net.com
networking & communications

HOLGER SCHMELING

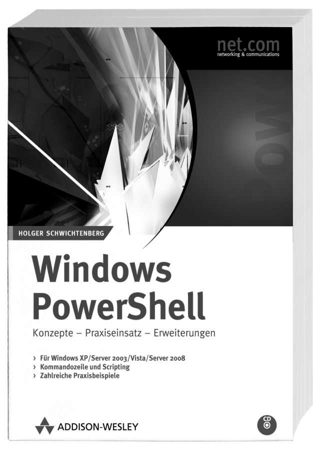
SQL Server 2008 Performanceoptimierung

Das Praxisbuch für Entwickler und Administratoren

SQL Server 2008
Performance-Optimierung

net.com
networking & communications

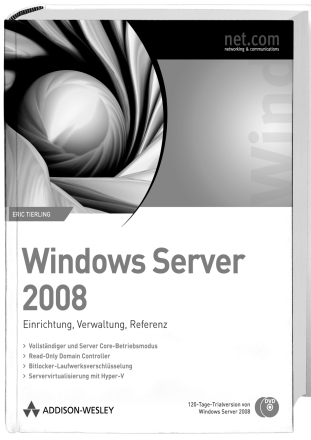
Netzwerke, Betriebssysteme, Sicherheit ... hierzu bietet Ihnen die Reihe net.com umfassende, praxisnahe Information. Neben Fragen der Systemverwaltung greift sie auch Themen wie Protokolle, Technologien und Tools auf. Profitieren Sie bei Ihrer täglichen Arbeit vom Praxiswissen unserer erfahrenen Autoren.



Windows PowerShell

Holger Schwichtenberg
456 Seiten, € 29,95 [D]
ISBN 3-8273-2533-4

Mit der neuen Windows PowerShell lassen sich Aufgaben bequem per Kommandozeile automatisieren. Der bekannte Scripting-Experte Dr. Holger Schwichtenberg bietet mit diesem Buch eine fundierte Einführung in die automatisierte Windows-Administration.



Windows Server 2008

Eric Tierling
1184 Seiten, € 59,95 [D]
ISBN 3-8273-2637-9

Dieses Buch zu Windows Server 2008 knüpft an den Bestseller zu Windows Server 2003 an und widmet sich eingehend den Neuerungen der 2008er-Serverversion. Erstkonfiguration, Rollen und Features, überarbeiteter Server-Manager, Server Core-Installationsoption, BitLocker-Laufwerksverschlüsselung, Read-Only-Domänencontroller (RODC), Netzwerkzugriffsschutz (NAP), RemoteApp-Programme für die Terminaldienste, Failover-Clustering sowie die Servervirtualisierung mittels der neuen Technologie Hyper-V sind einige der Highlights, die im Buch beschrieben sind.

Holger Schmeling

SQL Server 2008 Performance-Optimierung

Das Praxisbuch für Entwickler und Administratoren

eBook
Die nicht autorisierte Weitergabe dieses eBooks an Dritte ist eine Verletzung des Urheberrechts!

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen.
Trotzdem können Fehler nicht vollständig ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien. Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

Fast alle Hard- und Softwarebezeichnungen und weitere Stichworte und sonstige Angaben, die in diesem Buch verwendet werden, sind als eingetragene Marken geschützt.
Da es nicht möglich ist, in allen Fällen zeitnah zu ermitteln, ob ein Markenschutz besteht, wird das ®-Symbol in diesem Buch nicht verwendet.

Umwelthinweis:
Dieses Buch wurde auf chlorfrei gebleichtem Papier gedruckt.

10 9 8 7 6 5 4 3 2 1

11 10 09

ISBN 978-3-8273-2778-9

© 2009 by Addison-Wesley Verlag,
ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany
Alle Rechte vorbehalten
Einbandgestaltung: Marco Lindenbeck, mlindenbeck@webwo.de
Fachlektorat: Wim Nienkerke, wim.nienkerke@netcologne.de
Lektorat: Sylvia Hasselbach, shasselbach@pearson.de
Korrektorat: René Wiegand, info@wiegand-dokumentation.de
Herstellung: Claudia Bäurle, cbaeurle@pearson.de
Satz: mediaService, Siegen, www.media-service.tv
Druck und Verarbeitung: Bercker Graphischer Betrieb, Kevelaer
Printed in Germany

Inhaltsverzeichnis

Vorwort 12

1 Einleitung 13

1.1 Optimierungsmodell 14
1.2 Was vermittelt dieses Buch? 15
1.2.1 Tipps und Hinweise 16
1.3 Wo finde ich was? 16
1.4 Welche Voraussetzungen werden benötigt? 18
1.5 Was ist auf der Begleit-CD? 18
1.6 Danksagung 19
1.7 Performanz oder Performance? 19

Teil 1 Grundlagen 21

2 Datenverwaltung durch SQL Server 23

2.1 Datenbanken 23
2.2 Daten lesen 24
2.3 Daten schreiben 25
2.3.1 Experiment: Transaktionsgröße 26
2.4 Zusammenfassung 29

3 Ausführung von Abfragen 31

3.1 Logische Schritte bei der Abfrageausführung 31
3.2 Physikalischer Ausführungsplan 34
3.2.1 Parser 34
3.2.2 Algebrizer 35
3.2.3 Optimierer 35
3.2.4 Anzeigen des Ausführungsplans 39
3.3 Zusammenfassung 43

4 Werkzeuge und Indikatoren zum Messen der Leistung 45

4.1 Allgemeine Werkzeuge 48
4.1.1 Messen mit der Stoppuhr 48
4.1.2 Statistische Größen 49
4.2 Der Aktivitätsmonitor 50
4.2.1 Übersicht 51
4.2.2 Aktuelle Prozesse 51
4.2.3 Ressourcenwartevorgänge 52
4.2.4 Datendatei E/A 52
4.2.5 Aktuell wertvolle Abfragen 52
4.3 Ablaufverfolgungen und der SQL Server Profiler 53
4.3.1 Erstellen einer einfachen Ablaufverfolgung 55
4.3.2 Ereignisse und Ereignisspalten 59
4.3.3 Arbeiten mit Ablaufverfolgungen 62
4.3.4 Serverseitige Ablaufverfolgungen 63
4.3.5 Arbeiten mit Ablaufverfolgungsdateien 66
4.4 Der Windows-Systemmonitor 67
4.4.1 Wichtige Leistungsindikatoren 68
4.5 Verbindung von Systemmonitor-Berichten mit Ablaufverfolgungen 78
4.6 Dynamische Verwaltungssichten 79
4.6.1 Abfrage der aktuellen Aktivität 81
4.6.2 Abfrage der E/A-Vorgänge 81
4.6.3 Abfrage der insgesamt aufgetretenen Wartezustände 82
4.6.4 Abfrage der SQL Server-Leistungsindikatoren 84
4.7 Statistische Systemfunktionen 85
4.8 Gespeicherte Systemprozeduren 87
4.9 DBCC 89
4.10 SQLdiag 90
4.11 Ausführungspläne 93
4.11.1 Wichtige Operatoren in Ausführungsplänen 94
4.11.2 Eigenschaften von Operatoren 97
4.11.3 Analyse von Ausführungsplänen 100
4.12 Datenauflistungen 105
4.12.1 Konfiguration eines Verwaltungs-Data Warehouse 106
4.12.2 Konfigurieren von Datenauflistungen 108
4.13 Berichte 112
4.13.1 Allgemeine Berichte 113
4.13.2 Berichte der Datenauflistung 118
4.14 Zusammenfassung 122

Teil 2 **Physische Aspekte des Datenbankentwurfs** 123

5 Verwenden von Indizes 125

5.1 Der Heap: Eine Tabelle ohne Index 125
5.2 Der gruppierte Index 127
5.3 Der nichtgruppierte Index auf einem Heap 130
5.4 Der nichtgruppierte Index auf einem gruppierten Index 131
5.5 Eingeschlossene Spalten 132
5.6 Gefilterte Indizes 133
5.7 Indizierte Sichten 134
5.8 Erstellen von Indizes 135
5.8.1 Manuelles Erstellen von Indizes: CREATE INDEX 136
5.8.2 Automatische Erstellung von Indizes 137
5.8.3 Indizes auf Sichten 137
5.8.4 Index-Füllfaktor 138
5.8.5 Einen Index neu aufbauen 140
5.8.6 Löschen von Indizes 141
5.9 Zusammenfassung 141

6 Verwalten von Indizes 143

6.1 Fragmentierung und Reorganisation 144
6.1.1 Einen Index reorganisieren 147
6.1.2 Einen Index neu erstellen 148
6.1.3 Strategie zur Indexprüfung und Indexdefragmentierung 149
6.2 Fehlende Indizes 151
6.2.1 Fehlende Indizes in gespeicherten Ausführungsplänen 151
6.2.2 Die sys.dm_db_missing_index...-Systemsichten 155
6.3 Überflüssige Indizes 158
6.4 Zusammenfassung 162

7 Partitionierung 163

7.1 Horizontale Partitionierung 164
7.1.1 Partitionierte Sichten 165
7.2 Vertikale Partitionierung 166
7.3 Zusammenfassung 169

8 Komprimierung von Daten 171

8.1 Allgemeines zur Komprimierung 171
8.2 Vorteile einer Komprimierung 172
8.3 Komprimierungsarten 172
8.4 Beispiel: Auswirkung der Komprimierung auf die Abfrageleistung 173
8.5 Komprimierten Speicherplatz berechnen 175
8.6 Zusammenfassung 177

Teil 3 Optimierung 179

9 Analysieren und Optimieren von Abfragen 181

9.1 Ausführungspläne und der Plancache 181
9.1.1 Kompilierung und Re-Kompilierung von Ausführungsplänen 184
9.1.2 Entfernen von Plänen aus dem Plancache 184
9.1.3 Parametrisierte Abfragen 185
9.1.4 Wiederverwendung von Abfrageplänen 186
9.2 Die Rolle von Statistiken 189
9.2.1 Erstellen und Aktualisieren von Statistiken 197
9.2.2 Probleme mit Statistiken 207
9.3 Parametrisierte Abfragen 212
9.3.1 Positive Auswirkungen der Parametrisierung 212
9.3.2 Probleme mit der Parametrisierung 218
9.3.3 Erzwungene Parametrisierung 219
9.4 Parameter Sniffing 223
9.4.1 Probleme mit Parameter Sniffing 223
9.4.2 Lösung von Parameter Sniffing-Problemen 232
9.5 Physikalische JOIN-Operatoren 242
9.5.1 MERGE JOIN 244
9.5.2 HASH JOIN 245
9.5.3 NESTED LOOPS 247
9.6 Auffinden geeigneter Indizes 248
9.6.1 Suchargumente (SARGs) 249
9.6.2 Auswahl des gruppierten Index für eine Tabelle 251
9.6.3 Selektivität und Sortierung 253
9.6.4 Verknüpfungen und Fremdschlüssel (Foreign Keys) 261
9.7 Zusammenfassung 264

10 Auffinden problematischer Abfragen **265**

10.1 Überwachung durch dynamische Verwaltungssichten **266**
10.1.1 Auswertung der E/A-Operationen **266**
10.1.2 Ermitteln fehlender Indizes **269**
10.1.3 Auswerten der im Plancache gespeicherten Ausführungspläne **269**
10.1.4 Permanentes Speichern der Informationen aus dynamischen Verwaltungssichten **272**
10.1.5 Berichte **276**

10.2 Arbeiten mit dem Profiler **277**

10.3 Einsatz von Datenauflistungen **280**
10.3.1 Manuelle Abfragen des VDWH **282**
10.3.2 Erzeugen von Ablaufverfolgungen mit dem Datenauflister **284**

10.4 Zusammenfassung **287**

11 Optimierung des physischen Datenbankentwurfs **289**

11.1 Indexüberwachung mit Datenauflistungen **289**
11.1.1 Ein Auflistsatz für fehlende und überflüssige Indizes **290**
11.1.2 Ein Auflistelement für fehlende Indizes **291**
11.1.3 Ein Auflistelement für die Indexverwendung **293**
11.1.4 Daten sammeln und auswerten **294**

11.2 Partitionierung mit Indizes **299**
11.2.1 Horizontale Partitionierung **300**
11.2.2 Vertikale Partitionierung **303**

11.3 Arbeiten mit dem Datenbankoptimierungsratgeber **303**
11.3.1 Tipps zur Verwendung des Datenbankoptimierungsratgebers **309**

11.4 Zusammenfassung **310**

12 Kontrollieren von Ressourcen **311**

12.1 Funktionsweise der Ressourcenkontrolle **312**

12.2 Einrichten der Ressourcenkontrolle **313**
12.2.1 Erstellen von Ressourcenpools **313**
12.2.2 Einrichten von Arbeitsauslastungsgruppen **314**
12.2.3 Entwerfen einer Klassifizierungsfunktion **315**
12.2.4 Aktivieren der Ressourcenkontrolle **316**

12.3 Zusammenfassung **317**

13 Testen und Optimieren des E/A-Systems 319
13.1 Physikalisches Datenbanklayout 320
13.2 Testen des E/A-Systems 322
13.2.1 Testen auf Korrektheit von E/A-Operationen mit SQLIOSIM 322
13.2.2 Messen des E/A-Durchsatzes mit SQLIO 324
13.3 Zusammenfassung 331

Teil 4 Anhänge 333

A Häufige Fehler und Irrtümer 335
A.1 Vertrauen auf RAID 5 335
A.2 Planung des E/A-Systems nach Kapazität 335
A.3 Gruppierter Index für den Primärschlüssel 335
A.4 Verwenden von GUIDs als Primärschlüssel 336
A.5 Verwenden von Autogrow 336
A.6 Verwenden von SHRINK DATABASE 337
A.7 Aktualisieren der Statistiken nach dem Re-Index 337
A.8 Optimierung = leistungsfähigere Hardware anschaffen 337
A.9 Scans sind generell schlecht 338
A.10 Dynamisches SQL ist »ungesund« 338
A.11 Verwenden automatisch erstellter UNIQUE-Indizes 338
A.12 Cursors sind in jedem Fall zu vermeiden 339
A.13 Mehr Einschränkungen in der WHERE-Klausel senken die Abfragekosten 341
A.14 Unzureichende Einschränkungen 343

B Literatur 345

Stichwortverzeichnis 347

Für Karin und Claus

Macht weiter so!

Vorwort

Microsofts SQL Server ist mittlerweile sehr »erwachsen« geworden. SQL Server-Installationen begegnen Ihnen nahezu überall: sei es eine Desktop-Datenbank mit einer Größe von 10 MByte oder eine Installation in einem Cluster mit Datenbanken im TByte-Bereich.

Die meisten Installationen haben hierbei eine Gemeinsamkeit: Die zu verarbeitenden Datenmengen steigen beständig an. Gleichzeitig wachsen die Komplexität der Anwendungen und die Anforderungen an diese Anwendungen ebenso. Durch diese Konstellation wird auch das Thema Performance-Optimierung mit der Zeit immer wichtiger. Wachsende Datenmengen und dennoch kürzere Antwortzeiten sind heutzutage fast schon übliche Anforderungen an ein Datenbank-Management-System – und somit gewinnt das Thema Performance zunehmend an Bedeutung.

Ich habe dies in meinen Projekten sozusagen am eigenen Leibe erfahren dürfen. Ganz gleich, ob der Schwerpunkt meiner Tätigkeit Datenbankentwicklung oder Datenbankadministration ist: Die Optimierung von Datenbankschemata, Abfragen oder Indizes ist in jedem Projekt ein wichtiges Thema. Es gibt mittlerweile sogar Projekte, bei denen die Optimierung selbst den Schwerpunkt der Aufgabe bildet.

Irgendwann habe ich deshalb begonnen, mir Notizen anzufertigen, auf die ich immer wieder zurückgreife, wenn Probleme hinsichtlich der Performance auftreten bzw. eine Performance-Optimierung angegangen werden soll. Mit der Zeit wuchsen diese Notizen immer mehr, sodass ich gezwungen war, sie zu ordnen. Eines Tages ist dann die Idee zu dem vorliegenden Buch entstanden. Warum nicht die geordnete Materialsammlung in eine professionelle Form bringen? Das Resultat halten Sie gerade in der Hand. Es ist tatsächlich so, dass ich dieses Buch auch für mich selber geschrieben habe, um es als Nachschlagewerk zu verwenden.

Auf der anderen Seite möchte ich meine Erfahrungen sehr gerne für andere zur Verfügung stellen, die ebenfalls an der sehr komplexen Thematik Performance-Optimierung Interesse haben. Falls Sie Datenbankentwickler oder Datenbankadministrator sind, dann ist dieses Buch also für Sie geeignet. Selbst wenn Sie momentan vielleicht noch keinen Bedarf sehen: Optimierung wird gewiss ein Thema für Sie werden, sobald Sie Datenbankanwendungen entwickeln oder SQL Server administrieren.

Ich hoffe, dass dieses Buch für Sie von Nutzen ist, wenn Sie eigene Optimierungsstrategien entwerfen oder einfach die Performance-Optimierung anpacken. Ich wünsche Ihnen viel Spaß beim Lesen – wenigstens genau so viel, wie ich beim Schreiben hatte.

Noch ein Wort zum Abschluss.

Alle an diesem Buch Beteiligten haben sehr sorgfältig gearbeitet, um ein möglichst fehlerfreies Werk abzuliefern. Die Beispiele wurden allesamt sorgfältig entwickelt und überprüft. Dennoch kann es leider vorkommen, dass der eine oder andere Fehler unentdeckt geblieben ist. Zögern Sie daher bitte nicht, mich zu kontaktieren, wenn Sie irgendwelche Ungereimtheiten entdecken. Ich freue mich über Ihre Resonanz und bin gerne bereit zu helfen, falls Sie Fragen oder Anregungen haben.

Holger Schmeling

http://www.hschmeling.de

1 Einleitung

SQL Server ist ein komplexes System, in dem viele Komponenten optimal aufeinander abgestimmt sein müssen, damit es seine bestmögliche Leistung erzielt. Etliche Faktoren spielen hierbei eine Rolle, sodass eine Performance-Optimierung häufig sehr vielschichtig und schwierig ist.

Hinzu kommt, dass eine Optimierung auch immer eine Überwachung des Systems einschließt. Sie werden bestimmte Parameter beobachten müssen um herauszufinden, ob eine Optimierung erforderlich ist und sich überhaupt lohnt, und wo die vielversprechendsten Ansatzpunkte für eine Optimierung oder die Ursachen für bestehende Probleme liegen. Dies alles führt letztlich dazu, dass eine Optimierung recht mühevoll und knifflig sein kann.

