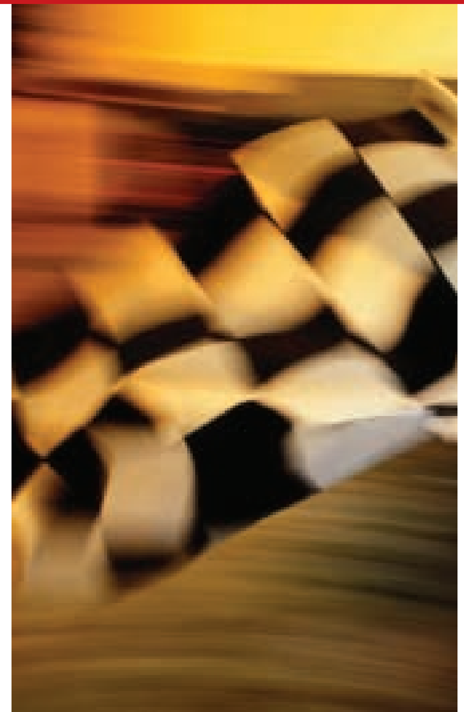

Boxenstopp

Optimale Performance mit SQL Server 2000 – Teil 2

von Urs Gehrig

Sie erinnern sich: Im ersten Teil dieses Artikels haben wir gesehen, dass SQL Server 2000 in der Top Ten Performanceliste von TPC ganz zuoberst steht. Wir lernten, wie wir das Optimale aus dem Server holen. In diesem zweiten Teil konzentrieren wir uns auf den Client. Dabei interessieren uns Fragen wie: „Wie nutze ich ADO.NET am effizientesten“ oder „Wie formuliere ich effiziente T-SQL-Batches“. Zum Schluss machen wir noch einen Griff in die Toolbox des SQL Servers und schauen, welche Tools uns Microsoft für die Performanceoptimierung zur Verfügung stellt. Danach sollte dann hoffentlich jeder Ansatzpunkte für seine Applikationsoptimierung gefunden haben.

Damit keine Verwechslungen aufkommen: Mit Client meine ich hier nicht zwingend das GUI für den Benutzer. Der Client, von dem wir hier sprechen, ist der Teil unserer Applikation, welcher direkt mit der Datenbank kommuniziert. Das kann zwar direkt das GUI sein, genauso gut kann es aber auch ein Service, die Business-Tier unserer n-Tier-Architektur, der DB-Layer oder was auch immer sein. Einziges Kriterium ist: Die Software greift auf unseren SQL Server zu.

Damit wir ein paar Beispiele durchgehen können, sollten Sie zuerst das Skript in Listing 1 ausführen. Dieses kopiert die *Customer*-Tabelle der *Northwind*-Datenbank und vervielfacht die Daten darin. Seien Sie aber bitte geduldig. Auf meinen Laptop braucht das Skript ungefähr 60 Minuten; dafür erhalten wir aber auch eine Tabelle mit ungefähr 9,1 Millionen Einträgen!

ADO.NET

Fangen wir nun an, mit ADO.NET unseren Client aufzubauen und wählen wir gleich zu Beginn den richtigen Namespace. Da wir unsere Applikation auf Performance trimmen wollen, verzichten wir auf jegliche generische Ansätze. Unsere Applikation läuft nur mit SQL Server zusammen, d.h. wir wählen den Namespace *System.Data.SqlClient*. Als nächsten Schritt müssen wir eine Connection zur Datenbank aufbauen. Connections sind Heavy Objects, welche eine Menge Systemressourcen benötigen – und dies nicht nur zum Zeitpunkt der Erstellung. ADO.NET kennt darum das Prinzip des Connection-Poolings. Wird eine Connection nicht mehr benötigt, bricht man die Verbindung zum Server nicht einfach ab, sondern gibt diese an einen Pool zurück, wo sie für eine nächste Anfrage bereits geöffnet und initialisiert zur Verfügung steht. Eigentlich müssen wir gar nichts Besonderes tun, denn Connection Pooling tritt in ADO.NET automatisch und per Default in Aktion. Mit *Connection.Open()* holen wir eine Verbindung aus dem Pool und mit *Connection.Close()* treten wir diese wieder an den Pool ab.