SQL Server 2008 bietet einige neue und sehr leistungsfähige Features, die sowohl den Datenbankentwickler als auch den Datenbankadministrator bei der Überwachung und Optimierung unterstützen. Allen voran kann hier sicherlich das Verwaltungs-Data Warehouse genannt werden, mit dem nun endlich Informationen für die Leistungsüberwachung an zentraler Stelle gespeichert und ausgewertet werden können. Die neuen Möglichkeiten erleichtern das Aufspüren von Problemen und deren Beseitigung selbstverständlich nur dann, wenn man sie auch verstanden hat und anwenden kann. Letztlich wird die Thematik »Performance-Optimierung mit SQL Server 2008« also zunächst einmal noch ein wenig komplexer, weil mehr Möglichkeiten und Werkzeuge zur Verfügung stehen, die erst einmal beherrscht werden müssen.

Natürlich ist es kompliziert, werden Sie sagen, denn ansonsten würden ja keine Bücher über diese Thematik geschrieben. Stimmt genau. Und da Sie dieses Buch in der Hand halten, haben Sie erkannt, dass das Thema Optimierung durchaus eine tiefergehende Betrachtung verdient. Egal, ob Sie bereits bestehende Performance-Probleme lösen möchten bzw. müssen, ob Sie Vorsorgemaßnahmen treffen möchten, damit die Wahrscheinlichkeit solcher Probleme minimiert wird, ob Sie Überwachungen Ihres Systems einrichten möchten, damit Sie möglichst frühzeitig über Probleme informiert werden, oder ob Sie die Konfiguration der Hardware für eine SQL Server-Installation planen: Dieses Buch wird Ihnen dabei helfen.

1.1 Optimierungsmodell

Wenn Sie eine Optimierung in Angriff nehmen, ist es natürlich sinnvoll, hierfür eine Vorgehensweise festzulegen. In der Praxis hat sich hierbei das in Abbildung 1.1 gezeigte Modell zur Performance-Optimierung bewährt.

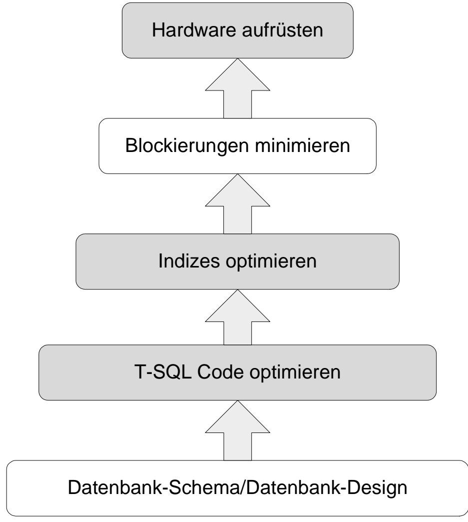


Abbildung 1.1: *Optimierungsmodell*

Die hier dargestellten Bereiche bauen jeweils aufeinander auf. Eine Optimierungsstrategie sollte also zunächst auf einem Datenbankschema aufsetzen, das für die Anwendung angepasst wurde. Hierbei ein ausgewogenes Verhältnis zwischen Normalisierung und De-Normalisierung zu finden, ist eine Aufgabe, die eine gewisse Erfahrung und natürlich ein Verständnis der zu erwartenden Abfragen erfordert.

Steht das Schema fest, so kann der SQL-Code optimiert werden. Nach meiner Erfahrung ist dieser Bereich in fast allen Fällen derjenige mit dem größten Potenzial. Oftmals wurde der bestehende SQL-Code von Entwicklern verfasst, die ursprünglich aus der »prozeduralen Welt« stammen und gewissermaßen einen Ausflug zu SQL unternommen haben. Für die Erstellung effektiver SQL-Anweisungen ist ein Verinnerlichen des SQL zugrunde liegenden mengenorientierten Ansatzes jedoch wesentlich. Außerdem ist es erforderlich, dass Sie ein tiefes Verständnis dafür entwickeln, wie SQL Server Abfragen verarbeitet.

Aufbauend auf dem optimierten SQL-Code sollte dann eine Indexoptimierung folgen, mit dem Ziel, die physikalische Datenbankstruktur durch geeignete Indizes zu ergänzen. Auch für diese Aufgabe sind ein beträchtliches Maß entsprechender Erfahrung sowie Kenntnisse des zugrunde liegenden Datenbank-Management-Systems und der Anwendungen unabdingbar.

An die erfolgreiche Indexoptimierung sollte sich eine Untersuchung der blockierenden Prozesse anschließen. In dieser Phase werden Transaktionen untersucht, die sich um Ressourcen »streiten«. Meist werden diese Ressourcen Tabellenzeilen sein; denkbar sind aber auch Blockierungen aus anderen Gründen, zum Beispiel CPUs oder E/A-Geräte.

Ist die Optimierung der ersten vier Stufen abgeschlossen, kann schließlich die Konfiguration der Hardware untersucht werden. Obwohl die meisten Optimierungsstrategien genau dort ansetzen, ist dieser Schritt kurioserweise der letzte in der Reihe der Optimierungsstufen. Die Erklärung hierfür ist sicherlich, dass für eine Hardware-Optimierung vergleichsweise wenige Kenntnisse notwendig sind. Möglicherweise wird mit der Einführung neuer Massenspeichergeräte, die ohne Mechanik auskommen (Solid State-Laufwerke), die Hardwareoptimierung tatsächlich einmal die insgesamt kostengünstigste und einfachste Verfahrensweise für das Tuning eines Systems.

1.2 Was vermittelt dieses Buch?

Wie bereits erwähnt, ist das Thema Optimierung insgesamt sehr umfangreich. Es ist daher schlichtweg nicht möglich, es vollständig in einem einzigen Buch abzuhandeln. Ein solches Buch müsste sicherlich einige Tausend Seiten Umfang haben. Sehr wahrscheinlich wird ein solches Projekt auch in der Zukunft nicht in Angriff genommen werden.

Es ist also erforderlich, sich auf bestimmte Bereiche bzw. Schwerpunkte zu konzentrieren. Schauen Sie sich bitte noch einmal Abbildung 1.1 an. Für die in dieser Abbildung dunkelgrau dargestellten Thematiken werden Sie in diesem Buch weiterführende Informationen finden. Die hell dargestellten Bereiche werden hingegen nur am Rande behandelt. Ich habe die Thematiken für dieses Buch sorgfältig ausgewählt, weil sie nach meiner Erfahrung die vielversprechendsten Ansatzpunkte für eine wirkungsvolle Optimierung bilden. Oftmals ist es auch der Fall, dass Sie gewisse Stufen der Optimierungspyramide überhaupt nicht in Angriff nehmen können. Dies gilt zum Beispiel für die Optimierung des Datenbankdesigns bzw. den logischen Datenbankentwurf. Die zu optimierenden Datenbanken sind vielfach bereits vorhanden, sodass Sie keine Möglichkeit haben, Änderungen am Datenbankdesign vorzunehmen. Dies gilt natürlich nicht, wenn Sie in der glücklichen Lage sind, Ihre Datenbank selbst zu entwerfen. In diesem Fall sollten Sie unbedingt darauf achten, dass Ihr Datenbankdesign auf ihre Erfordernisse hin abgestimmt ist. Zum Thema Datenbankdesign finden Sie allerdings in diesem Buch nur wenige Hinweise. Ich empfehle Ihnen insbesondere die Lektüre von [3]. Dies ist ein hervorragendes Buch zum Datenbankdesign, speziell abgestimmt auf SQL Server.

Die Optimierung von Abfragen erhält in diesem Buch den höchsten Stellenwert. Den Schwerpunkt bildet dabei gar nicht so sehr eine Optimierung des T-SQL-Codes selber. Vielmehr geht es darum, in welcher Weise SQL Server Abfragen verarbeitet, und wie Sie in diesem Zusammenhang Probleme aufspüren und beheben können. Ziel ist, dass Sie ein tiefergehendes Verständnis für die Verarbeitung von Abfragen entwickeln.

1.2.1 Tipps und Hinweise

An vielen Stellen werden Ihnen speziell gekennzeichnete Textpassagen begegnen. Die unterschiedlichen Hervorhebungen haben die folgenden Bedeutungen:

Hier stehen weiterführende Informationen und Hinweise. Prägen Sie sich diese Textpassagen ein.

An dieser Stelle finden Sie zusätzliche Tipps zur Arbeitserleichterung.

Hier heißt es »Aufgepasst!«. An dieser Stelle finden Sie Punkte, die Sie besser unterlassen sollten oder die Probleme verursachen.

1.3 Wo finde ich was?

Das vorliegende Buch ist in drei Teile gegliedert.

Teil 1: Grundlagen

In Teil 1 geht es zunächst um grundlegende Punkte. Hier erhalten Sie eine Einführung in die generelle Arbeitsweise von SQL Server sowie die für eine Überwachung und Optimierung zur Verfügung stehenden Werkzeuge.

- **Kapitel 2: Datenverwaltung durch SQL Server**. Hier erfahren Sie, in welcher Weise SQL Server die in Datenbanken vorhandenen Informationen speichert und liest.
- **Kapitel 3: Ausführung von Abfragen**. In diesem Kapitel erhalten Sie einen ersten Überblick über die Ausführung von Abfragen durch SQL Server. Diese Übersicht bildet die Grundlage für die nachfolgenden Kapitel.
- **Kapitel 4: Werkzeuge und Indikatoren zum Messen der Leistung**. Wenn Sie eine Optimierung angehen, müssen Sie oftmals herausfinden, wo die Ursachen für ein Problem liegen. SQL Server stellt hierzu einige Werkzeugen bereit. In Kapitel 4 erhalten Sie eine Einführung in die zur Verfügung stehenden Instrumente.

Teil 2: Physische Aspekte des Datenbankentwurfs

Teil 2 befasst sich damit, wie Sie bestehende Datenbanken in ihrer physischen Struktur verändern können, um Abfragen zu beschleunigen. Der Schwerpunkt liegt dabei auf Indizes.

- **Kapitel 5: Verwenden von Indizes**. Dieses Kapitel erklärt, wie Indizes funktionieren und wie Sie mit ihnen arbeiten.
- **Kapitel 6: Verwalten von Indizes**. Indizes bringen nicht nur Vorteile, sondern erfordern auch einen entsprechenden Verwaltungsaufwand. In Kapitel 6 erfahren Sie, wie Sie Indizes verwalten und sie kontrollieren.
- **Kapitel 7: Partitionierung**. Größere Tabellen oder Indizes können davon profitieren, wenn Ihre Daten auf unterschiedliche physische Speicherorte verteilt, also partitioniert, werden. Kapitel 7 beleuchtet die Vor- und Nachteile einer solchen Aufteilung.
- **Kapitel 8: Komprimierung von Daten**. SQL Server 2008 bietet erstmals die Möglichkeit, Daten transparent zu komprimieren, um E/A-Vorgänge zu minimieren. In diesem Kapitel erhalten Sie einen Überblick über die Auswirkungen der Komprimierung.

Teil 3: Optimierung

Es ist sicherlich nicht erstaunlich, dass der umfangreichste Teil des Buches sich mit der eigentlichen Optimierung auseinandersetzt. Aufbauend auf den vorherigen beiden Teilen werden Sie in diesem Teil erfahren, welche Möglichkeiten der Optimierung SQL Server bietet. Im Mittelpunkt steht dabei die Optimierung von Abfragen. Andere Aspekte wie beispielsweise eine Optimierung der Hardware werden aber ebenfalls behandelt.

- **Kapitel 9: Analysieren und Optimieren von Abfragen**. Dieses Kapitel vermittelt detaillierte Einblicke in die Arbeitsweise des Optimierers. Außerdem erfahren Sie hier, wie Sie Abfragepläne für die Optimierung verwenden, und welche Rolle Statistiken bei der Erstellung von Ausführungsplänen spielen.
- **Kapitel 10: Auffinden problematischer Abfragen**. An dieser Stelle sehen Sie die Werkzeuge aus Kapitel 4 in Aktion. Sie erfahren, wie Sie vorgehen können, um Abfragen aufzuspüren, die optimiert werden sollten.
- **Kapitel 11: Optimieren des physischen Datenbankentwurfes**. Schwerpunkt dieses Kapitels sind noch einmal Indizes. Sie lernen weiterführende Konzepte zur Überwachung der Indexverwendung kennen und erfahren, wie Sie eine Indexoptimierung angehen.
- **Kapitel 12: Kontrollieren von Ressourcen**. SQL Server 2008 gestattet erstmals die Zuteilung begrenzter Ressourcen an bestimmte Verbindungen. Wie Sie dieses Instrument für eine Optimierung einsetzen können, ist Gegenstand von Kapitel 12.
- **Kapitel 13: Testen und Optimieren des E/A-Systems**. Ein bestmöglich abgestimmtes E/A-System ist eine wesentliche Komponente für ein optimal funktionierendes System. Kapitel 13 erklärt, wie Sie Ihr E/A-System strukturieren sollten, und welche Möglichkeiten Sie für Messungen des E/A-Durchsatzes sowie der Funktionalität haben.

Ganz am Ende des Buches gibt es schließlich noch einen Anhang, in dem ich in loser Auflistung Fehler und Irrtümer zusammengefasst habe, die mir immer wieder begegnen.

Sie werden in diesem Buch übrigens keine Angaben zu URLs finden, denn nichts ist so flüchtig wie Informationen im Internet. Wann immer ein Verweis auf das Internet erforderlich ist, erhalten Sie statt einer URL die entsprechenden Suchbegriffe, sodass Sie in die Lage versetzt werden, die Informationen mit der Suchmaschine Ihrer Wahl zu aufzuspüren.

1.4 Welche Voraussetzungen werden benötigt?

Persönliche Kenntnisse

Für das Verständnis der in diesem Buch behandelten Thematiken sollten Sie zumindest über Grundkenntnisse von T-SQL verfügen. Darüber hinaus wird vorausgesetzt, dass Sie mit dem SQL Server Management Studio umgehen können.

Es ist weiter von Vorteil, wenn Sie auch die anderen Programme aus der SQL Server-Programmgruppe bedienen können und deren Zweck kennen. Dies ist allerdings nicht unbedingt erforderlich, da die benötigten Werkzeuge in diesem Buch auch noch einmal erklärt werden. Allerdings sind diese Erklärungen eher knapp gehalten.

Systemvoraussetzungen

Wenn Sie die präsentierten Beispiele und Konzepte nachvollziehen möchten, dann benötigen Sie die SQL Server Developer Edition oder Enterprise Edition. Nur in diesen Editionen steht der volle Funktionsumfang zur Verfügung, den Sie für entsprechende Experimente benötigen. Mit anderen Editionen können Sie zum Beispiel keine Komprimierung oder Ressourcenkontrolle ausprobieren.

Eine 180 Tage lang verwendbare Testversion der SQL Server Enterprise Edition können Sie kostenlos aus dem Internet herunterladen.

Beispieldaten

Viele der in diesem Buch verwendeten Beispiele verwenden Daten, die nach dem Zufallsprinzip erzeugt werden. Bitte erwarten Sie daher keine hundertprozentige Übereinstimmung der Ergebnisse, wenn Sie solche Beispiele nachvollziehen.

Sie benötigen für einige Versuche die von Microsoft zur Verfügung gestellte Beispieldatenbank `AdventureWorks2008`. Diese Datenbank können Sie aus dem Internet herunterladen. Sie finden die Datenbankdateien aber auch auf der Begleit-CD.

1.5 Was ist auf der Begleit-CD?

Auf der Begleit-CD finden Sie den Quellcode für alle in diesem Buch vorhandenen Beispiele, nach Kapitel geordnet. Außerdem sind auf der Begleich-CD die Datenbank- und Protokolldateien für die Beispieldatenbank `AdventureWorks2008` enthalten.

1.6 Danksagung

Wenn Sie den Einband dieses Buches betrachten, dann finden Sie dort den Namen des Autors. Zweifelsohne hat der Autor den größten Teil der Arbeit geleistet und daher steht sein Name auch völlig zu Recht auf dem Buchdeckel. Das Erscheinen eines Buches ist jedoch nicht denkbar ohne die Mithilfe weiterer Menschen und damit letztendlich das Resultat einer Teamarbeit. Ohne diese Unterstützung vieler Menschen des Verlags und meines persönlichen Umfelds wäre dieses Buch niemals erschienen.

Aus diesem Grund möchte ich mich bei allen, die am Erscheinen dieses Buches beteiligt waren, ganz ehrlich bedanken. Allen voran ist da natürlich meine Lektorin Sylvia Hasselbach von Addison Wesley zu nennen, die es mir überhaupt erst ermöglicht hat, dieses Buch zu schreiben.

Ein weiteres Dankeschön geht an Wim Nienkerke, der dieses Buch auf fachliche Korrektheit hin überprüft und geholfen hat, den einen oder anderen Fehler auszumerzen. Er hat es außerdem verstanden, mich mit dem nötigen Nachdruck anzutreiben, auch wenn dies nicht immer so erfolgreich war.

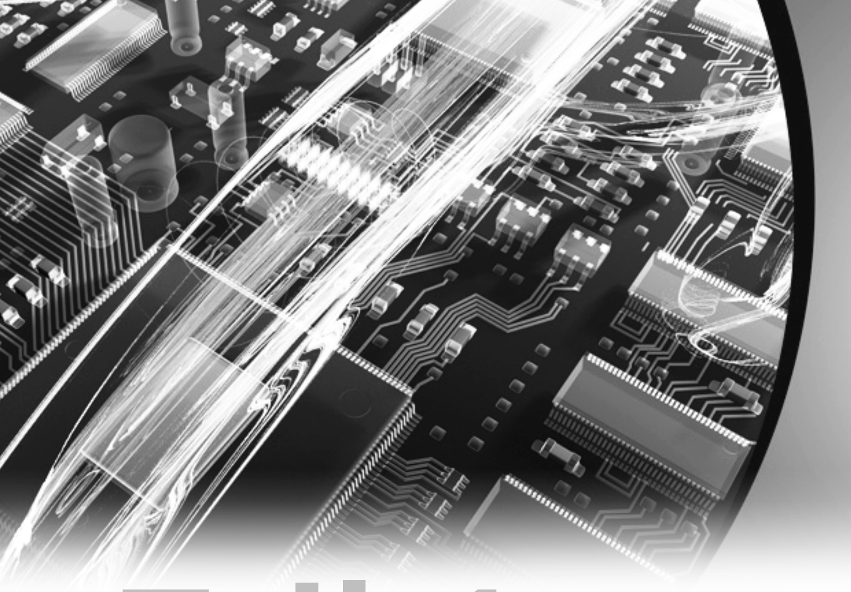
Meinen Kollegen Maxim Akimov und Marian Grzesik gebührt ebenso Dank dafür, dass sie mich beständig angespornt und ermuntert haben.

Das größte Dankeschön verdient natürlich Angelika, die auf vieles verzichtet hat, während das vorliegende Buch in Arbeit war. Danke, dass Du mich in dieser Zeit ertragen hast!

Bestimmt habe ich den einen oder anderen in der Auflistung vergessen. Falls also irgendwer seinen Namen hier vermisst, dann bitte nicht traurig sein. Es ist keine Absicht.