Auf ein, zwei Dinge sollten wir unser Augenmerk dennoch richten, da ansonsten das ganze Pooling nicht so recht zum Laufen kommt. Unsere Connection-Strings müssen immer identisch formuliert sein, da ansonsten ADO.NET einen neuen Pool anlegt und wir von den bereits initialisierten Connections des anderen Pools nicht profitieren können. Mit einem identisch formulierten Connection-String meine ich einen wirklich identischen. Achten Sie also auf gleichbleibende Groß- und Klein-Schreibung, die Reihenfolge der Parameter etc. In der Regel bereitet uns diese Forderung allerdings kaum Schwierigkeiten, haben wir den Connection-String doch irgendwo zentral generiert und abgelegt, von wo wir diesen immer wieder referenzieren. Etwas mehr Mühe müssen wir uns bei der nächsten Forderung geben: Die Connection soll so spät wie möglich geöffnet und so früh wie möglich wieder geschlossen werden. Nur so können wir gewährleisten, dass möglichst immer eine freie Connection im Pool vorhanden ist und somit bei einer Anfrage keine neue erzeugt werden muss. Na ja, mit ein wenig Disziplin auf unserer Seite sollten wir auch das noch hinkriegen.

Zeit für ein Beispiel, anhand dessen wir dann gleich noch ein paar weitere Dinge anschauen wollen. Den Quellcode für das ganze Beispiel finden Sie auf der beiliegenden Heft-CD. Das Ganze sieht in etwa wie in Abbildung 1 aus. Vorerst interessiert uns aber nur die Subroutine *ExamPaging* (siehe Listing 2).

© Software & Support Verlag

Die ersten drei Schritte können wir hier gut nachvollziehen. Erstens, wir verwenden den *System.Data.SqlClient*-Namespace; zweitens, die Connection wird mit einem konstanten String *CONNECTION_STRING* instanziert und drittens wird diese so kurz wie möglich offen gehalten. Das Öffnen und Schließen der Connection haben wir sogar .NET überlassen: Der *SqlDataAdapter* öffnet und schließt diese selbständig. Kürzer kann die Connection nicht mehr geöffnet sein.

Eine immer wiederkehrende Problematik ist das seitenweise Laden und Anzeigen von Daten. Möchten wir dem Benutzer alle Zeilen einer Tabelle zur Auswahl anzeigen, wenn diese Tabelle Tausende und Abertausende von Zeilen besitzt, dann ist das nicht ganz so einfach. Wir können unmöglich alle Zeilen auf einmal vom Server lesen und diese insgesamt, z.B. in einem Datagrid, visualisieren. Hierfür müssen wir in unsere Trickkiste greifen und die Daten nur seitenweise laden und visualisieren. Dabei spricht man auch von Paging. Auf den ersten Blick scheint uns hierfür eine der diversen überladenen *DataAdapter.Fill()*-Methoden passend zu sein. Es existiert nämlich eine *Fill*-Methode, welche die Argumente *startRecord* und *maxRecords* kennt, also genau das, was wir zum Pagen benötigen. Wie die bereits vorgängig gezeigte Subroutine *ExamPaging* zeigt, taugt diese Variante der *Fill()*-Methode leider überhaupt nichts. Nur schon das Laden der zweiten 100 Zeilen unserer *Customer*-Tabelle löst eine *Timeout expired*-Exception aus. Zimmern wir aber unser Paging-SQL-Statement mit verschachtelten *SELECT TOP*-Statements selbst zusammen, dann ist die ganze Abfrage bereits in einem Bruchteil einer Sekunde abgearbeitet. Warum dies so ist, schauen wir uns später an.

Eine ebenfalls ständig wiederkehrende Performancefalle bilden Transaktionen. Während der Laufzeit einer Transaktion sind die darin geänderten Daten gelockt und parallel dazu verlaufende Transaktionen, welche dieselben Daten referenzieren, werden geblockt. Blocking verschlechtert aber ganz entscheidend unsere Performance. Wir müssen also darauf achten, dass all unsere Transaktionen so kurz wie möglich sind. So genannte Long-Running Transactions sind unter allen Umständen zu vermeiden. Aber wie können wir unsere Transaktionen kurz halten? Da gibt es gleich mehrere Angriffspunkte, welche leider allzu oft übersehen werden. Während einer laufenden Transaktion haben Benutzereingaben nichts zu suchen! Wenn wir eine Transaktion starten und vor dem *Commit*-Statement den Benutzer fragen, ob er diese Änderungen wirklich und endgültig speichern will, dann können wir sicher sein, dass sich der Benutzer genau zu diesem Zeitpunkt einen Kaffee holt, im Gang noch einen kurzen Schwatz abhält und unsere Frage erst in einer halben Stunde beantworten wird – eine halbe Stunde, in der die Daten gelockt sind, das System blockiert und unsere Performance gleich Null ist. Dasselbe gilt auch für die Validierung der Daten. Die Daten werden vollständig validiert und berechnet noch bevor die Transaktion gestartet wird.

Natürlich gibt es immer mal Situationen, wo sich Long-Running Transactions nicht vermeiden lassen. Da müssen wir eben schauen, dass die hierzu parallel verlaufenden Abfragen nicht blockieren. Eine Möglichkeit hierzu bildet der Locking Hint *READPAST*. *READPAST* verhindert, dass ein *SELECT*-Statement blockiert, indem einfach alle gelockten Daten übersprungen und nicht zurückgegeben werden. Es gibt durchaus Situationen, in welchen dies erlaubt sein und einen Sinn

Listing 1

```
-- Erstellen von 'PerfTestDB' und abfüllen mit Daten.
-- Diese DB wird als Test-Datenbank für unsere
                              Performancetests gebraucht.
--
-- Urs Gehrig, Feburar 2004, bbv Software Services AG

-- DB anlegen
CREATE DATABASE PerfTestDB
GO
USE PerfTestDB
GO

-- Tabelle 'Customers' anlegen
SELECT IDENTITY(int, 1,1) AS ID, 0 AS Copy, *
INTO Customers
FROM Northwind.dbo.Customers
GO

-- Tabelle 'Customers' mit Daten abfüllen
DECLARE @i AS int
SET @i = 1
WHILE @i <= 100000
BEGIN
 BEGIN TRANSACTION
 INSERT Customers
 SELECT @i, *
 FROM Northwind.dbo.Customers
 COMMIT
 SET @i = @i + 1
END
GO

-- Indizes für Tabelle 'Customers' anlegen
ALTER TABLE dbo.Customers ADD
 CONSTRAINT PK_Customers PRIMARY KEY CLUSTERED (ID)
GO

CREATE INDEX IX_Customers ON dbo.
                                  Customers(CustomerID)
GO
```

Listing 2

```
Private Sub ExamPaging()
 Dim connTestDB As SqlConnection =
          New SqlConnection(CONNECTION_STRING)
 Dim dsLeft As DataSet = New DataSet
 Dim daLeft As SqlDataAdapter = _
    New SqlDataAdapter("SELECT ID, Copy,
         CustomerID FROM Customers", connTestDB)
 Dim dsRight As DataSet = New DataSet
 Dim daRight As SqlDataAdapter = _
    New SqlDataAdapter("SELECT * FROM
                 (SELECT TOP 100 * FROM " & _
         "(SELECT TOP 200 ID, Copy,
               CustomerID FROM Customers " & _
         "ORDER BY id ASC) AS Customers " & _
         "ORDER BY id DESC)
               AS Customers ORDER BY id ASC", _
         connTestDB)

 'schlechte Lösung
 Try
  'erwartetes Verhalten: "Timeout expired" Exception
  daLeft.Fill(dsLeft, 100, 100, "Customers")
  dgLeft.DataSource = dsLeft.Tables(0)
  dgLeft.Update()
 Catch ex As SqlException
  MsgBox(ex.Message, MsgBoxStyle.
     Critical + MsgBoxStyle.OKOnly, "SQL Exception")
 End Try

 'gute Lösung
 Try
  'erwartetes Verhalten: korrektes und rasches Resultat
  daRight.Fill(dsRight)
  dgRight.DataSource = dsRight.Tables(0)
  dgRight.Update()
 Catch ex As SqlException
  MsgBox(ex.Message, MsgBoxStyle.Critical +
          MsgBoxStyle.OKOnly, "SQL Exception")
 End Try
End Sub
```

© Software & Support Verlag

ergeben kann. Zum Beispiel in einer Ticketing-Applikation. Wenn gerade ein Kundenberater einen Sitzplatz für die nächste Vorstellung am Buchen ist, dann darf eine Freiplatzanfrage eines zweiten Kundenberaters den zurzeit gelockten Sitzplatz durchaus überspringen und nur noch alle anderen freien anzeigen. Wie das Ganze funktioniert sehen wir wiederum in unserer Beispielapplikation mit der Subroutine *ExamLongRunningTransaction*. Eine Alternative zum Locking Hint *READPAST* finden Sie im Kasten „Snapshot Isolation Level“.