1.7 Performanz oder Performance?

Zum Abschluss der Einleitung möchte ich noch eine weitere Entschuldigung loswerden. SQL Server ist ein amerikanisches Produkt. Dieses Produkt existiert zwar auch in einer deutschen Version; die Originalsprache für die Begriffswelt ist jedoch Englisch. Dies ist auch der deutschen Übersetzung deutlich anzumerken. Teilweise wurden Begriffe gar nicht oder sehr unglücklich in die deutsche Sprache übersetzt. Was herauskommt, ist an einigen Stellen eine Vermischung von deutschen und englischen Begriffen, *Denglish* eben. Mittlerweile ist es auch üblich, die englischen Begriffe einfach in der deutschen Sprache zu verwenden. Wo immer es möglich war, habe ich in diesem Buch deutsche Begriffe verwendet. An einigen Stellen war dies jedoch schlichtweg nicht machbar. Um eine konsistente Begriffswelt zu wahren, habe ich stets die Begriffe aus der deutschen Originaldokumentation verwendet, auch wenn die dort präsentierte Übersetzung meiner Ansicht nach nicht besonders gelungen ist. Ich wollte jedoch nicht durch eine eigene Übersetzung zusätzlich Verwirrung stiften. Für das dennoch an einigen Stellen vorhandene Sprachwirrwarr bitte ich Sie deshalb an dieser Stelle um Entschuldigung.



Teil 1

Grundlagen

2 Datenverwaltung durch SQL Server 23

3 Ausführung von Abfragen 31

4 Werkzeuge und Indikatoren zum Messen der Leistung 45

2 Datenverwaltung durch SQL Server

Aus Performance-Sicht ist das E/A-System eines Computers sicherlich der größte Schwachpunkt. Bei physikalischen Ein-/Ausgabevorgängen sind mechanische Elemente beteiligt, die aufgrund der geltenden physikalischen Gesetze entsprechenden Einschränkungen hinsichtlich ihrer Geschwindigkeit unterworfen sind. Ein wesentlicher Ansatz für eine Optimierung ist daher immer die Minimierung der physikalischen Lese- und Schreiboperationen. Dafür ist es natürlich unerlässlich, dass Sie verstehen, in welcher Weise SQL Server das E/A-System verwendet.

In diesem Kapitel wird erläutert, in welcher Weise SQL Server die Daten für Tabellen und Indizes organisiert. Hierbei wird allerdings nicht im Detail darauf eingegangen, wie Sie als Administrator Ihre Datenbanken physikalisch auf die Speichermedien verteilen. Da dies kein Buch zur Adminstration von SQL Server ist, wird in diesem Kapitel lediglich zusammengefasst dargestellt, wie SQL Server-Datendateien, -Indexdateien und -Protokolldateien verwalten, wobei vor allem der Performance-Aspekt im Vordergrund steht.

2.1 Datenbanken

Eine SQL Server-Datenbank wird in Dateien des Betriebssystems gespeichert, die von SQL Server verwaltet werden. Jede Datenbank besteht dabei grundsätzlich aus zwei Dateitypen:

- **Datendateien**. In diesen Dateien, welche die Standardendung MDF oder NDF besitzen, werden die eigentlichen Tabellendaten gespeichert. Auch die Daten für Indizes sind in diesen Dateien abgelegt. Je nach Anforderung können mehrere dieser Datendateien existieren, die in SQL Server-Dateigruppen organisiert sind. Hierbei muss es immer eine primäre Datendatei in der primären Dateigruppe geben, in der zumindest die Systemkataloge gespeichert sind.
- **Protokolldateien**. Für die Sicherstellung der transaktionalen Integrität werden alle verarbeiteten Transaktionen zusätzlich in separaten Dateien protokolliert. Diese Protokolldateien haben die Standardendung LDF. Auch hier ist es möglich, mehrere Dateien anzulegen, die allerdings nicht in Dateigruppen eingegliedert sind.

Die kleinste Verwaltungseinheit für Tabellen-, Index- und Protokolldaten ist hierbei nicht etwa das Byte, sondern eine Seite mit einer Größe von 8 kByte. Es ist sehr wichtig, dass Sie sich diese Tatsache einprägen, da Ihnen dieses Konzept immer wieder begegnen wird. In jeder Datenseite sind hierbei einige Bytes für den Seitenkopf reserviert, in welchem zum Beispiel eine Prüfsumme existiert, mit deren Hilfe die Konsistenz einer Seite überprüft werden kann. Für reine Nutzdaten bleiben 8.060 Byte übrig. Auf dem Speichermedium werden stets acht Seiten zusammenhängend verwaltet, die als ein Block bezeichnet wer-

den. So ein Block ist also 64 kByte groß. Wann immer zusätzlicher Speicher für Tabellen oder Indizes benötigt wird, erfolgt die Anforderung dieses Speichersblocks – und nicht etwa seitenweise. Lediglich kleine Tabellen, deren Größe unterhalb von 64 kByte liegt, können sich einen Block teilen.

2.2 Daten lesen

Zur Beschleunigung von Lese- und Schreiboperationen verwaltet SQL Server einen Datencache im Hauptspeicher. In diesem Cache werden Daten gespeichert, die von der Festplatte gelesen wurden. Ziel ist hier, langsame physikalische Leseoperationen zu eliminieren.

Benötigt eine Abfrage Daten, so wird zunächst geprüft, ob die entsprechenden Datenseiten bereits im Cache existieren. Ist dies der Fall, können die Daten durch sogenannte logische Lesevorgänge aus dem Cache geholt werden. Es sind dann keine physikalischen Leseoperationen erforderlich. Jeder logische Lesevorgang liest hierbei genau eine Datenseite aus dem Cache. Existieren die erforderlichen Daten dagegen nicht im Cache, werden sie vom Datenträger in den Cache übertragen. In diesem Fall sind also physikalische Leseoperationen erforderlich. Eine physikalische Leseoperation liest Daten nicht unbedingt seitenweise, sondern normalerweise in größeren Blöcken. Falls größere Datenmengen von der Festplatte gelesen werden müssen, arbeitet SQL Server mit einem Trick: Noch während der Abfrageprozessor bereits gelesene Daten verarbeitet, werden durch sogenannte Read Aheads weitere Leseoperationen ausgeführt. Dadurch steigt die Wahrscheinlichkeit, dass Daten bereits im Cache vorhanden sind, wenn sie benötigt werden.

Der Read Ahead-Ansatz ist übrigens recht rigoros: Es werden so viele Daten wie möglich so schnell wie möglich gelesen. In der SQL Server Enterprise Edition kann eine Read Ahead-Operation hierbei bis zu 1.024 kByte in einem Lesevorgang von der Festplatte holen.

Benötigt eine Abfrage Daten, die von der Festplatte gelesen werden müssen, dann sind die erforderlichen Lesevorgänge typischerweise nicht seriell, sondern über Sektoren verteilt. Dadurch werden die erforderlichen physikalischen Leseoperationen zusätzlich verlangsamt. Dieser Effekt wird noch verstärkt, wenn die Tabellen- oder Indexdaten bereits stark fragmentiert sind, was mit der Zeit eintreten kann, wann immer Daten verändert werden.

Eine hohe Cache-Trefferquote ist ein wesentliches Ziel der Optimierung. Hierzu ist es einerseits erforderlich, dass die SQL Server-Maschine mit ausreichend Hauptspeicher bestückt wird. Auf der anderen Seite sollte dafür gesorgt werden, dass unnötige physikalische Leseoperationen vermieden werden. Zunächst einmal sind diese Operationen langsam, was sich natürlich negativ auf die Leistung auswirkt. Hinzu kommt hier auch noch, dass durch vermeidbare physikalische Leseoperationen Daten in den Datencache übertragen werden, die dort nicht sehr nützlich sind, da sie möglicherweise nur selten benötigt werden.

Ich kann an dieser Stelle nur noch einmal wiederholen: Der wichtigste Ansatz bei der Optimierung von Abfragen besteht darin, mit so wenig Leseoperationen wie nur möglich auszukommen. Der Hauptteil dieses Buches befasst sich daher mit den Ihnen hierfür zur Verfügung stehenden Möglichkeiten.

2.3 Daten schreiben

Datenänderungen sind grundsätzlich sogenannte logische Schreibvorgänge. Dies bedeutet, dass Änderungen zunächst nur im Pufferspeicher vorgenommen werden, was sehr schnell geht. Alle im Pufferspeicher modifizierten Seiten (die sogenannten »dirty pages«) werden entweder beim Erreichen eines Prüfpunktes oder durch den Lazy Writer-Prozess, der den Pufferspeicher in zyklischen Abständen nach geänderten Seiten durchsucht, permanent (also auf der Festplatte) gespeichert. Da diese physikalischen Schreibvorgänge asynchron verlaufen, fallen sie für OLTP-Anwendungen, die typischerweise Datenänderungen in kleinen Blöcken durchführen, nicht besonders ins Gewicht.

Ganz anders verhält es sich mit dem Transaktionsprotokoll. Änderungen in der Protokolldatei werden immer synchron geschrieben. Eine Transaktion ist erst dann beendet, wenn die Einträge in der Protokolldatei permanent gespeichert sind. Dies bedeutet, dass jedes `COMMIT` erst dann erfolgreich ist, wenn die zugehörigen Protokolleinträge auf der Festplatte »verewigt« wurden. Diese Methode, die als *Write Ahead Log* bezeichnet wird, stellt letztlich sicher, dass eine Transaktion immer ganz oder gar nicht ausgeführt wird. Dies ist das wesentliche Merkmal einer Transaktion. Somit ist auch bei einem Systemausfall während des Speicherns einer Transaktion durch diese Verfahrensweise die transaktionale Integrität gewährleistet.

Da Transaktionen nacheinander ausgeführt werden, erfolgt das Schreiben des Transaktionsprotokolls immer seriell. Jeder neue Protokolleintrag beginnt dabei auf einer Sektorgrenze der Festplatte. Wenn Ihre Festplatte also eine Sektorgröße von 4.096 Byte besitzt und Sie viele kleine Transaktionen ausführen, deren Protokolleintrag weniger als 4.096 Byte Speicherplatz benötigt (was typisch für ein OLTP-System ist), so wird einiges an Speicherplatz verschwendet.

Generell ist es daher so, dass die Dauer von Schreibvorgängen im Wesentlichen dadurch bestimmt wird, wie schnell die Protokolleinträge geschrieben werden können.

Aus Performance-Sicht ist die Konsequenz hieraus ziemlich klar: Für das Transaktionsprotokoll sollte ein Speichermedium verwendet werden, das schnelle Schreibvorgänge ermöglicht. Insbesondere ist es wichtig, dass für den entsprechenden Datenträger der Schreibcache eingeschaltet wird. Hierbei müssen Sie jedoch darauf achten, dass der Schreibcache batteriegepuffert ist, da ansonsten bei einem Systemausfall keine automatische Wiederherstellung der Datenbanken bei einem Neustart gewährleistet ist.

Die Verwaltung von Tabellen- und Protokolldaten unterscheidet sich also grundsätzlich. Auf das Transaktionsprotokoll wird in fast allen Fällen nur seriell schreibend zugegriffen, während der Zugriff auf Daten eher zufällig ist. Sie werden daher überall einen Hinweis finden, der auch an dieser Stelle ebenfalls nicht fehlen darf: Trennen Sie Daten- und Protokolldateien so auf, dass sie auf unterschiedlichen Datenträgern liegen. Allzu oft begegne ich Systemen, die ein RAID 5-System als Speichermedium verwenden, wobei sowohl die

Daten- als auch die Protokolldateien auf diesem System abgelegt werden. Für Leseoperationen ist diese Konfiguration natürlich in Ordnung. Für Schreiboperationen allerdings ist dies eindeutig nicht die optimale Lösung. Das Transaktionsprotokoll ist besser auf einem RAID 0- oder RAID 10-System aufgehoben.

2.3.1 Experiment: Transaktionsgröße

Um zu untersuchen, welchen Einfluss die Geschwindigkeit, mit der das Transaktionsprotokoll geschrieben wird, auf Datenänderungen hat, führen wir ein kleines Experiment durch. In diesem Experiment werden 10.000 Zeilen in eine Tabelle eingefügt, wobei jeder `INSERT`-Befehl in einer eigenen Transaktion ausgeführt wird. Das folgende Skript erledigt diese Aufgabe:

```
if (object_id('T1') is not null)
  drop table T1
go
-- Erzeuge eine Test-Tabelle
create table T1
 (
   id int not null default 0
  ,daten nchar(20) null
 )
go
set nocount on
-- Setze alle Leistungsindikatoren zurück.
dbcc sqlperf('sys.dm_os_wait_stats',clear) with no_infomsgs
-- Füge 10.000 Zeilen in einer Transaktion hinzu
declare @i int
set @i = 1
while (@i < 10000)
 begin
   begin tran
    insert T1(id) values(@i)
   commit
   if (@i % 1000) = 0
     raiserror('1000 Zeilen hinzugefügt', 0, 1) with nowait
   set @i = @i + 1
 end
-- Zeige an, wie oft und wie lange auf des Schreiben
-- des Transaktionsprotokolls gewartet wurde.
select * from sys.dm_os_wait_stats
 where wait_type = 'WRITELOG'
```

Es wird zunächst eine Tabelle angelegt, in die anschließend in einer Schleife 10.000 Zeilen eingefügt werden. Am Anfang des Skripts werden alle Leistungsindikatoren zurückgesetzt, damit die bisher ermittelten Werte keinen Einfluss auf unser Messergebnis haben. Dies geschieht durch die folgende Zeile:

```
dbcc sqlperf('sys.dm_os_wait_stats',clear) with no_infomsgs
```

Die letzte SELECT-Anweisung ermittelt dann die Werte für den Indikator WRITELOG, der angibt, wie oft und wie lange auf das Schreiben des Transaktionsprotokolls gewartet werden musste.

```
select * from sys.dm_os_wait_stats
 where wait_type = 'WRITELOG'
```

Der im Listing fett gedruckte Teil ist hier entscheidend: Jedes INSERT wird in einer eigenen Transaktion ausgeführt. Dies bedeutet, dass die Schleife insgesamt 10.000 Protokolleinträge und somit auch 10.000 Schreibvorgänge erzeugt.

Wir wollen das obige Skript zweimal laufen lassen: einmal mit und einmal ohne eingeschalteten Schreibcache der Festplatte. Sie können den Schreibcache über den Gerätemanager konfigurieren. Wählen Sie dafür bitte die *Eigenschaften* für das entsprechende Laufwerk und dort das Register *Richtlinien* aus (Abbildung 2.1).

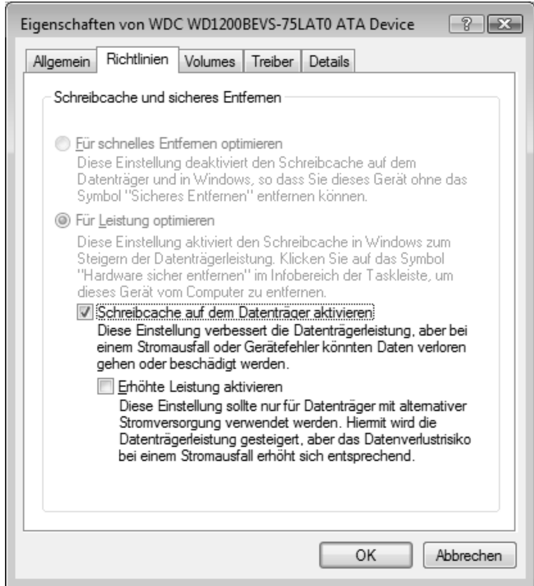


Abbildung 2.1: *Konfiguration des Festplatten-Schreibcache*

In Abbildung 2.1 sehen Sie das Ergebnis beider Ausführungen auf meiner Maschine.

	Zeit	WRITELOG	WRITELOG
mit Schreibcache	6 s	6 s	100 %
ohne Schreibcache	139 s	138 s	99 %

Tabelle 2.1: *Schreiben vieler kleiner Transaktionen mit und ohne Schreibcache*

Wie nicht anders zu erwarten, hat der Schreibcache einen ganz erheblichen Einfluss auf die Ausführungszeit. Es ist außerdem deutlich zu erkennen, dass die Zeit, die auf das Schreiben des Protokolls gewartet wurde, so gut wie identisch mit derjenigen Zeit ist, welche die Abfrage insgesamt gedauert hat. Dies war auch nicht anders zu erwarten, denn das Schreiben des Protokolls geschieht synchron. Ein `COMMIT` ist also erst dann abgeschlossen, wenn die entsprechenden Protokolleinträge in der Protokolldatei gespeichert sind.

Wir wiederholen nun das Experiment mit einem etwas geänderten Skript:

```
if (object_id('T1') is not null)
  drop table T1
go
-- Erzeuge eine Test-Tabelle
create table T1
 (
   id int not null default 0
  ,daten nchar(20) null
 )
go
set nocount on
-- Setze alle Leistungsindikatoren zurück.
dbcc sqlperf('sys.dm_os_wait_stats',clear) with no_infomsgs
-- Füge 10.000 Zeilen in einer Transaktion hinzu
begin tran
  declare @i int
  set @i = 1
  while (@i < 10000)
   begin
    insert T1(id) values(@i)
    if (@i % 1000) = 0
      raiserror('1000 Zeilen hinzugefügt', 0, 1) with nowait
    set @i = @i + 1
  end
commit
-- Zeige an, wie oft und wie lange auf des Schreiben
-- des Transaktionsprotokolls gewartet wurde.
select * from sys.dm_os_wait_stats
 where wait_type = 'WRITELOG'
```

Schauen Sie bitte auf den fett gedruckten Teil im obigen Listing. Nun werden alle `INSERT`-Befehle in nur einer einzigen Transaktion ausgeführt. Es wird also auch nur ein einziger Protokolleintrag geschrieben; die Anzahl der `WRITELOG`-Wartezustände ist folglich 1.

Die Werte für die Ausführung der Abfrage liegen nun allesamt deutlich unter einer Sekunde, wobei es egal ist, ob der Schreibcache der Festplatte aktiv ist oder nicht.

Und damit haben wir einen weiteren Ansatzpunkt für eine Optimierung: Die Transaktionsgröße, oder auch die Transaktionsrate. Aus dem Ergebnis des obigen Experiments könnten Sie schließen, dass möglichst große Transaktionen eine Verbesserung der Abfrageleistung mit sich bringen. Ganz so einfach ist es aber leider nicht. Im Experiment haben wir

einen Extremfall untersucht, um zu demonstrieren, welche Auswirkungen die Transaktionsrate auf die Abfrageleistung bei Datenänderungen hat. Größere und länger andauernde Transaktionen benötigen auch mehr Sperren, die länger aufrechterhalten werden. Dadurch erzeugen sie in der Regel viel mehr Blockierungen, wodurch in Mehrbenutzersystemen vermehrt Wartezustände auftreten. Hierdurch sinkt letztlich die Leistung.