Effiziente T-SQL-Batches

Vermeiden wir *SELECT* *-Statements! Das sind die großen Performancefallen für die Zukunft. Heute kann dieses Statement zwar noch legal sein, weil wir wirklich alle Spalten benötigen. Wie sieht es aber morgen damit aus? Vielleicht fügt der DBA bereits morgen noch eine Spalte *Photo* in unserer *Customers*-Tablle hinzu und füllt diese für eine andere Applikation mit Bergen von JPEG-Files. Unsere Applikation kann diese Informationen gar nicht verarbeiten. Warum sollten wir also diese Fülle von Daten überhaupt erst abholen?

In der letzten Ausgabe haben wir gesehen, dass wir für unsere Tabellen Indi-

Snapshot Isolation Level

In unserem Beispiel Long-Running Transaction haben wir gesehen, wie ein Update einer Tabelle ein *SELECT*-Statement blockieren kann. Wenn eine Transaktion eine Zeile ändert, kann eine weitere Transaktion die selbe Zeile so lange nicht lesen, bis das Update vollständig beendet wurde, d.h. ein Commit oder Rollback vollzogen wurde. *Read Uncommited Isolation Level* schafft hier Abhilfe, wenngleich das Ganze auch sehr gefährlich ist. Die zweite Transaktion wird nicht blockiert und die *uncommited* Werte werden zurückgegeben. Mit dem Locking Hint *READPAST* haben wir einen weiteren Workaround gefunden, der das Blockieren des *SELECT*-Statements verhindert, indem die gelockten Daten einfach übersprungen werden. Mit diesen beiden Workarounds haben wir schon beinahe alle Situationen im Griff.
Aber da war doch noch was, was wir von Oracle her schon lange kennen? Richtig! Standardmäßig blockiert Oracle in solch einer Situation nicht; es bringt einfach den alten Wert der Daten zurück, d.h. denjenigen Wert, den die Daten im Rollback-Fall wieder erhalten würden. Bei Oracle nennt man dies *old and new row value*. Leider kennt SQL Server 2000 diese Möglichkeit genauso wenig wie etwa DB2. Ein Silberstreifen am Horizont ist allerdings auszumachen. Mit Yukon, der nächsten Version von SQL Server 2000, welche zurzeit als Beta 1 vorliegt, gehört dieses Manko der Vergangenheit an. Snapshot Isolation Level von Yukon verhält sich identisch zum standardmäßigen *old and new row value*-Mechanismus von Oracle. Damit wird ein Migrieren von Oracle-Applikationen auf SQL Server noch einfacher.

© Software & Support Verlag

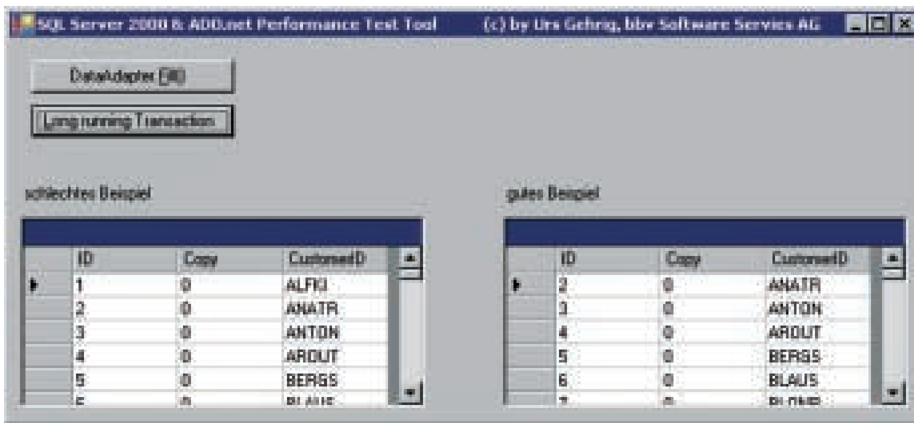

Abb. 1: SQL Server 2000 & ADO.NET Performance Test Tool

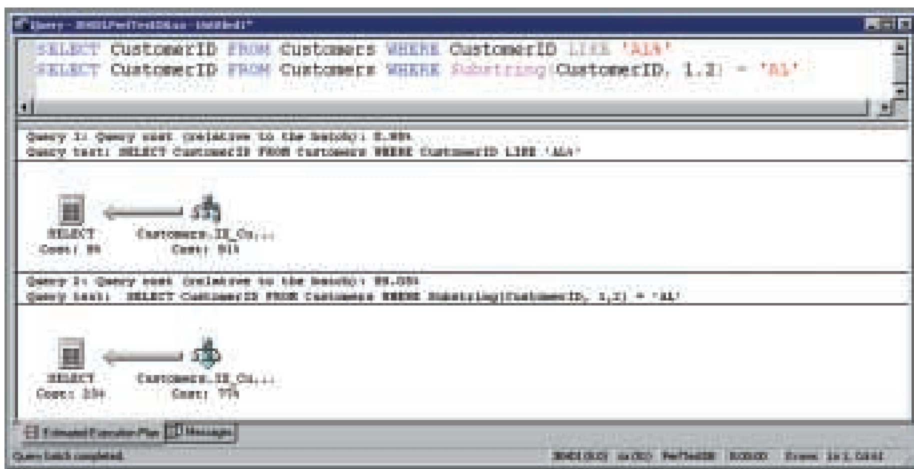

Abb. 2: Estimated Execution Plan im SQL Query Analyzer

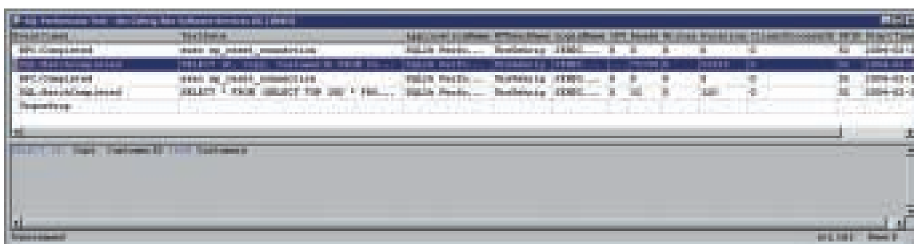

Abb. 3: *DataAdapter.Fill*-Trace im SQL Profiler

zes anlegen müssen. Nun gibt es da aber noch Bedingungen, damit SQL Server diese Indizes auch wirklich verwenden kann. Die wichtigste dabei ist wohl, dass die *WHERE*-Klausel *SARGable* sein muss. SARGable *WHERE*-Klauseln sind diejenigen, bei welchen die Suchkriterien mit Hilfe eines Index nachgeschlagen werden können. Hört sich kompliziert an. Hier zwei Beispiele:

```
'Beispiel 1:
' gut
SELECT CustomerID FROM Customers WHERE
                    CustomerID LIKE 'AL%'
' schlecht
SELECT CustomerID FROM Customers WHERE Substring
                    (CustomerID, 1,2) = 'AL'
'Beispiel 2:
' gut
SELECT CustomerID FROM Customers WHERE
                    CustomerID = 'alfki'
' schlecht; und unnötig,
        da per SQL Server per dafault case insensitive ist
SELECT CustomerID FROM Customers WHERE Lower
                    (CustomerID) = 'alfki'
```

Für jedes Beispiel liefern die beiden *SELECT*-Statements zwar dasselbe Resultat, aber mit einem immensen Unterschied in der Performance. Generell lässt sich sagen, dass Funktionen, welche eine Spalte als Argument haben, nicht SARGable sind. Hier tun wir gut daran, nach Alternativen zu suchen. Vielleicht finden wir diese Alternativen auch bei der Erfassung der Daten, sodass die Abfragen anschließend gar keine Funktionen mehr benötigen.

SQL Server Performance Toolbox

Welche Hilfsmittel stehen uns für die Optimierung der Performance zur Verfügung? Da ist einmal der SQL Query Analyzer. Hiermit können wir unter anderem den Execution Plan und die Query Cost für jedes Statement innerhalb eines T-SQL-Batches anzeigen lassen. Schauen wir uns das Ganze am Beispiel 1 von oben an. Da wir für den schlechten Fall eine recht lange Verarbeitungszeit erwarten, wollen wir das Statement nicht wirklich ausführen lassen, sondern begnügen uns mit dem Estimated Execution Plan. Geben wir also die beiden *SELECT*-Statements ins Fenster für den Querytext ein und lassen wir uns den Estimated Query Plan anzeigen (*Ctrl+L*). Das sieht dann in etwa wie in Abbildung 2 aus.