Wie so oft, gilt es also auch hier, ein Optimum herauszufinden. Der Indikator `WRITELOG` ist hier für eine entsprechende Überwachung sehr gut geeignet. Falls die Wartezeit für `WRITELOG` in der Hitliste der gesamt gewarteten Zeit weit oben steht, wobei Sie natürlich Ihr System über einen längeren Zeitraum hin überwachen sollten, dann lohnt sich sicherlich eine nähere Untersuchung.

2.4 Zusammenfassung

Dieses Kapitel hat Ihnen die Grundlagen der SQL Server-Datenverwaltung aufgezeigt. Da die Minimierung von Ein-/Ausgabeoperationen bei der Optimierung von Abfragen im Vordergrund steht, ist es natürlich wichtig, dass Sie wissen, in welcher Art und Weise SQL Server Daten liest und schreibt.

Insbesondere sollten Sie sich einprägen, dass die Verwaltung von Tabellen- und Protokolldaten auf unterschiedliche Weise erfolgt. Tabellendaten werden in der Regel zufällig gelesen und geschrieben, wohingegen Protokolldaten normalerweise nur sequenziell geschrieben werden.

SQL Server verwaltet einen Datencache, in den alle von der Festplatte gelesenen Daten übertragen werden. Der Abfrageprozessor bedient sich stets aus diesem Cache. Auch Datenänderungen werden zunächst nur im Cache gespeichert und von dort asynchron auf die Festplatte übertragen.

3 Ausführung von Abfragen

Nach dem in Kapitel 1 vorgestellten Optimierungsmodell spielt die Optimierung von Abfragen eine entscheidende Rolle bei der Performance-Optimierung. Damit Sie wissen, welche Ansatzpunkte sich hier bieten, ist es wichtig zu verstehen, in welchen Schritten die Verarbeitung von Abfragen durchgeführt wird, das heißt, wie SQL Server also letztlich aus einer SQL-Anweisung ein Abfrageergebnis ermittelt.

In diesem Kapitel werden Sie die an der Verarbeitung von Abfragen beteiligten Komponenten kennenlernen, wobei der Schwerpunkt auf dem Abfrageoptimierer liegt. Ausgangspunkt ist hierbei zunächst die logische Ausführung einer Abfrage, die sich direkt aus der SQL-Anweisung ergibt. Wir werden anschließend untersuchen, wie aus einer SQL-Anweisung in verschiedenen Schritten ein physikalischer Ausführungsplan erstellt wird, der das logisch korrekte Abfrageergebnis auf dem (hoffentlich) günstigsten Weg ermittelt.

3.1 Logische Schritte bei der Abfrageausführung

Wenn Sie eine Abfrage entwerfen und ausführen, haben Sie sich (hoffentlich) zuvor überlegt, wie das Ergebnis aussehen soll und welche Daten aus welchen Tabellen Sie benötigen. Hierbei werden Sie auch bedenken, in welcher Weise Ihre Abfrage ausgeführt wird, wobei Sie hier die logische Reihenfolge der Abarbeitung der einzelnen Klauseln berücksichtigen müssen. Mit anderen Worten: Sie sollten wissen, welche Schritte in welcher Reihenfolge erforderlich sind, um ein korrektes Abfrageergebnis zu erhalten.

Wir wollen hierfür ein Beispiel untersuchen. Betrachten Sie bitte die folgende Abfrage, die alle Produktunterkategorien mit Namen und der Anzahl der in ihr enthaltenen Produkte zurückgibt, wobei nur Unterkategorien berücksichtigt werden, die im Namen die Zeichenkette »bike« enthalten, und die mehr als fünf Produkte umfassen:

```
-- Alle Produktkategorien, die mehr als 5 Produkte enthalten
-- und deren Name die Zeichenfolge "bike" enthält.
use AdventureWorks2008;
select sc.Name, count(*) as Anzahl
  from Production.Product as p
       left outer join Production.ProductSubcategory as sc
                  on sc.ProductSubCategoryID = p.ProductSubcategoryID
 where sc.Name like '%bike%'
 group by sc.Name
  having count(*) > 5
  order by Anzahl desc
```

Die einzelnen Bereiche der Abfrage, wie zum Beispiel `FROM`, `WHERE` oder `ORDER BY` werden logisch in einer genau festgelegten Reihenfolge nacheinander abgearbeitet, damit das Abfrageergebnis korrekt ist. Abbildung 3.1 zeigt, in welcher Reihenfolge die einzelnen Schritte der Abfrage ausgeführt werden.

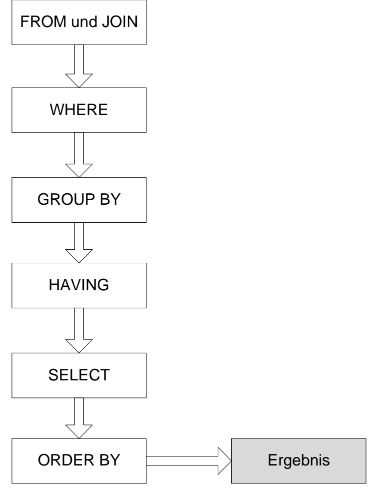


Abbildung 3.1: *Logische Abfragereihenfolge*

Aus Gründen der Übersichtlichkeit haben wir in unserer Abfrage einige weniger wichtige Klauseln weggelassen (zum Beispiel: `DISTINCT`, `TOP`, `CUBE` und `ROLLUP`). Es geht hier nur darum, dass Sie sich die folgenden beiden Punkte einprägen:

- Die logische Ausführungsreihenfolge ist deterministisch. Mit anderen Worten: Wenn Sie eine Abfrage ausführen, so ist immer sichergestellt, dass das Ergebnis der logischen Abarbeitungsreihenfolge entspricht.
- Die logische Abarbeitungsreihenfolge entspricht nicht der Reihenfolge des Auftretens der Klauseln in einer `SELECT`-Anweisung. Für SQL-Neulinge ist diese Tatsache stets gewöhnungsbedürftig. Die `SELECT`-Klausel beispielsweise wird erst beinahe erst zum Schluss ausgewertet, obwohl sie am Beginn einer Abfrage steht.

Sie können sich die Verarbeitung so vorstellen, dass eine virtuelle Tabelle von Schritt zu Schritt »durchgereicht« wird. Jeder Verarbeitungsschritt verändert die virtuelle Tabelle, indem er Zeilen bzw. Spalten hinzufügt oder entfernt und anschließend die modifizierte Tabelle an den nächsten Verarbeitungsschritt übergibt, bis zum Schluss das Ergebnis feststeht.

Aus der Ausführungsreihenfolge lassen sich einige Besonderheiten feststellen: Zunächst einmal werden bei Abfragen, die Gruppierungen verwenden, zwei Zeilenfilter ausgewertet: `WHERE` und `HAVING`. Da die `WHERE`-Klausel deutlich vor der `HAVING`-Klausel ausgewertet wird, ist es generell günstiger, eine Filterung über die `WHERE`-Klausel durchzuführen, da die Ergebnismenge in diesem Fall bereits sehr früh eingeschränkt wird und die nachfolgenden Schritte dadurch weniger Zeilen verarbeiten müssen. Hinzu kommt noch, dass für die in der `WHERE`-Klausel angegebenen Bedingungen Indizes verwendet werden kön-

nen, wodurch die Abfrageleistung erheblich verbessert werden kann. (Hierauf kommen wir in den Kapiteln 5 und 7 noch zurück.) Da die `HAVING`-Klausel berechnete Werte auswertet, können für diese Art der Filterung keine Indizes verwendet werden. Wann immer möglich, sollten Sie also `WHERE` verwenden, um die Ergebnismenge bereits vor der Gruppierung einzuschränken.

Ein weiterer Punkt, der vor allem SQL-Neulinge regelmäßig in Erstaunen versetzt, betrifft OUTER JOINs. Es ist so, dass die Auswertung von JOIN-Bedingungen ganz am Anfang der Ausführungsreihenfolge steht. Die `WHERE`-Klausel wird erst nach dem Ergebnis des OUTER JOIN ausgewertet. Da im Ergebnis eines OUTER JOIN immer alle Zeilen der inneren Tabelle enthalten sind (auch dann, wenn keine äußeren Zeilen gefunden werden; in diesem Fall sind die Spaltenwerte für die äußere Tabelle sämtlich `NULL`), existiert dadurch beispielsweise auch die Möglichkeit, das Ergebnis in der `WHERE`-Klausel hinsichtlich `NULL`-Werten in der äußeren Tabelle einzuschränken. Aus dieser Tatsache ergibt sich die Konsequenz, dass die JOIN-Bedingung in einem INNER JOIN wahlweise über die `ON`-Klausel oder über die `WHERE`-Klausel angegeben werden kann. Beide Varianten liefern stets das gleiche Ergebnis. Bei einem OUTER JOIN ist dies nicht so. Hier hat die über die `WHERE`-Klausel angegebene Bedingung eine andere Bedeutung, als wenn die Bedingung über die `ON`-Klausel des OUTER JOIN angegeben wird.

Betrachten Sie bitte das folgende Beispiel:

```
-- OUTER JOIN: Die Abfragen liefern verschiedene Ergebnisse!
select p.Name as ProductName
      ,sc.Name as SubcategoryName
  from Production.Product as p
       left outer join Production.ProductSubcategory as sc
                    on sc.ProductSubcategoryID = p.ProductSubcategoryID
                   and sc.Name like '%bike%'
-----------------------------------------
select p.Name as ProductName
      ,sc.Name as SubcategoryName
  from Production.Product as p
       left outer join Production.ProductSubcategory as sc
                    on sc.ProductSubcategoryID = p.ProductSubcategoryID
 where sc.Name like '%bike%'
```

Beide Abfragen verwenden einen OUTER JOIN, wobei in der ersten Abfrage eine Filterung in der `ON`-Klausel der JOIN-Bedingung und in der zweiten Abfrage eine Filterung in der `WHERE`-Klausel des `SELECT` erfolgt. Die beiden Abfragen sind logisch nicht äquivalent! In der ersten Abfrage erfolgt keine Filterung im Sinne einer Einschränkung von Zeilen. Stattdessen bewirkt die Bedingung in der `ON`-Klausel hier, dass alle Spaltenwerte für die Ausgabespalte `SubcategoryName`, die nicht die Zeichenfolge »bike« enthalten, `NULL` sind. In der zweiten Abfrage hingegen wird das Ergebnis über die `WHERE`-Klausel reduziert. Hier werden tatsächlich alle Zeilen aus dem Ergebnis entfernt, die nicht die Zeichenfolge »bike« in der Spalte `SubcategoryName` enthalten. Die zuvor über den OUTER JOIN hinzugefügten Zeilen mit `NULL`-Werten in der Spalte `SubcategoryName` werden dadurch wieder aus dem Ergebnis gestrichen – das Ergebnis entspricht dadurch dem eines INNER JOIN.

Sie könnten nun vermuten, dass ein OUTER JOIN generell »teurer« ist als ein INNER JOIN, da bei einem OUTER JOIN mehr Zeilen verarbeitet werden müssen. Generell trifft diese Aussage zu. Wann immer möglich, sollten Sie einen OUTER JOIN vermeiden und durch einen günstigeren INNER JOIN ersetzen. Falls Sie dennoch OUTER JOINs verwenden, die durch einen entsprechenden INNER JOIN abgebildet werden können, ist dies in der Regel nicht so dramatisch. Der Optimierer erkennt eine solche Situation und übernimmt die Ersetzung für Sie. Hierauf kommen wir später noch einmal zurück, wenn wir die Arbeitsweise des Optimierers ein wenig genauer untersuchen.

Bei einem INNER JOIN ist es unerheblich, ob Sie die Zeilen in der `ON`-Klausel der JOIN-Bedingung oder über die `WHERE`-Klausel herausfiltern. Die folgenden beiden Abfragen sind daher logisch äquivalent:

```
-- INNER JOIN: Beide Abfragen liefern dasselbe Ergebnis.
select p.Name as ProductName
      ,sc.Name as SubcategoryName
  from Production.Product as p
       inner join Production.ProductSubcategory as sc
               on sc.ProductSubcategoryID = p.ProductSubcategoryID
              and sc.Name like '%bike%'
------------------------------------------
select p.Name as ProductName
      ,sc.Name as SubcategoryName
  from Production.Product as p
       inner join Production.ProductSubcategory as sc
               on sc.ProductSubcategoryID = p.ProductSubcategoryID
 where sc.Name like '%bike%'
```

Kapitel 9 beschäftigt sich ausführlich mit physikalischen JOIN-Operationen.

3.2 Physikalischer Ausführungsplan

Damit der Abfrageprozessor eine Abfrage verarbeiten kann, benötigt er einen entsprechenden physikalischen Ausführungsplan. Dieser Plan wird aus dem Abfragetext in drei Schritten und durch drei unterschiedliche Komponenten erstellt.

3.2.1 Parser

Der erste Schritt bei der Erstellung des Abfrageplans wird – wie bei Übersetzungsvorgängen üblich – durch einen Parser ausgeführt. Hierbei werden unter anderem Syntaxüberprüfungen durchgeführt und Namen (zum Beispiel von Tabellen und Spalten) auf eine korrekte Schreibweise hin überprüft. Der Parser erstellt letztlich einen Baum, der die Ausführungslogik der Abfrage repräsentiert.

3.2.2 Algebrizer

Der vom Parser erzeugte Baum wird vom sogenannten *Algebrizer* im zweiten Schritt weiterverarbeitet. In diesem Schritt werden unter anderem Spalten- und Tabellennamen überprüft und zugeordnet sowie Datentypen auf ihre Korrektheit hin kontrolliert. Der vom Parser erzeugte Baum wird weiter vereinfacht und normalisiert – zum Beispiel durch das Entfernen redundanter Operationen. Das Ergebnis ist am Ende ein vereinfachter Baum, der die Abfrage logisch widerspiegelt. Dieser Baum dient schließlich als Eingabe für den Optimierer.

3.2.3 Optimierer

Der Optimierer ermittelt für eine Abfrage einen physikalischen Ausführungsplan. Hierzu kann der Optimierer aus einer Vielzahl von physikalischen Operatoren auswählen, auf die wir im weiteren Verlauf dieses Buches noch zurückkommen werden. Der Optimierer legt bei der Erstellung des Plans unter anderem fest, welche Indizes verwendet werden, in welcher Reihenfolge auf die beteiligten Tabellen zugegriffen wird, oder wie Verknüpfungen (Joins) physikalisch ausgeführt werden. Sie können sich sicherlich leicht vorstellen, dass – gerade bei komplexeren Abfragen – nicht nur eine Möglichkeit existiert, einen solchen physikalischen Ausführungsplan zu erstellen. Für die meisten Abfragen gibt es mehrere Möglichkeiten; bei komplexen Abfragen können dies sogar einige Millionen sein. Der Optimierer erstellt mehrere (aber in der Regel nicht alle möglichen) Ausführungspläne, die alle gültig sind, also alle zum selben korrekten Ergebnis führen. Unter allen gefundenen Ausführungsplänen wird dann letztlich derjenige mit den geringsten »Kosten« ausgewählt. Dies ist das Prinzip eines kostenbasierten Optimierers. Der einzige Kostenfaktor, der hierbei berücksichtigt wird, ist die Zeit, die benötigt wird, um das Abfrageergebnis zu ermitteln. Etwas weiter unten werden Sie sehen, in welcher Weise Sie sich die Abfragekosten anzeigen lassen können.

Die vom Optimierer ausgegebenen erwarteten Abfragekosten werden allerdings nicht etwa in einer uns bekannten Zeiteinheit, also zum Beispiel in Sekunden, ausgegeben. Stattdessen verwendet der Optimierer ganz offensichtlich eine eigene Uhr, welche die Zeit in einer nur dem Optimierer bekannten Zeiteinheit misst. Verstehen Sie also bitte die ausgegebenen Abfragekosten als eine Art Kennzahl, die einen direkten Bezug zur Ausführungsdauer der Abfrage besitzt (aber keinesfalls linear ist). Generell gilt hier die Aussage, dass niedrigere Kosten für kürzere Ausführungszeiten stehen.

Der Optimierer wird also den Plan mit der vermeintlich geringsten Ausführungszeit auswählen. Hierbei ist es nicht etwa so, dass dieser Plan generell auch den geringsten Ressourcenverbrauch aufweist. Es kann durchaus sein, dass der ausgewählte Ausführungsplan mehr CPU und/oder mehr E/A-Vorgänge benötigt als ein vergleichbarer Plan, der vom Optimierer verworfen wurde. Ausschlaggebend ist allein die geschätzte Ausführungszeit.

Auf Grund der Komplexität ist die Optimierung zugleich der wichtigste und auch kompliziertestе Teil bei der Erstellung eines physikalischen Plans für die Abfrageausführung. Hierbei kann die physikalische Ausführungsreihenfolge durchaus von der logischen Reihenfolge abweichen. So ist es zum Beispiel bei einem INNER JOIN unerheblich, in welcher Reihenfolge auf die beteiligten Tabellen zugegriffen wird. Auch die Anwendung zusätzlicher Filterbedingungen kann bei einem INNER JOIN vor oder nach der Verknüpfung erfolgen – das Ergebnis der Abfrage ist in beiden Fällen identisch.

Für die Erstellung des Plans zieht der Optimierer diverse Kriterien in Betracht. Einerseits muss selbstverständlich die logische Ausführungsreihenfolge berücksichtigt werden, also letztlich der Text der SQL-Anweisung. Hinzu kommt, dass der Optimierer auch auf Datenverteilungsstatistiken zurückgreift, um Kardinalitätsschätzungen vorzunehmen. Anhand dieser Schätzungen werden Operatoren für die einzelnen Schritte des Plans ausgewählt. Um zum Beispiel einen JOIN auszuführen, kann der Abfrageprozessor aus diversen physikalischen JOIN-Operatoren auswählen. Die Wahl hängt wesentlich von der Abschätzung der Zeilenanzahl der am JOIN beteiligten Tabellen ab. Anders gesagt: Die durch einen JOIN-Operator erzeugten Kosten werden durch die Anzahl der beteiligten Zeilen bestimmt. In Kapitel 9 gehen wir hierauf genauer ein.

Die Erstellung eines Abfrageplans wird oftmals auch als Kompilieren der Abfrage bezeichnet. Hiermit sind alle Phasen der Planerstellung gemeint, also vom Parsen bis zum Optimieren. Die Optimierung ist hierbei derjenige Schritt, der am kostspieligsten ist. Wie bei jedem Kompilier-Vorgang erzeugt dieser Vorgang vor allem Prozessorlast. Hierbei kann es durchaus vorkommen, dass die Prozessoren zum »Flaschenhals« werden, falls Ihre Anwendungen die Abfragen so stellen, dass viele Übersetzungsvorgänge erforderlich sind.