Im Execution Plan sehen wir sehr schön, dass im Falle des ersten *SELECT*-Statements ein performanter Index Seek durchgeführt wird, währendem das zweite Statement lediglich mit einem Index Scan durchgeführt werden kann. Hierin liegt auch der große Performanceunterschied begründet. Interessant zu sehen ist aber auch die Tatsache, dass der Index Scan trotzdem den Index *IX_Customers* und nicht den Clustered Index verwendet. Warum? Ganz einfach: Für unser *SELECT*-Statement bildet dieser Index einen Covered Index, d.h. die Query kann alleine mit Hilfe des Indexes und ohne Nachschlagen im Clustered Index durchgeführt werden. Da der ganze Index *IX_Customers* aber mit wesentlich weniger Diskzugriffen gelesen werden kann als der Clustered Index, bringt die Verwendung des Indexes immer noch eine, wenn auch bescheidene, Performancesteigerung mit sich.

Ein weiteres wertvolles Tool für die Performance-Optimierung ist der SQL Profiler. Mit seiner Hilfe können wir alle uns interessierenden Aktivitäten des SQL Servers aufzeichnen. Für unsere Performanceoptimierung konzentrieren wir uns

© Software & Support Verlag

im Wesentlichen auf T-SQL-Batches, RPC-Calls, Stored Procedures, Transaktionen, Lockings und dergleichen. Mit dem online verfügbaren Trace Template *SQL Performance.tdf* haben wir einen guten Einstiegspunkt für unsere ersten Analyseschritte. Starten wir den SQL Profiler mit diesem Template und nehmen wir wieder unsere Beispielapplikation zur Hand. Wenn wir unser Beispiel für die *DataAdapter.Fill*-Methode ausführen, erhalten wir einen Trace in etwa wie in Abbildung 3.

Jetzt sehen wir sehr schön, warum die Paging-Lösung mit *DataAdapter.Fill* nicht zum Erfolg führte: ADO.NET fordert die ganze Tabelle vom Server an und nimmt das Paging clientseitig vor. Kein Wunder also, dass eine *Timeout expired*-Exception ausgelöst wird. Wir wollten lediglich 100 Zeilen in unserem DataGrid anzeigen, ADO.NET forderte dafür aber Millionen von Zeilen vom Server an. Das kann nun wirklich keinen Sinn machen. Quintessenz daraus: Paging überlassen wir nie der *Fill*-Methode des *DataAdapter*-Objekts.

Wollen wir uns eine Übersicht über den Zustand des Servers insgesamt verschaffen, können wir uns hierfür des Windows Performance Monitors bedienen. Informationen über Memory, Netzwerk, Disk, Prozessor oder gar SQL Server als Gesamtes liefern wertvolle Hinweise zu Engpässen in den Ressourcen. Abbildung 4 zeigt ein Beispiel, wie so etwas aussehen könnte. Im Beispiel sehr schön zu sehen ist, wie nach dem ersten *DataAdapter.Fill*-Test (erster Peak in *Avg. Disk Queue Length*) *Available MBytes* praktisch auf Null fällt. SQL Server versucht hier, die Millionen von Zeilen aus der *Customer*-Tabelle in den Speicher zu laden. Dies kommt uns beim zweiten *DataAdapter.Fill*-Test (zweiter Peak in *Avg. Disk Queue Length*) zugute; die Festplatte wird weniger stark beansprucht, weil der SQL Server jetzt die Daten aus dem Buffer liest.