SQL Server trifft einige Vorkehrungen, um die durch Kompilier-Vorgänge erzeugte Prozessorlast zu minimieren. Hierzu zählt unter anderem das Cachen von kompilierten Abfrageplänen zur späteren Wiederverwendung ohne vorherige (erneute) Übersetzung. Diese Thematik greifen wir ebenfalls in Kapitel 9 auf.

Eine weitere Maßnahme zur Minimierung der Prozessorlast ist die Art und Weise der Erstellung eines Abfrageplans. Wie bereits erwähnt, gibt es bei komplexen Abfragen eine Vielzahl von in Frage kommenden physikalischen Plänen. Dadurch ist es in den meisten Fällen nicht möglich, dass die Kosten aller in Frage kommenden Pläne miteinander verglichen werden, um den günstigsten Plan zu finden. Vielmehr wendet der Optimierer diverse Heuristiken an, die darauf abzielen, dass mit einer möglichst geringen Anzahl erstellter Pläne der optimale Plan gefunden werden kann. Andernfalls wäre die Erstellung eines Abfrageplans in den meisten Fällen teurer als die eigentliche Ausführung der Abfrage. Diese Tatsache sollten Sie sich gut einprägen. In der Regel sind die angewandten Heuristiken vollkommen ausreichend, um einen sehr guten Plan, also einen, der möglichst nahe am theoretisch möglichen Optimum liegt, zu finden. Genau genommen wird jedoch in vielen Fällen nur ein »fast optimaler« Plan erstellt. Hierbei kann es durchaus auch einmal vorkommen, dass ein solcher Plan Kosten verursacht, die deutlich über den des Optimums liegen.

Um einen möglichst kostengünstigen Plan zu finden, durchläuft die Optimierung verschiedene Phasen:

Trivialer Plan

Die Optimierung beginnt mit der Prüfung, ob ein trivialer Plan existiert. Ist die Abfrage einfach genug, so existiert unter Umständen nur ein einziger möglicher Abfrageplan. In diesem Fall ist die Optimierung mit dem Auffinden dieses trivialen Plans abgeschlossen. Betrachten Sie hierzu bitte die folgende Abfrage:

```
select Color
  from Production.Product
```

Hier ist keine Optimierung erforderlich. Es werden einfach alle `Color`-Spaltenwerte aller Zeilen der Tabelle gelesen. Der Optimierer erkennt, dass es nur einen vernünftigen Plan gibt und verwendet diesen.

Ein weiteres Beispiel für einen trivialen Plan ist die folgende Abfrage:

```
select Color
  from Production.Product
 where Name is null
```

Die in der `WHERE`-Klausel angegebene Bedingung kann niemals wahr werden, weil durch einen entsprechenden `CHECK`-Constraint deklariert wurde, dass die Spalte `Name` nicht `NULL` werden kann. Folglich wird die Abfrage immer ein leeres Ergebnis zurückliefern. Der Optimierer erkennt dies, weil er auch die Meta-Daten untersucht, und erzeugt einen trivialen Plan, der keinerlei Zugriffe auf Tabellendaten erfordert.

Vereinfachung

Falls kein trivialer Plan existiert, erfolgt im zweiten Schritt der Versuch einer Vereinfachung der Abfrage. Dies geschieht größtenteils durch eine syntaktische Umwandlung oder eine Neuanordnung der Operationen. Zum Beispiel erfolgt eine Konvertierung von OUTER JOINs in günstigere INNER JOINs, sofern dies möglich ist.

Schauen Sie sich hierzu bitte die folgende Abfrage an:

```
-- Alle Produktkategorien, die mehr als fünf Produkte enthalten
-- und deren Name die Zeichenfolge "bike" enthält.
select sc.Name, count(*) as Anzahl
  from Production.Product as p
       left outer join Production.ProductSubcategory as sc
         on sc.ProductSubCategoryID = p.ProductSubcategoryID
 where sc.Name like '%bike%'
 group by sc.Name
```

Durch die `WHERE`-Bedingung werden alle Zeilen, in denen die Spalte `sc.Name` den Wert `NULL` enthält, aus dem Ergebnis entfernt. Dies sind aber gerade diejenigen Zeilen, die durch den OUTER JOIN hinzugefügt wurden. Folglich ist der OUTER JOIN in diesem Fall identisch mit einem INNER JOIN; der Optimierer verwendet somit den wesentlich günstigeren INNER JOIN-Operator.

Ein weiteres Beispiel für eine Vereinfachung ist das Anwenden von Filteroperationen zu einem möglichst frühen Zeitpunkt, da die nachfolgenden Schritte dann entsprechend weniger Zeilen verarbeiten müssen.

Auch das Entfernen von unnötigen Tabellen übernimmt der Optimierer bei der Vereinfachung. Sehen Sie sich bitte dazu die folgende Abfrage an:

```
select distinct sc.Name
  from Production.Product as p
       right outer join Production.ProductSubcategory as sc
                    on sc.ProductSubCategoryID = p.ProductSubcategoryID
 where sc.Name like '%bike%'
```

Die Tabelle `Production.Product` ist überflüssig, da von ihr keine Spalten in der Abfrage benötigt werden und sie über den JOIN auch nicht zu einer Einschränkung der Ausgabezeilen führt. Daher nimmt der Optimierer die Tabelle erst gar nicht in seinen Plan auf.

Erstellung mehrerer Pläne und Kostenvergleich

Nach der Vereinfachung erstellt der Optimierer einige Abfragepläne, wobei zunächst nur einfache Optimierungsschritte, wie zum Beispiel das Vertauschen von Tabellenreigenfolgen in JOINs, durchgeführt werden. Sobald hierbei ein Plan gefunden wird, dessen Kosten kleiner als 0,2 sind, ist die Optimierung beendet. Das Ende der Optimierung ist auch dann erreicht, wenn alle möglichen Pläne untersucht wurden. Dies gilt auch für die noch folgenden Schritte. Bei weniger komplexen Abfragen kann dieser Fall durchaus eintreten.

Weitere Versuche mit erhöhter Kostenschwelle

Konnte im ersten Schritt kein Plan gefunden werden, so erfolgt im Anschluss eine erweiterte Optimierung. Hierbei werden zum Beispiel mehr Vertauschungen vorgenommen, als im ersten Schritt. Sobald ein Plan gefunden wird, dessen Kosten kleiner als 1,0 sind, wird dieser Plan ausgewählt, und die Optimierung ist abgeschlossen.

Testen paralleler Ausführungspläne

Bis hierher wurden noch keine Pläne erstellt, die eine parallele Verarbeitung durchführen. Falls der Computer über mehr als eine CPU verfügt und die Kosten für den bislang gefundenen günstigsten Plan höher sind als durch den Konfigurationsparameter *cost treshold for parallelism* angegeben (der Standardwert ist 5), wird die vorherige Optimierungsphase wiederholt. Diesmal ist das Ziel der Optimierung aber die Erstellung eines parallelen Abfrageplanes, also eines Planes, der mehrere Prozessoren verwendet. Anschließend wird für den günstigeren der beiden Pläne (parallel oder nicht parallel) eine weitere volle Optimierungsphase eingeleitet. Hierbei werden dann zum Beispiel auch indizierte Sichten berücksichtigt.

Generell kann der gefundene Ausführungsplan in drei Kategorien eingeteilt werden:

- Der Plan ist trivial. Eine Optimierung war nicht erforderlich, da es nur einen Plan gibt. Damit ist der Plan zugleich auch optimal.
- Der Plan ist optimal. Alle möglichen Ausführungsvarianten konnten geprüft werden; der günstigste Plan hat gewonnen.

- Der Plan ist fast optimal. Für die Überprüfung aller möglichen Ausführungsvarianten ist die Abfrage zu komplex. Aus diesem Grund wurde die Optimierung entsprechend der oben geschilderten Heuristiken verkürzt. Der hierdurch gefundene Plan muss nicht das Optimum sein, er wird jedoch als fast optimal angesehen.

Physikalische Operatoren

Für die Erstellung eines Abfrageplans kann der Optimierer aus über 100 unterschiedlichen physikalischen Operatoren die passenden auswählen. Ich möchte Ihnen an dieser Stelle die reine Auflistung dieser Operatoren ersparen, eine solche Liste finden Sie in der Online-Dokumentation. Wir werden im weiteren Verlauf dieses Buches sehr häufig mit Ausführungsplänen arbeiten und an den entsprechenden Stellen auch etwas zu den verwendeten Operatoren sagen. Haben Sie bitte noch etwas Geduld. – Bereits im folgenden Kapitel werden wir die ersten Abfragepläne untersuchen. An dieser Stelle sollen zunächst nur einige einleitende und allgemeine Kommentare gegeben werden.

Generell verarbeiten alle Operatoren eine gewisse Anzahl von Eingabezeilen und produzieren entsprechende Ausgabezeilen. Die von einem Operator produzierte Ausgabe bildet dann die Eingabe für einen anderen Operator. Hierbei werden prinzipiell zwei Arten von Operatoren unterschieden: Die eine Sorte verarbeitet eintreffende Zeilen sofort und leitet sie an den nächsten Operator weiter. Hierzu zählt zum Beispiel der Filter-Operator, der für jede Zeile unmittelbar ausgewertet wird. Auf der anderen Seite stehen die sogenannten »Stop And Go«-Operatoren, die erst dann eine Ausgabe produzieren können, wenn alle Eingabezeilen vorliegen. Ein Beispiel hierfür ist der Operator zur Sortierung, der natürlich erst dann die Ausgabezeilen erzeugen kann, wenn alle zu sortierenden Zeilen vorliegen.

3.2.4 Anzeigen des Ausführungsplans

SQL Server stellt eine ganze Reihe von Analysemöglichkeiten zur Verfügung, die eine Inspektion der Zustände und einen Einblick die Arbeitsweise ermöglichen. Unter diesen Möglichkeiten sind auch einige, die eine Darstellung bzw. Untersuchung von Abfrageplänen gestatten. Der vom Optimierer erstellte Plan ist also keinesfalls geheim, sondern kann auf unterschiedliche Weise angezeigt werden. Die Ausgabe von Abfrageplänen kann allgemein in zwei Kategorien eingeteilt werden:

- **Anzeige des geschätzten Ausführungsplanes**. Hierbei wird die eigentliche Abfrage nicht ausgeführt. Sie erhalten nur den Abfrageplan, also das Ergebnis der Optimierung. Diese Möglichkeit ist zum Beispiel dann nützlich, wenn Sie den Plan für eine lang dauernde Abfrage sehen möchten. Sie müssen dann nicht jedesmal auf das Ergebnis der Abfrage warten.
- **Anzeige des tatsächlichen Ausführungsplans**. Bei Wahl dieser Option wird die Abfrage ausgeführt, und der Abfrageplan wird anschließend angezeigt. Der tatsächliche Ausführungsplan enthält etwas mehr Informationen als der geschätzte Ausführungsplan. Zum Beispiel finden Sie im tatsächlichen Ausführungsplan auch Angaben über die genaue Anzahl verarbeiteter Zeilen – eine Information, die im geschätzten Ausführungsplan nicht enthalten ist. Dass der Plan generell vor der Ausführung einer Abfrage erstellt wird, gilt natürlich gleichermaßen auch für den tatsächlichen Ausführungsplan. Daher ist der Plan, also die verwendeten Operatoren und die Reihenfolge der Abarbeitung, mit dem geschätzten Ausführungsplan identisch.

Anzeige des Ausführungsplans in grafischer Form

Dies ist sicherlich die beliebteste und meistverwendete Möglichkeit. Der Abfrageplan wird als Graph angezeigt, wobei die Knoten des Graphen die Operatoren repräsentieren und die Kanten den Datenfluss veranschaulichen.

Die Anzeige des geschätzten oder tatsächlichen Ausführungsplans können Sie zum Beispiel über die Menüleiste im Management Studio ein- bzw. ausschalten (Abbildung 3.2).

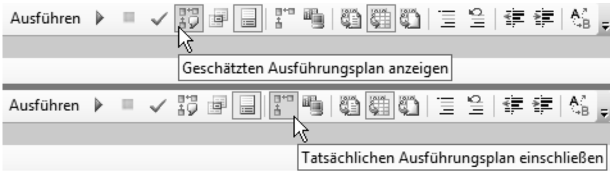


Abbildung 3.2: *Anzeigen des grafischen Ausführungsplans*

Ist eine Option gewählt, dann erfolgt für jede danach ausgeführte Abfrage in dem geöffneten Fenster die Ausgabe des Abfrageplans in einem separaten Reiter. Abbildung 3.3 zeigt ein Beispiel für einen grafischen Abfrageplan.

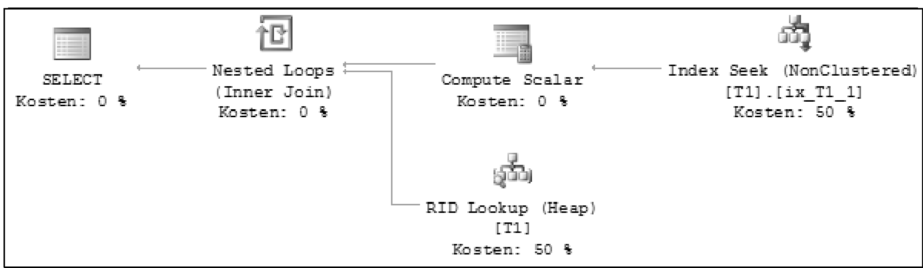


Abbildung 3.3: *Ein einfacher Abfrageplan*

Der Abfrageplan wird von rechts nach links und von unten nach oben gelesen. Der oben links stehende Knoten repräsentiert also den zuletzt ausgeführten Operator – in unserem Fall die `SELECT`-Anweisung. Die Pfeile stehen für den Datenfluss, wobei die Stärke eines Pfeils ein Indiz für die Anzahl der Zeilen ist; je dicker also ein Pfeil ist, desto mehr Zeilen werden von rechts nach links übertragen. Für jeden enthaltenen Operator wird außerdem der prozentuale Kostenanteil angezeigt, den diese Operation in Bezug auf die gesamte Abfrage verursacht.

Sobald Sie die Maus über einen Operator oder einen Pfeil bewegen, öffnet sich ein Fenster mit näheren Informationen zum Operator oder Datenfluss. Abbildung 3.4 zeigt dies an einem Beispiel.

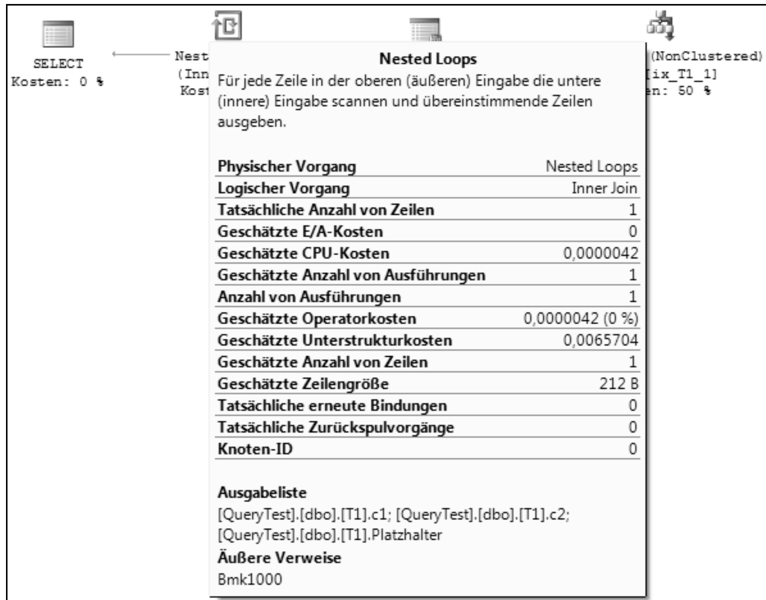


Abbildung 3.4: *Informationen über den Nested Loops-Operator*

Hier sehen Sie zum Beispiel auch die absoluten Kosten des gesamten Zweiges, also die Kosten des Operators und aller weiter rechts stehenden Operationen. Der linke obere Knoten – also in unserem Fall die `SELECT`-Anweisung – enthält dann die Kosten des gesamten Plans.

Bei komplexeren Abfragen kann der grafische Ausführungsplan sehr schnell ziemlich groß und damit auch unübersichtlich werden. Dann erfordert es schon einige Übung, den Plan zu interpretieren und eventuelle Schwachstellen aufzufinden. Ich kann Ihnen hier nur empfehlen, dass Sie sich nach und nach in diese Materie einarbeiten. Die hierfür investierte Zeit ist wirklich gut angelegt. Wir werden auch im weiteren Verlauf sehr viel mit Ausführungsplänen arbeiten. Allerdings werden hierbei sehr komplexe Ausführungspläne, mit deren Ausdruck man eine ganze Wand tapezieren könnte, schon allein aus Gründen der Formatierung nicht betrachtet.

Anzeige des Ausführungsplans in Textform

Die Ausführungspläne können auch textuell angezeigt werden – eine Möglichkeit von der wir in diesem Buch allerdings keinen Gebrauch machen werden. Die textuelle Form ist vor allem für den Austausch nützlich, wenn Sie also beispielsweise einen Plan in einer Newsgroup veröffentlichen und diskutieren möchten.

Auch für die Textform können Sie zwischen dem geschätztem und dem tatsächlichem Ausführungsplan wählen. Hierzu stehen Ihnen die folgenden SET-Befehle zur Verfügung:

- SET SHOWPLAN_TEXT ON. Es wird der geschätzte Ausführungsplan in einem Kurzformat angezeigt. Kurzformat deshalb, weil in diesem Plan nur die Operatoren ohne geschätzte Kosten enthalten sind.
- SET SHOWPLAN_ALL ON. Auch hier wird der geschätzte Ausführungsplan angezeigt. Der Plan enthält neben den Operatoren auch die geschätzten Kosten, ist also umfangreicher und aussagekräftiger als bei SHOWPLAN_TEXT.
- SET STATISTICS PROFILE ON. Hier wird der tatsächliche Ausführungsplan angezeigt, die Abfrage wird also ausgeführt. Daher werden hier auch Informationen über die Anzahl der tatsächlich verarbeiteten Zeilen ausgegeben.

Anzeige des Ausführungsplans im XML-Format

Ab der SQL Server-Version 2005 können Ausführungspläne auch im XML-Format ausgegeben werden. Diese Variante ist besonders deshalb interessant, weil das Management Studio Abfragepläne, die im XML-Format vorliegen, für die Darstellung in das grafische Format überführen kann. Dadurch erhalten Sie die eine einfache Möglichkeit, Abfragepläne auszutauschen. Hierfür müssen Sie lediglich XML-Dateien versenden bzw. empfangen.

Die folgenden beiden SET-Befehle erzeugen den XML-Plan als Ausgabe:

- SET SHOWPLAN_XML ON. Es wird der geschätzte Ausführungsplan angezeigt; die Abfrage wird also nicht ausgeführt.
- SET STATISTICS XML ON. Bei dieser Option erhalten Sie den tatsächlichen Ausführungsplan. Dieser Plan wird als zusätzliches Abfrageergebnis ausgegeben.