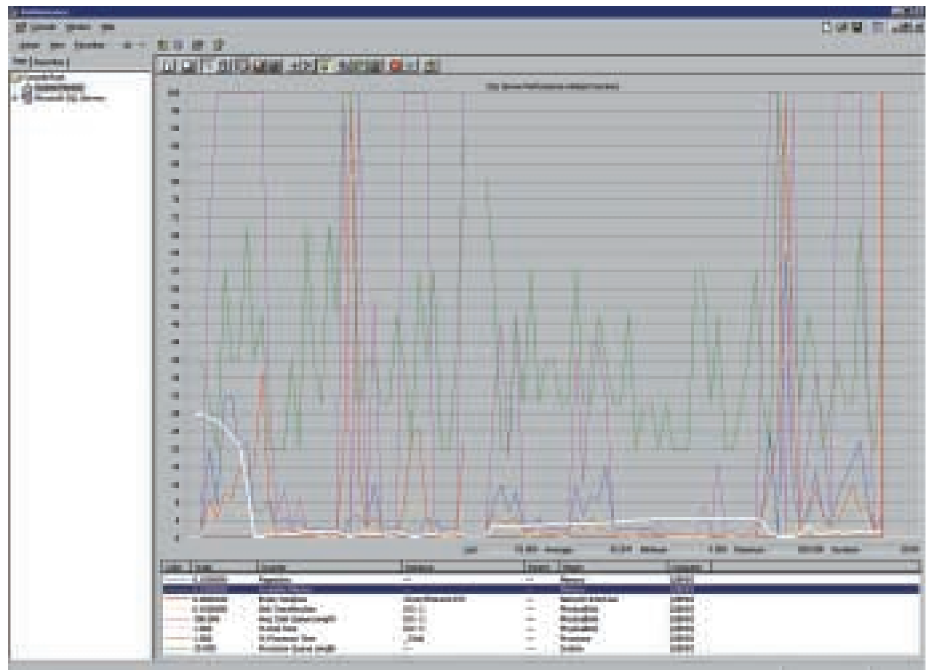

Abb. 4: Windows Performance Monitor

Dokumentieren ist das A und O

Unsere Applikation wird von einer Stunde auf die andere langsamer. Woran mag das nur liegen? Wir gehen der Sache auf den Grund und beschaffen uns mit dem SQL Query Analyzer, SQL Profiler und Windows Performance Monitor einen Einblick in das Geschehen. Und nun? Wo ist die Ursache für den Performanceverlust zu suchen? Tja, ohne gute Vorbereitung lässt sich diese Frage kaum beantworten. Gute Vorbereitung heißt: Wir sind dokumentiert. Nachdem unsere Applikation fertig entwickelt und auf dem Produktivsystem installiert ist, machen wir die ersten Performancetests und dokumentieren diese. Dieselben Tests wiederholen wir, wenn die Applikation produktiv läuft und wir mit der Performance zufrieden sind. Die Testbedingungen und Resultate dokumentieren wir erneut. Im Sinne von „Vorbeugen ist besser als Heilen“ wiederholen wir die Tests periodisch und versuchen so, einen Trend für einzelne Performancecounter festzustellen. Werden z.B. die Disks immer voller, lässt sich rechtzeitig deren Kapazität ausbauen, bevor das Ganze zu einem Problem führt. Und wenn wir dann doch zu einem Peformanceproblem kommen, können wir die aktuellen Testresultate mit unserer Dokumentation vergleichen und finden so diejenigen Performancecounter, welche uns davonlaufen. Haben wir diese erst einmal aufgedeckt, lässt sich einfacher nach dem Verursacher des Problems suchen. Nehmen zum Beispiel die *Full Scan/sec*-Werte des SQL Servers dramatisch zu, dann haben wir vielleicht ein paar Indizes verloren, die Auto-Statistik ausgeschaltet oder die typischen Anfragen haben sich plötzlich verändert und wir müssen zusätzliche Indizes bilden.

Schluss

So, das war's. Ich hoffe dass nun jeder von Ihnen ein, zwei Ansatzpunkte zur Optimierung seiner Applikation gefunden hat. Nein? Dann habe ich nur noch folgende drei Antworten für Sie: Erstens, Ihre Applikation läuft schon performant. Gratulation! Zweitens, studieren Sie ein wenig Hardcore direkt aus Microsofts SQL Server customer lab [1]. Drittens, schreiben Sie mir eine eMail mit Ihrem Problem. Vielleicht entsteht daraus gar ein Teil 3 dieser Artikelfolge, den Sie dann wieder hier im *dot.net magazin* lesen können.

Urs Gehrig ist Head of System Services der bbv Software Services AG. Dieser Service hat sich auf die kundenorientierte Softwareentwicklung mit .NET spezialisiert. Er selber ist seit über 15 Jahren als Entwickler von Windows-Applikationen und EDV-Berater tätig. Seine besonderen Steckenpferd sind Datenbanken und Security. Sie erreichen ihn unter urs.gehrig@bbv.ch.

Links & Literatur

[1] Tom Davidson, „Opening Microsoft's Performance-Tuning Toolbox“: www.sqlmag.com/Articles/Index.cfm?ArticleID=40925&pg=1

© Software & Support Verlag