In beiden Fällen können Sie einfach durch einen Klick auf das ausgegebene XML-Dokument den Plan sofort in das grafische Format überführen. Auch die umgekehrte Transformation ist möglich: Wenn Sie den Ausführungsplan in grafischer Form erstellt haben, können Sie aus dem Kontextmenü dieses Plans das XML-Format erstellen (Abbildung 3.5).

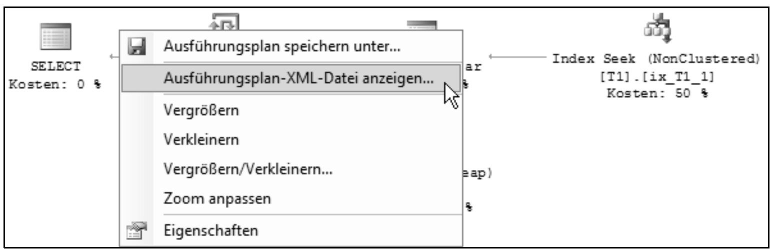


Abbildung 3.5: *Den grafischen Ausführungsplan in das XML-Format überführen*

Seien Sie bitte vorsichtig mit DDL-Anweisungen, sofern Sie eine Option gewählt haben, die nur einen geschätzten Ausführungsplan anzeigt. Da die DDL-Anweisungen in diesem Fall nicht ausgeführt werden, können Ihre T-SQL-Stapel ein eigenartiges Verhalten an den Tag legen.

Schauen Sie sich zum Beispiel das folgende Skript an:

```
set showplan_text on
go
create table T1(c1 int)
go
insert t1 values(1)
```

Da die Tabelle `T1` nicht erzeugt wird, kann für die `INSERT`-Anweisung kein Ausführungsplan erstellt werden. Stattdessen erhalten Sie einen Übersetzungsfehler, der besagt, dass es keine Tabelle `T1` gibt.

3.3 Zusammenfassung

Dieses Kapitel hat Ihnen einige wesentliche Grundlagen über die Ausführung von Abfragen vermittelt. Insbesondere die Analyse von Ausführungsplänen ist eine äußerst nützliche Methode, wenn es gilt, Performance-Engpässe aufzuspüren und zu beseitigen. Diese Methode wird Sie durch den Rest des Buches begleiten. Es ist daher im Moment auch nicht allzu schlimm, wenn bei Ihnen diesbezüglich noch einige Fragen offen geblieben sind. In den verbleibenden Kapiteln werden wir immer wieder mit Ausführungsplänen arbeiten und die offenen Fragen dabei sicherlich beantworten.

4 Werkzeuge und Indikatoren zum Messen der Leistung

Wie bereits in der Einleitung erwähnt, ist Performance-Optimierung eine sehr komplexe Thematik. Damit ein System reibungslos läuft, müssen viele Parameter optimal konfiguriert sein. Falls es einmal nicht so funktioniert wie erwartet, dann gilt es natürlich, die Ursache hierfür zu ermitteln. Dies kann in einem komplexen System wie SQL Server eine echte Herausforderung sein. SQL Server stellt eine Reihe von Werkzeugen zur Verfügung, mit denen das System überwacht werden kann. Diese Werkzeuge werden Sie in diesem Kapitel kennenlernen und in den restlichen Kapiteln des Buches verwenden.

Die Vielzahl der verfügbaren Werkzeuge und somit der Möglichkeiten zur Messung ist dabei oftmals nicht ganz leicht zu überblicken. Ich möchte daher ein passendes Sprichwort an den Anfang dieses Kapitels stellen:

> *»It is possible to own too much. A man with one watch knows what time it is; a man with two watches is never quite sure.«*
>
> *Lee Segal*

Ein wesentliches Ziel dieses Kapitels ist daher vor allem eine Systematisierung der vorhandenen Möglichkeiten. Sie sollen in die Lage versetzt werden, geeignete Methoden zur Messung der relevanten Systemgrößen auszuwählen und bei auftretenden Problemen gezielt nach deren Ursachen zu forschen.

Falls Sie SQL Server bereits ausreichend beherrschen und die hier vorgestellten Werkzeuge kennen, können Sie dieses Kapitel auch getrost »überfliegen«. An dieser Stelle erfolgt noch keine tiefergehende Betrachtung der existierenden Möglichkeiten. Hier werden Ihnen die einzelnen Werkzeuge und Parameter vorgestellt, welche mit den unterschiedlichen Werkzeugen überwacht oder gemessen werden können. Diese Erklärungen bilden die Grundlage für die Kapitel 9, 10 und 11, in denen gezeigt wird, wie diese Werkzeuge zum Auffinden von Problemen eingesetzt werden können.

Den Einsatz dieser Werkzeuge werden wir in der Regel anhand von konkreten Beispielen aufzeigen. Hierfür erzeugen wir in vielen Fällen Testtabellen und füllen diese Tabellen mit zufällig generierten Daten.

Für diesen Zweck ist es sehr nützlich, eine Tabelle anzulegen, die lediglich als Zeilenlieferant dient; eine Idee, die in [1] veröffentlicht wurde. Wir erzeugen diese Tabelle `Numbers` in einer eigenen Datenbank und fügen 1.048.576 (= 220) Zeilen ein:

```
create database QueryTest
go
use QueryTest
go
if (object_id('Numbers', 'U') is not null)
  drop table Numbers
go
set nocount on
create table Numbers(n int not null primary key)
go
-- Füge 2^20 Zeilen ein
insert Numbers(n) values (1)
go
insert Numbers(n) select (select max(n)
                            from Numbers)
                         + row_number() over(order by current_timestamp)
                    from Numbers
go 20
-- Kontrolle
select min(n) as NMin, max(n) as Nmax
  from Numbers
go
```

Die Tabelle `Numbers` hat nur eine einzige Spalte `n`, in der die Zahlen von 1 bis 1.048.576 stehen. Wie gesagt, wir benötigen diese Tabelle in vielen Fällen lediglich zum Generieren von Testdaten.

Falls Sie die Tabelle nicht permanent speichern möchten – etwa weil Sie Speicherplatz sparen wollen –, können Sie auch eine entsprechende Sicht verwenden.

```
create view Numbers as
  select row_number() over(order by current_timestamp) as n
    from sys.all_columns as c1
         cross join sys.all_columns as c2
         cross join sys.all_columns as c3
```

Die in diesem Kapitel vorgestellten Möglichkeiten gestatten generell eine Überwachung eines laufenden Systems und sind nicht nur für das Auffinden von Performance-Problemen geeignet. Wir werden uns jedoch vor allem auf die Performance-Aspekte konzentrieren, um dem Titel des Buches gerecht zu werden.

Dabei geht es nicht so sehr darum, wie man es anstellt, das System von vornherein möglichst so zu konfigurieren, dass möglichst wenige Probleme auftreten. Vielmehr soll gezeigt werden, welche Hilfsmittel Sie einsetzen können, um Performance-Probleme in einem laufenden System zu erkennen.

Hierfür untersuchen wir die folgenden Fragen:

- Was muss eigentlich überwacht werden? Der SQL Server-Dienst wird vom Windows-Betriebssystem gehostet. Er kann also nicht isoliert betrachtet werden, sondern muss immer in diesem Zusammenhang beurteilt werden. Sowohl das Windows- als auch das SQL Server-Betriebssystem stellen eine ungeheure Vielfalt an Parametern bereit, deren Werte Sie messen können, um aus den gemessenen Werten auf den Zustand des Systems zu schließen und Maßnahmen für die Behebung aufgetretener Probleme einzuleiten. Im Verlaufe dieses Buches – und auch bereits in diesem Kapitel – werden Ihnen viele dieser Parameter begegnen. Prinzipiell haben all diese Parameter eines gemeinsam: Sie geben Auskunft über die Verwendung der folgenden Ressourcen:
  - Zeit (CPU-Zeit und Ausführungszeit insgesamt)
  - Hauptspeicher
  - E/A-Vorgänge
- Womit werden die im ersten Punkt genannten Parameter überwacht? Welche Werkzeuge stehen Ihnen also zur Verfügung?

Zunächst einmal ist eine Antwort auf die Frage, wie man Performance-Probleme auffindet, gar nicht so einfach. Wenn Sie die Vielzahl der Möglichkeiten bedenken, die Ihnen für eine Überwachung zur Verfügung stehen, und auch die große Menge der Systemparameter in Betracht ziehen, deren Werte überwacht werden können, ist dies nicht verwunderlich. Oftmals erlebe ich es daher, dass der Ausgangspunkt für eine Performance-Analyse die Beschwerde von Benutzern ist, die feststellen, dass plötzlich »alles so langsam geht«. Eine solche Situation sollten Sie natürlich vermeiden. Besser ist es in jedem Fall, wenn Sie Ihr System so überwachen, dass Sie mögliche Probleme erkennen, bevor Ihre Anwender dies tun. Aber wie gesagt: In der Praxis kommt so etwas häufig vor, und meistens steht nur wenig Zeit für die Behebung der Probleme zur Verfügung.

Wie können auftretende Probleme bereits frühzeitig erkannt werden? Nun, ganz einfach: Durch entsprechende Tests. Ich kann Ihnen hier nur raten, dass Sie sich eine Performance-Grundlage erstellen und dass Sie Ihr System von Zeit zu Zeit darauf hin überprüfen, inwieweit Abweichungen von den gefundenen Basiswerten auftreten. Dieses Benchmarking können Sie durchaus auch auf einem laufenden Produktivsystem zu Zeiten geringer Serverauslastung durchführen. Was Sie hierfür benötigen, ist eine entsprechende Arbeitsauslastung, die im einfachsten Fall aus SQL-Anweisungen besteht, die für Ihre Anwendung typisch sind. Auch der SQL Server Profiler kann Ihnen hierbei helfen. Wie dies funktioniert, erfahren Sie etwas später in diesem Kapitel.

Messen Sie einfach einige wichtige Ressourcen – allen voran die Ausführungszeit – während die Arbeit auf dem Server ausgeführt wird und protokollieren Sie die Ergebnisse. Dadurch erhalten Sie sehr schnell Anhaltspunkte für eventuell vorhandene Probleme.

4.1 Allgemeine Werkzeuge

Wir beginnen mit zwei einfachen und universell einsetzbaren Möglichkeiten zum Messen der Ausführungszeit und der E/A-Vorgänge von Abfragen.

4.1.1 Messen mit der Stoppuhr

Natürlich können Sie eine Stoppuhr benutzen, während eine Abfrage ausgeführt wird. Wenn Sie die Abfrage aus dem Abfrageeditor des Management Studios ausführen, dann geht es auch etwas bequemer: einfach durch das Ablesen der Ausführungszeit in der rechten unteren Ecke des Fensters.

| QueryTest | 00:07:36 | 12345 Zeilen |

Möglich ist auch die Messung mittels T-SQL. Hierfür stehen Ihnen integrierte Funktionen zur Abfrage und Konvertierung von Datum und Uhrzeit zur Verfügung. Sie die Uhrzeit am Beginn und Ende eines Skripts festhalten und die Differenz ausgeben oder protokollieren. Dies funktioniert zum Beispiel so:

```
declare @start datetime
       ,@dauer bigint
set @start = current_timestamp

-- SQL Anweisungen...

set @dauer = datediff(s, @start, current_timestamp)
select @dauer as Sekunden
      ,@dauer / 60000.0 as Stunden
```

Anstatt das Ergebnis auszugeben, können Sie es auch in einer eigens hierfür angelegten Protokolltabelle speichern:

```
insert Protokoll(Zeitpunkt,Kommentar,Dauer)
  values (current_timestamp, 'Datenimport erfolgreich', @dauer)
```

Diese Möglichkeit ist nützlich für die Kontrolle der weiter oben erwähnten Performance. So können Sie zum Beispiel Ihre T-SQL-Arbeitsauslastung mit entsprechenden INSERT-Anweisungen nach dem obigen Muster anreichern, und für besonders wichtige Punkte die Ausführungszeiten zu protokollieren.

Bedenken Sie hierbei, dass auf die oben beschriebene Art nur bis zu einer Genauigkeit von 3,33 ms gemessen werden kann – mehr geben die Systemfunktionen nicht her. Für die meisten Anwendungsfälle dürfte diese Genauigkeit aber vollkommen ausreichend sein.

4.1.2 Statistische Größen

Die folgenden beiden Optionen erlauben die Protokollierung von statistischen Werten zu Ausführungs- und Übersetzungszeiten sowie zu E/A-Vorgängen:

SET STATISTICS IO ON

Ist diese Option eingeschaltet, so erhalten Sie nach der Ausführung einer Abfrage in einem T-SQL-Stapel Informationen über die erforderlichen E/A-Operationen. Diese Informationen werden im Meldungsbereich des Abfragefensters ausgegeben und sehen zum Beispiel so aus:

```
'syscolrdb'-Tabelle. Scananzahl 1, logische Lesevorgänge 184,
physische Lesevorgänge 4, Read-Ahead-Lesevorgänge 198,
logische LOB-Lesevorgänge 0, physische LOB-Lesevorgänge 0,
Read-Ahead-LOB-Lesevorgänge 0.
'syscolpars'-Tabelle. Scananzahl 1, logische Lesevorgänge 13,
physische Lesevorgänge 3, Read-Ahead-Lesevorgänge 24,
logische LOB-Lesevorgänge 0, physische LOB-Lesevorgänge 0,
Read-Ahead-LOB-Lesevorgänge 0.
```

Die Meldung enthält Informationen darüber, wie oft auf eine Tabelle in der Abfrage zugegriffen wurde (Scananzahl), wie viele (8 kByte große) Datenseiten aus dem Datencache gelesen (logische Lesevorgänge) und wie viele Blöcke von der Festplatte gelesen wurden (physische Lesevorgänge und Read-Ahead-Lesevorgänge). Falls die Abfrage LOB-Spalten verarbeitet, also zum Beispiel Text- oder Image-Spalten zurückgibt, so finden Sie die entsprechenden Statistiken in den entsprechenden LOB-Informationen.

`STATISTICS IO` bietet eine sehr einfache und dennoch extrem nützliche Möglichkeit zur Bestimmung von benötigten E/A-Operationen. Wie bereits in den beiden vorangegangenen Kapiteln erwähnt, ist die Minimierung von E/A-Vorgängen ein wichtiger Ansatzpunkt bei der Optimierung. `STATISTICS IO` ist hierbei sehr hilfreich.

Durch die folgende Anweisung schalten Sie die Protokollierung wieder aus:

```
set statistics io off
```

SET STATISTICS TIME ON

Diese Option bewirkt, dass zusätzliche Informationen im Meldungsbereich des Abfragefensters ausgegeben werden. Die Informationen geben Auskunft über die benötigten Abfrage- und Übersetzungszeiten. Eine Ausgabe sieht zum Beispiel so aus:

```
SQL Server-Analyse- und Kompilierzeit:
, CPU-Zeit = 16 ms, verstrichene Zeit = 173 ms.

SQL Server-Ausführungszeiten:
, CPU-Zeit = 15 ms, verstrichene Zeit = 534 ms.
```

Der erste Bereich bezieht sich auf die Erstellung des Abfrageplans. Für die Erstellung des Plans wurden insgesamt 173 ms benötigt, wobei die CPU 16 ms benötigt hat. Die zweite

Ausgabe enthält die Information über die Ausführung der Abfrage selber. Im Beispiel hat die Ausführung insgesamt 534 ms gedauert, die CPU wurde 15 ms lang beansprucht.

Wundern Sie sich bitte nicht, wenn Sie CPU-Zeiten sehen, die größer sind als die Ausführungszeit. In diesem Fall wurde die Abfrage parallel, also auf mehreren CPUs gleichzeitig ausgeführt. Die CPU-Zeiten der beteiligten Prozessoren werden einfach addiert, sodass dieser Fall durchaus eintreten kann.

4.2 Der Aktivitätsmonitor

Bereits seit der Version 6.5, die etwa im Jahr 1995 erschienen ist, gibt es den SQL Server Aktivitätsmonitor, der – wie der Name bereits besagt – eine Überwachung der aktuellen SQL Server-Aktivität ermöglicht. Der Aktivitätsmonitor kann für jede aktuell bestehende Verbindung zum SQL Server Informationen zur gerade ausgeführten Abfrage, aber beispielsweise auch zu existierenden Blockierungen und Ursachen für Wartezustände anzeigen. Dadurch ist der Aktivitätsmonitor eine große Hilfe, wenn es gilt, den aktuellen Zustand von SQL Server zu kontrollieren.

Natürlich gibt es den Aktivitätsmonitor auch im SQL Server Management Studio 2008. Allerdings wurde sein Erscheinungsbild stark geändert. Der Aktivitätsmonitor wird nun nicht mehr über einen Eintrag unter dem Knoten VERWALTUNG des Objekt-Explorers gestartet. Ich musste etwas suchen um herauszufinden, wo sich der Aktivitätsmonitor in der neuen Version versteckt. Der Start erfolgt nun aus dem Kontextmenü der SQL Server-Instanz oder über die Menüleiste, sofern Sie die entsprechende Schaltfläche zur Menüleiste hinzugefügt haben (siehe Abbildung 4.1).

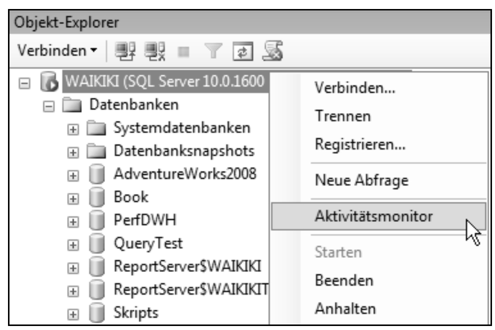


Abbildung 4.1:
Starten des Aktivitätsmonitors

Wenn Sie SQL Server aus einer früheren Version her kennen, dann werden Sie einigermaßen überrascht sein. Der Aktivitätsmonitor hat ein völlig anderes Aussehen, das stark an den Ressourcenmonitor in Windows erinnert (siehe Abbildung 4.2).

Die angezeigten Informationen gliedern sich in vier Bereiche:

- Übersicht
- Ressourcenwartevorgänge
- Datendatei E/A
- Aktuell wertvolle Abfragen

4.2.1 Übersicht

Der erste Bereich enthält grafische Übersichten zur Prozessorauslastung durch den SQL Server-Dienst, zu wartenden Tasks, zum E/A-Durchsatz sowie zu T-SQL-Stapelanforderungen je Sekunde. Dieser Bereich ist standardmäßig eingeblendet, so wie in Abbildung 4.2 zu sehen.

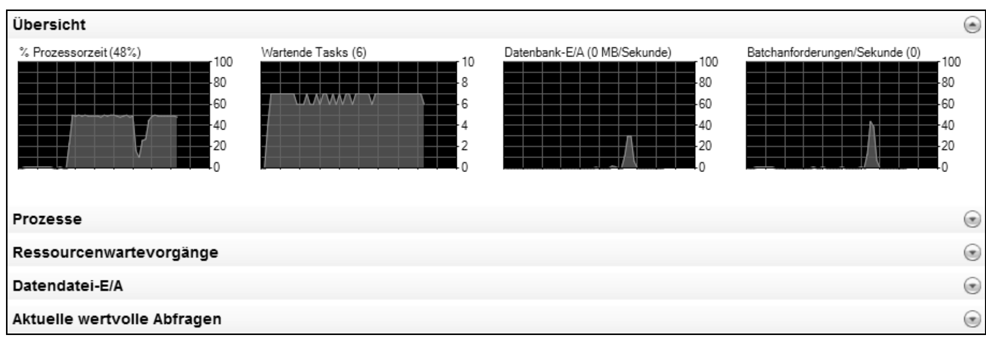


Abbildung 4.2: *Der Aktivitätsmonitor in Aktion*

4.2.2 Aktuelle Prozesse

Dieser Bereich zeigt Informationen zu den aktuellen Benutzerverbindungen an. Eine nützliche Erweiterung ist die Möglichkeit, nach den einzelnen Spalten zu sortieren und zu filtern. Sehr willkommen ist sicherlich auch, dass nun für einen Prozess sofort aus dem Kontextmenü eine Standard-Ablaufverfolgung mit dem SQL Server Profiler geöffnet werden kann (siehe Abbildung 4.3). Wenn Sie an dieser Stelle mit dem Profiler noch nichts anfangen können, haben Sie bitte ein wenig Geduld; in Abschnitt 4.3 behandeln wir ihn ausführlich.

Prozesse

Sit	Be	Datenb	Tasksta	Befehl	Anwendung	Wartezeit (ms)	Wartetyp	Warte	Bl vo	Kc
51	1				SQLAgent - G...	0				1
52	1				SQLAgent - Jo...	0				1
53	1				Microsoft SQL...	0				1
54	1	master	RUNNAB...	SELECT	Microsoft SQL...	0				
55	1	tempdb	RUNNING	SELECT	Microsoft SQL...	0				
56	1				Microsoft SQL...	0				1
57	1	Adventure...	RUNNING	SELECT	Microsoft SQL...					

Details
Prozess abbrechen
Ablaufverfolgungsprozess in SQL Server Profiler

Abbildung 4.3: *Anzeige der aktuell ausgeführten Prozesse und Starten des Profilers*

4.2.3 Ressourcenwartevorgänge

In diesem Fenster finden Sie eine Momentaufnahme der Wartezeiten auf SQL Server-Ressourcen, wie zum Beispiel CPU, Puffer (Caches) oder auch Netzwerk (Abbildung 4.4). Die hier angegebenen Wartezeiten geben die Dauer an, die ein Prozess (oder genauer: ein Worker Thread) warten musste, um Zugriff auf die entsprechende Ressource zu erhalten. Diese Wartezeiten können als Indikatoren für eventuelle Ressourcen-Engpässe verwendet werden.

Ressourcenwartevorgänge

Wartekategorie	Wartezeit (ms/Sek.)	Aktuelle Wartezeit (ms/Sekunde)	Durchschnittliche Zahl an Wartevorgängen	Kumulierte Wartezeit (Sekunde)
Network I/O	0	1	0,0	357
Buffer Latch	0	0	0,0	83
Compilation	0	0	0,0	0
CPU	0	0	0,0	5774
Latch	0	0	0,0	45
Lock	0	0	0,0	710
Logging	0	0	0,0	1765
Memory	0	0	0,0	4

Abbildung 4.4: *Ressourcenwartevorgänge*

Sehr nützlich ist auch die Angabe der insgesamt aufgetretenen Wartezeit für jede Ressource (in Abbildung 4.4 in der letzten Spalte dargestellt).

4.2.4 Datendatei E/A

Dieses Fenster enthält Informationen über die momentanen E/A-Vorgänge je Datenbank. Präsentiert wird hier lediglich eine Momentaufnahme – und nicht etwa der gesamte E/A-Durchsatz seit dem letzten Start von SQL Server. Abbildung 4.5 zeigt ein Beispiel hierzu.

Datendatei-E/A

Datenbank	Dateiname	MB/Sekunde gelesen	MB/Sekunde geschrieben	Antwortzeit (ms)
QueryTest	C:\Program Files\Microsoft SQL Server\MSSQ...	2,0	3,6	45
AdventureWorks2008	C:\Program Files\Microsoft SQL Server\MSSQ...	0,0	0,0	0
AdventureWorks2008	C:\Program Files\Microsoft SQL Server\MSSQ...	1,3	0,0	141
Book	C:\Program Files\Microsoft SQL Server\MSSQ...	0,0	0,0	0
Book	C:\Program Files\Microsoft SQL Server\MSSQ...	0,0	0,0	0
master	C:\Program Files\Microsoft SQL Server\MSSQ...	0,1	0,0	55

Abbildung 4.5: *Momentane E/A-Operationen je Datenbank*

4.2.5 Aktuell wertvolle Abfragen

Hier handelt es sich zunächst einmal um eine Fehlübersetzung (oder zumindest um eine recht unglückliche Übersetzung). In der englischen Version lautet der Fenstertitel »Recent Expensive Queries«. Angezeigt werden also die teuersten Abfragen der letzten Zeit. Inwieweit diese Abfragen wirklich wertvoll sind, kann der Aktivitätsmonitor wohl nicht beurteilen. Abbildung 4.6 zeigt ein Beispiel.

Aktuelle wertvolle Abfragen

Abfrage	Ausführungen	CPU (ms/Sekunde	Physische Lesevorgäng	Logische Schreibvorgä	Logische Lesevorg
select checksum_agg(checksum(*)) from sys.all...	0	925	0	0	
SELECT [Session ID] = s.session_id, [Us...	7	2	0	0	
WITH merged_query_stats AS (SELECT ...	3	1			
DELETE FROM #am_resource_mon_snap WH...	15	0			
SELECT @current_total_io_mb = SUM(num_of...	7	0	0	0	

Abfragetext bearbeiten
Ausführungsplan anzeigen

***Abbildung 4.6:** Die kostspieligsten Abfragen der letzten Zeit*

In der Abbildung ist auch zu sehen, wie Sie für eine solche Abfrage, die ja möglicherweise problematisch ist, den Abfrageplan im grafischen Format aus dem Kontextmenü heraus öffnen können. Dies ist wirklich sehr praktisch. Dadurch haben Sie die Möglichkeit, eventuell kritische Abfragen sofort näher zu untersuchen.

Der Aktivitätsmonitor ist sehr häufig ein geeigneter Einstieg in die Untersuchung von Performance-Problemen. Mit nur einigen wenigen Mausklicks erhalten Sie schnell einen guten Überblick über den momentanen Zustand Ihres Systems und können dann weiter in die Tiefe gehen. Für diese tiefergehenden Analysen bietet der Aktivitätsmonitor in einigen Fällen sogar selber Hilfsmittel an, wie beispielsweise die gerade erwähnte Möglichkeit, Abfragepläne anzuzeigen (und auch zu speichern) oder Ablaufverfolgungen für laufende Prozesse zu starten.

4.3 Ablaufverfolgungen und der SQL Server Profiler

Der SQL Server Profiler ist ein sehr mächtiges Werkzeug zur Überwachung von SQL Server-Aktivitäten. In diesem Kapitel erhalten Sie zunächst eine Einführung in die Bedienung und die Möglichkeiten des Profilers. In Kapitel 10 werden Sie sehen, wie der Profiler zum Aufspüren von Problemen eingesetzt werden kann.

Der Profiler erstellt Ablaufverfolgungen (engl.: traces) von SQL Server-Aktivitäten. Welche Aktivitäten in der Ablaufverfolgung protokolliert werden, bestimmen Sie durch die Auswahl bestimmter Ereignisse. Hierbei können Sie aus einer großen Vielzahl von vordefinierten Ereignissen diejenigen auswählen, die für Ihre Zwecke erforderlich sind.

Der Profiler kann aber noch mehr:

- **Fehlersuche**. Die grafische Benutzeroberfläche des Profilers ist bei der Suche nach Abfragefehlern sehr hilfreich. Es ist möglich, die protokollierten Ereignisse so festzulegen, dass alle an den Server gesendeten Abfragen in die Ablaufverfolgung eingeschlossen werden. Dadurch können Sie beim Auftreten eines Fehlers nicht nur nachverfolgen, von wem und wann eine entsprechende Abfrage gesendet wurde. Sie sehen auch den Text der Abfrage und können so leicht fehlerhafte Abfragen entdecken.
- **Lernen**. Der Profiler kann nützlich sein, wenn Sie Interna des SQL Servers und der Verwaltungswerkzeuge »ausspionieren« möchten. So können Sie zum Beispiel mit eingeschalteter Ablaufverfolgung bestimmte Dialoge im Management Studio öffnen und

dann in der Ablaufverfolgung sehen, welche Abfragen an den SQL Server gesendet wurden, um die Informationen im Dialogfenster darzustellen. Auch die Erstellung von im Management Studio integrierten Berichten lassen sich auf diese Weise nachverfolgen. Hierdurch lernen Sie beispielsweise die dynamischen Verwaltungssichten von SQL Server kennen.

- **Optimieren**. Die in der Ablaufverfolgung enthaltenen Informationen können sehr wirkungsvoll für eine Analyse der Abfrageleistung oder des Serververhaltens herangezogen werden. So ist es zum Beispiel möglich, die Ausführungsdauer, die Anzahl der Lesevorgänge oder auch direkt Abfragepläne in die Ablaufverfolgung mit aufzunehmen. All diese Indikatoren liefern Hinweise auf die Abfrageleistung.
- **Wiedereinspielen einer Ablaufverfolgung**. Eine einmal aufgezeichnete Ablaufverfolgung kann in unterschiedlichen Formaten gespeichert werden. Ist die Ablaufverfolgung einmal gespeichert, so kann sie erneut abgespielt werden. Hierzu müssen allerdings einige Voraussetzungen erfüllt sein. Zunächst einmal muss die Ablaufverfolgung ganz bestimmte Ereignisse enthalten, damit die enthaltenen Anweisungen erneut ausgeführt werden können. Welche Ereignisse dies sind, erfahren Sie etwas weiter unten in diesem Abschnitt. Darüber hinaus müssen auch die Datenbanken, auf die zugegriffen wird, die wiederholte Ausführung der in der Ablaufverfolgung enthaltenen Anweisungen »verkraften«. Wenn Sie beispielsweise Zeilen einfügen und dabei Primärschlüssel vergeben, so funktioniert dies natürlich nicht mehrfach. Hier können Sie sich aber damit behelfen, dass Sie vor der Erstellung der Ablaufverfolgung eine Datensicherung vornehmen, die Sie dann vor jedem erneuten Einspielen der Ablaufverfolgung wiederherstellen. Eine weitere Voraussetzung ist, dass die *database_id* aller Datenbanken, auf die bei der Ablaufverfolgung zugegriffen wird, beim Wiedereinspielen gleich sein muss. Dies ist in der Regel dann problematisch, falls Sie Datenbanken trennen und wieder anfügen. So können Sie beispielsweise bestimmte Aktivitäten, die für Ihre Anwendung relevant oder problematisch sind in einer Ablaufverfolgung protokollieren und dann mit der Konfiguration oder der Hardware experimentieren, wobei Sie nach bestimmten Änderungen die Ablaufverfolgung erneut einspielen.

Bevor Sie nun daran gehen zu untersuchen, wie der Profiler bedient wird, möchte ich allerdings noch eine Warnung loswerden: Es gibt ein aus der Physik bekanntes Prinzip, welches besagt, dass jede Messung auch Auswirkungen auf das Messergebnis selbst hat. Mit anderen Worten: Jede Messung verfälscht auch das Ergebnis. Dieses Prinzip gilt auch für den Profiler. Wenn Sie den Profiler für das Messen der Abfrageleistung verwenden, so müssen Sie darauf achten, dass nicht etwa der Profiler selber zum Problem wird, weil letztlich er es ist, der für eine schlechte Abfrageleistung verantwortlich zeichnet. Der Profiler kann das Laufzeitverhalten stark verschlechtern, wenn Sie zu viele Ereignisse in die Ablaufverfolgung einschließen.

Bitte beachten Sie daher die folgenden Hinweise für das Erstellen einer Ablaufverfolgung:

- Falls möglich, verzichten Sie auf die grafische Oberfläche des Profilers. Diese Art der Repräsentation erzeugt unnötige Netzwerk- und SQL Server-Last. Verwenden Sie statt der grafischen Darstellung besser eine serverseitige Ablaufverfolgung. Wie Sie hierbei vorgehen, erfahren Sie gleich.
- Verwenden Sie nur die Ereignisse, die Sie wirklich benötigen, gestalten Sie die Ablaufverfolgung also möglichst »leichtgewichtig«.
- Speichern Sie die Ablaufverfolgung in einer Datei. Der Profiler bietet auch die Möglichkeit, Ablaufverfolgungen in einer eigenen Tabelle in einer Datenbank Ihrer Wahl abzulegen. Die dadurch erforderlichen zusätzlichen `INSERT`-Operationen sorgen ebenfalls für eine zusätzliche Belastung Ihres Servers. Die Speicherung in einer Ablaufverfolgungsdatei ist erheblich günstiger. Diese Datei kann später in einem separaten Schritt sehr einfach in eine Tabelle übertragen werden. Auch hierzu gibt es etwas weiter unten ein Beispiel.

4.3.1 Erstellen einer einfachen Ablaufverfolgung

Wir wollen zunächst eine einfache Ablaufverfolgung konfigurieren, welche die grafische Oberfläche des Profilers verwendet – auch wenn ich Ihnen gerade geraten habe, dies nach Möglichkeit zu vermeiden. Zum Einstieg geht es jedoch erst einmal darum, wie der Profiler überhaupt verwendet wird, und wie man eine Ablaufverfolgung konfiguriert.

Der Profiler wird entweder über das Startmenü MICROSOFT SQL SERVER 2008 • LEISTUNGSTOOLS • SQL SERVER PROFILER oder alternativ aus dem Management Studio heraus über EXTRAS • SQL SERVER PROFILER gestartet.

Es erscheint zunächst ein leeres Fenster. Sie haben nun die Möglichkeit, eine neue Ablaufverfolgung zu beginnen. Hierzu können Sie zum Beispiel aus der Menüleiste die entsprechende Schaltfläche betätigen (Abbildung 4.7).

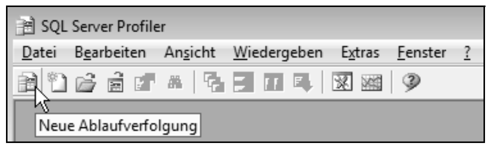


Abbildung 4.7:
Starten einer neuen Ablaufverfolgung

Nachdem Sie sich am Server angemeldet haben, auf dem die Ablaufverfolgung ausgeführt werden soll, öffnet sich ein Dialog, in dem Sie die Ereignisse auswählen, die Sie in Ihre Ablaufverfolgung einschließen möchten. In Abbildung 4.8 sehen Sie diesen Dialog.

Auf der Seite ALLGEMEIN legen Sie fest, ob die Ablaufverfolgungsdaten gespeichert werden sollen. Hierbei können Sie wählen, ob die Speicherung in einer Datei oder in einer Tabelle erfolgen soll. Für das Speicherformat *Datei* können Sie eine maximale Dateigröße und ein *Dateirollover* (ein weiteres Beispiel für eine grauenhafte Übersetzung) festlegen.

Falls Sie dies tun, wird nach dem Erreichen der konfigurierten Maximalgröße eine neue Datei für das Protokoll begonnen. Die einzelnen Dateien werden dabei fortlaufend nummeriert. Falls Sie eine maximale Größe für die Datei angeben und *Dateirollover* dagegen nicht aktivieren, so wird die Ablaufverfolgung beim Erreichen der maximalen Dateigröße automatisch beendet.

Abbildung 4.8: *Konfiguration einer Ablaufverfolgung*

Auch bei Speicherung der Ablaufverfolgung in einer Tabelle können Sie eine maximale Größe konfigurieren, wobei Sie hier die maximal zulässige Zeilenanzahl angeben. Sobald diese Zeilenanzahl erreicht ist, wird die Protokollierung in der Tabelle – nicht aber die Ablaufverfolgung selber – beendet. Sie sehen die Protokolldaten dann zwar noch in der grafischen Oberfläche des Profilers, eine Speicherung, etwa in der Art, dass ein »Rollover« durch das Ersetzen der ältesten Tabellenzeilen durch die neu hinzukommenden durchgeführt wird, erfolgt jedoch nicht.

Falls Sie keine Speicherung der Ablaufverfolgung konfigurieren, so wird das Ergebnis der Ablaufverfolgung später lediglich in der grafischen Oberfläche des Profilers angezeigt. Wie der Profiler eine solche Ablaufverfolgung darstellt, erfahren Sie etwas weiter unten. Sie haben dann aber immer noch die Möglichkeit, die Ablaufverfolgung über das Menü DATEI in einer Datei oder Tabelle zu speichern.

Das Aktivieren der Option ABLAUFVERFOLGUNGSDATEN VON SERVERPROZESSEN bewirkt, dass tatsächlich alle konfigurierten Ereignisse protokolliert werden, und zwar auch dann, wenn der Server, der das Protokoll erstellt, bereits unter voller Last läuft. Andernfalls kann es durchaus vorkommen, dass einige Ereignisse nicht in das Protokoll aufgenommen werden, da die Belastung des Servers bereits zu hoch ist und eine Ablaufverfolgung diese Belastung noch weiter erhöhen würde.

Im unteren Bereich haben Sie noch die Möglichkeit, eine automatische Beendigung der Ablaufverfolgung zu einer bestimmten Zeit festzulegen. So können Sie zum Beispiel die Ablaufverfolgung unbeaufsichtigt laufen lassen und etwa in der Nacht anhalten, da Sie wissen, dass ab diesem Zeitpunkt keine Überwachung mehr erforderlich ist. Etwas weiter unten werden Sie noch sehen, wie der SQL Server Agent verwendet werden kann, um eine Ablaufverfolgung automatisch zu starten und zu beenden.

Wenn Sie die Vorlage *Leer* auswählen (so wie in der Abbildung zu sehen) und auf die Seite EREIGNISAUSWAHL wechseln, stellen Sie fest, dass es eine Vielzahl von Ereignissen gibt, aus denen Sie auswählen können. Je nachdem, wofür Sie Ihre Ablaufverfolgung erstellen möchten, können Sie einen passenden Satz von Ereignissen und Ereignisspalten – dies sind die Daten, welche beim Auftreten des Ereignisses protokolliert werden – konfigurieren. Auch hier gilt der Grundsatz: Nur so viel wie nötig und so wenig wie möglich.

Glücklicherweise müssen Sie nicht alle möglichen Ereignisklassen beherrschen und detailliert wissen, wofür diese Klassen im Einzelnen verwendet werden sollten. Der Profiler stellt eine Reihe von Vorlagen zur Verfügung, die bereits vordefinierte Sätze von Ereignissen und Ereignisspalten enthalten. Die Anzahl der Vorlagen ist erheblich kleiner als die Anzahl der Ereignisse. Daher ist die Konfiguration einer Ablaufverfolgung durch die Verwendung einer Vorlage erheblich einfacher zu bewerkstelligen. Ich empfehle Ihnen, stets mit einer Vorlage zu beginnen und dann die durch die Verwendung der Vorlage in die Ablaufverfolgung aufgenommenen Ereignisse zu ergänzen bzw. einzuschränken. In Tabelle 4.1 finden Sie eine Übersicht über die standardmäßig installierten Vorlagen.

Vorlage	Verwendungszweck
SP_Counts	Protokolliert den Start gespeicherter Prozeduren.
Standard	Protokolliert An- und Abmeldungen sowie die Ausführung von T-SQL-Stapeln und die Ausführung gespeicherter Prozeduren.
TSQL	Protokolliert den Start von T-SQL-Stapeln und gespeicherten Prozeduren, wobei nur ein sehr eingeschränkter Satz von Ereignisspalten in das Protokoll eingeschlossen wird. Sehr leichtgewichtig.
TSQL_ Duration	Protokolliert das Ende von T-SQL-Stapeln und Ausführungen von gespeicherten Prozeduren. Auch hier werden nur sehr wenige Ereignisspalten eingeschlossen. Insbesondere wird die Ausführungsdauer in die Ablaufverfolgung eingeschlossen.
TSQL_ Grouped	Protokolliert den Start von Prozeduraufrufen und von T-SQL-Stapeln. In die Ereignisspalten werden auch Benutzerinformationen aufgenommen, sodass eine Zuordnung der ausgeführten Anweisungen zum angemeldeten bzw. die Anweisung ausführenden Benutzer möglich ist.
TSQL_Locks	Protokolliert Informationen zu Sperren und Blockierungen. Hier sind zum Beispiel Ereignisse zu Deadlocks und Sperrenausweitung (engl.: Lock Escalation) enthalten. Auch die Ausführung gespeicherter Prozeduren und T-SQL-Stapeln wird jeweils mit Start und Ende in die Ablaufverfolgung eingeschlossen.

Tabelle 4.1: *Vorlagen für die Konfiguration von Ablaufverfolgungs-Ereignissen*

Vorlage	Verwendungszweck
TSQL_Replay	Diese Vorlage enthält alle erforderlichen Ereignisse, die für eine Wiedergabe, also ein erneutes Einspielen der Ablaufverfolgung erforderlich sind. Diese sind eine ganze Menge Ereignisse, so dass eine solche Ablaufverfolgung einiges an Ressourcen erfordern kann.
TSQL_SPs	Protokolliert Informationen zur Ausführung von gespeicherten Prozeduren sowie die innerhalb einer Prozedur ausgeführten Anweisungen. Eine solche Ablaufverfolgung kann sehr umfangreich werden. Beispielsweise dann, wenn Ihre Anwendungen größtenteils mit gespeicherten Prozeduren arbeiten, die viele Anweisungen enthalten.
Tuning	Diese Vorlage konfiguriert alle erforderlichen Ereignisse, die zur Erstellung einer Arbeitsauslastung für das Datenbankmodul *Optimierungsratgeber* (siehe Kapitel 11) erforderlich sind. Dies sind im Wesentlichen alle T-SQL-Anweisungen mit ihrer Ausführungsdauer. Auch eine solche Ablaufverfolgung kann sehr umfangreich werden.

Tabelle 4.1: *Vorlagen für die Konfiguration von Ablaufverfolgungs-Ereignissen (Forts.)*

Nachdem Sie sich für die Verwendung einer bestimmten Vorlage entschieden haben, können Sie auf der Seite EREIGNISAUSWAHL die entsprechenden Ereignisse erweitern oder einschränken. Abbildung 4.9 zeigt ein Beispiel für die Vorlage *Tuning*. Für jedes Ereignis haben Sie die Möglichkeit, die Daten, die beim Auftreten des Ereignisses protokolliert werden sollen, auszuwählen. Dies sind die sogenannten Ereignisspalten. Falls Sie eine Vorlage verwenden, sehen Sie zunächst nur diejenigen Ereignisse und Ereignisspalten, die in der Vorlage enthalten sind. Über die beiden Optionen ALLE EREIGNISSE ANZEIGEN und ALLE SPALTEN ANZEIGEN können Sie sich den vollständigen Satz Spalten und Zeilen anzeigen lassen.

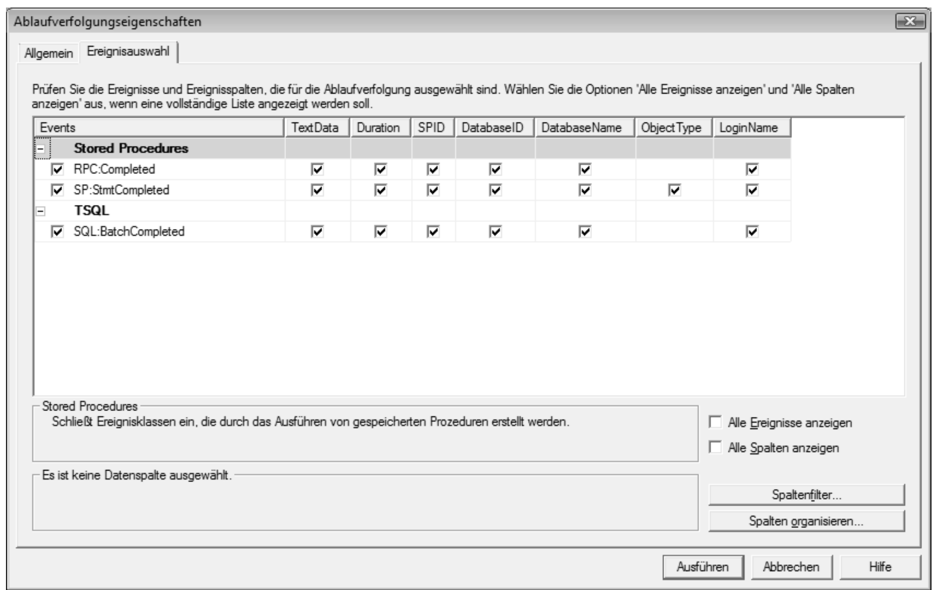


Abbildung 4.9: *Ereignisse der Vorlage »Tuning«*

Sobald Sie auf die Schaltfläche AUSFÜHREN klicken, wird die Ablaufverfolgung gestartet und Sie können die auftretenden Ereignisse online am Bildschirm verfolgen (Abbildung 4.10).

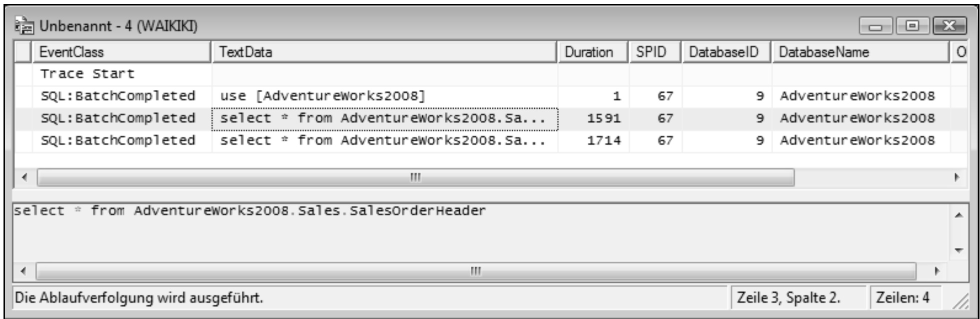


Abbildung 4.10: *Beispiel für eine Ablaufverfolgung*

4.3.2 Ereignisse und Ereignisspalten

Die Vielzahl vorhandener Ereignisse macht die Auswahl der »richtigen« Ereignisse natürlich nicht gerade einfach. Wenn Sie sich hier zu Beginn etwas hilflos fühlen, weil Sie nicht wissen, welche Ereignisse Sie in Ihre Ablaufverfolgung aufnehmen sollen, dann stehen Sie mit diesem Problem nicht alleine da. So ziemlich jeder, der mit dem Profiler beginnt, hat am Anfang dieses Problem. Widerstehen Sie jedoch bitte der Versuchung, alle möglichen Ereignisse in die Ablaufverfolgung aufzunehmen, nach dem Motto: »Lieber zu viel Information, als zu wenig.« Wenn Sie dies tun, so werden Sie schnell merken, dass Ihre Abfrageleistung dramatisch sinkt. Auch können Sie die Fülle an Informationen, die im Protokoll enthalten sind, gar nicht mehr sinnvoll auswerten.

Welche Ereignisse sind denn aber die »richtigen«? Nun, die Antwort auf diese Frage ist gleichermaßen einfach wie auch unbefriedigend: Das hängt davon ab, welchen Problemen Sie auf den Grund gehen möchten. Es gibt kein allgemeines Rezept. Sie werden nicht umhinkommen, Ihre eigenen Erfahrungen zu sammeln. Die folgende Vorgehensweise kann ich Ihnen allerdings empfehlen:

- Beginnen Sie mit einer der vorhandenen Vorlagen, wobei Sie Tabelle 4.1 als Hilfe für die Auswahl verwenden können.
- Starten Sie die Ablaufverfolgung und beobachten Sie, welche Ereignisse in das Protokoll aufgenommen werden.
- Entfernen Sie nicht benötigte Ereignisse und fügen Sie fehlende Ereignisse hinzu.

Für eine leichtere Orientierung sind die Ereignisse in Gruppen organisiert. In Abbildung 4.9 sehen Sie zum Beispiel die Gruppen *Stored Procedures* und *TSQL*. Darüber hinaus erhalten Sie im unteren Bereich der Seite EREIGNISAUSWAHL für jedes Ereignis und für jede Ereignisspalte zusätzliche Informationen, sobald Sie die Maus über ein Ereignis oder eine Spalte bewegen.

Eine komplette Aufzählung aller vorhandenen Ereignisse mit einer entsprechenden Dokumentation finden Sie in der Online-Dokumentation. An dieser Stelle möchte ich nur einige wichtige Ereignisse nennen, die Sie häufig benötigen (Tabelle 4.2).

Gruppe	Ereignis	Erklärung
Errors and Warnings	*User Error Message*	Das Protokoll enthält Informationen über aufgetretene Fehler.
Locks	*Deadlock graph*	Bei Auftreten von Deadlocks wird eine grafische Darstellung der beteiligten Ressourcen und des Deadlock-Opfers geliefert.
Performance	*Showplan All*	Übernimmt den tatsächlichen Ausführungsplan in das Protokoll. Dieser Plan wird standardmäßig in der grafischen Form angezeigt.
Scans	*Scan started*	Eine sequenzielle Suche in einer Tabelle oder einem Index wurde gestartet.
	Scan stopped	Eine sequenzielle Suche in einer Tabelle oder einem Index wurde abgeschlossen.
Stored Procedures	*SP:Starting*	Ein Prozeduraufruf beginnt.
	SP:Completed	Ein Prozeduraufruf ist beendet.
	SP:StmtStarting	Eine Anweisung innerhalb einer gespeicherten Prozedur beginnt.
	SP:StmtCompleted	Eine Anweisung innerhalb einer gespeicherten Prozedur ist beendet.
	RPC:Completed	Der Aufruf einer gespeicherten Prozedur vom Client ist abgeschlossen.
TSQL	*SQL:BatchStarting*	Die Ausführung eines T-SQL-Stapels beginnt.
	SQL:BatchCompleted	Die Ausführung eines T-SQL-Stapels ist beendet.
	SQL:StmtStarting	Eine Anweisung innerhalb eines T-SQL-Stapels startet.
	SQL:StmtCompleted	Eine Anweisung innerhalb eines T-SQL-Stapels ist abgearbeitet.

Tabelle 4.2: *Ausgewählte Profiler-Ereignisse*

Welche Daten bei Auftreten eines der in Tabelle 4.2 genannten Ereignisse in das Protokoll aufgenommen werden, bestimmen Sie durch die Auswahl der Ereignisspalten. Die Anzahl der Spalten ist ähnlich groß wie die Anzahl der verfügbaren Ereignisse. Daher möchte ich auch hier wiederum nur die wichtigsten aufführen:

- **TextData**. Hier finden Sie weiterführende und wichtige Informationen zum aufgetretenen Ereignis. Der Inhalt dieser Spalte ist vom Ereignis abhängig. So enthält diese Spalte zum Beispiel für das Ereignis *User Error Message* den Text der Fehlermeldung und für das Ereignis *SQL:StmtStarting* den Text der SQL-Anweisung. Da der in dieser Spalte enthaltene Text sehr lang werden kann, wird er im Ablaufverfolgungsfenster nicht nur in der Spalte selber, sondern nochmals im unteren Bereich des Fensters angezeigt. In Abbildung 4.10 sehen Sie den Text der Abfrage.
- **StartTime**. Enthält den Start-Zeitpunkt eines Ereignisses.
- **EndTime**. Enthält das Ende-Zeitpunkt eines Ereignisses.
- **Duration**. Enthält die Dauer eines Ereignisses. In einer gespeicherten Ablaufverfolgung wird dieser Wert in der Einheit µs (Mikrosekunden) gespeichert. Die Anzeige in der grafischen Oberfläche des Profilers erfolgt standardmäßig in ms (Millisekunden). Über das Menü EXTRAS • OPTIONEN können Sie konfigurieren, dass die Anzeige ebenfalls in µs erfolgt.
- **SPID**. Enthält die *ProzessId* der Clientverbindung. Diese Ereignisspalte muss in die meisten Ereignisse aufgenommen werden.
- **Reads**. Gibt die Anzahl logischer Lesevorgänge an, die das Ereignis ausgelöst hat.
- **Writes**. Gibt die Anzahl von Schreibvorgängen an, die das Ereignis erfordert hat.
- **CPU**. Gibt die CPU-Zeit (in ms) an, die das Ereignis benötigt hat.
- **DatabaseName**. Enthält den Namen der Kontextdatenbank der Verbindung. Dies ist die Datenbank, die zum Beispiel beim Verbindungsaufbau als *InitialCatalog* angegeben oder über USE eingestellt wurde und nicht etwa die Datenbank, von der Daten abgefragt werden. Das folgende Skript beispielsweise protokolliert die Datenbank *master* und nicht etwa *AdventureWorks2008* als DatabaseName:

  ```
  use master;
  select * from AdventureWorks2008.Sales.SalesOrderHeader
  ```

- **NTUserName**. Enthält den Windows-Benutzernamen des die Anweisung ausführenden Benutzers.

Übrigens stehen nicht alle vorhandenen Ereignisspalten für jedes Ereignis zur Verfügung. So können Sie die Spalte *Duration* zum Beispiel nur dann auswählen, sofern auch ein entsprechendes *Completed* Ereignis (etwa *SP:Completed*) in das Protokoll aufgenommen wurde.

Damit soll die kurze Einführung in die vorhandenen Ereignisse und Ereignisspalten zunächst abgeschlossen sein. Sie werden im weiteren Verlauf dieses Buches noch mehr Ereignisse kennenlernen, sobald wir diese für entsprechende Konzepte oder Messungen benötigen.

Ereignisse filtern

Es ist möglich, die auftretenden Ereignisse nach bestimmten Werten in den Ereignisspalten zu filtern. So können Sie zum Beispiel nur Ereignisse in die Ablaufverfolgung einschließen, die eine Minimaldauer überschreiten oder etwa nur im Kontext einer bestimmten Datenbank auftreten. Über die Schaltfläche SPALTENFILTER (siehe Abbildung 4.9) öffnet sich der Dialog zur Konfiguration von Spaltenfilterbedingungen. Abbildung 4.11 zeigt ein Beispiel

für eine Filterbedingung, die bewirkt, dass nur Anweisungen in das Protokoll geschrieben werden, bei denen eine Datenbank betroffen ist, deren Name mit »Adventure« beginnt.

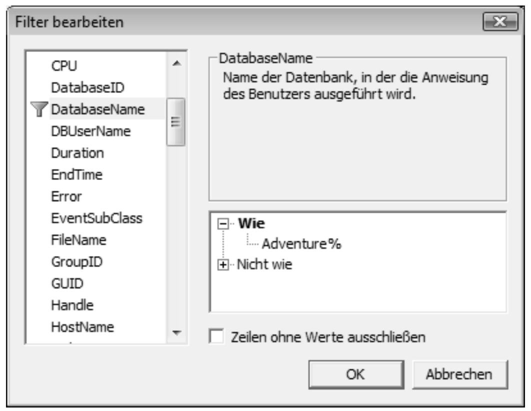


Abbildung 4.11:
Konfiguration einer Filterbedingung für die Ablaufverfolgung

Bitte beachten Sie, dass Sie nur nach solchen Spalten filtern können, die Sie auch in das Protokoll eingeschlossen haben.

4.3.3 Arbeiten mit Ablaufverfolgungen

Über die Menüleiste oder über den Menüeintrag im Hauptmenü ist es möglich, Ablaufverfolgungen anzuhalten, zu beenden und erneut zu starten. Von dieser Möglichkeit werden Sie zum Beispiel dann Gebrauch machen, wenn eine Ablaufverfolgung läuft, deren Eigenschaften (also etwa Ereignisse oder Ereignisspalten) Sie verändern möchten. Für eine laufende Ablaufverfolgung ist eine Änderung der Eigenschaften nicht möglich. Hierfür müssen Sie die Ablaufverfolgung zunächst beenden und dann – nach der Änderung der Eigenschaften – erneut starten.

Falls Sie feststellen, dass Sie versehentlich einen Zeitraum protokolliert haben, der nicht interessant für Sie ist, können Sie eine laufende Ablaufverfolgung durch Betätigen der entsprechenden Schaltfläche in der Menüleiste auch leeren – dadurch wird das Protokollfenster bereinigt.

Möglich ist auch eine Speicherung der Ablaufverfolgung. Hierbei können Sie wiederum wählen, ob Sie die Ablaufverfolgung in einer Datei oder in einer Datenbanktabelle ablegen möchten. Entscheiden Sie sich für das Dateiformat, können Sie hier wiederum zwischen einem XML-Format oder einem nativen Format wählen. Die Datei im nativen Format hat standardmäßig die Endung *.trc*. Dateien mit dieser Endung sind nach der Installation der SQL Server Client-Werkzeuge automatisch mit dem Profiler verknüpft, können also einfach per Doppelklick geöffnet werden. Dadurch haben Sie die Möglichkeit, Ablaufverfolgungen auch mit Kollegen auszutauschen oder zu archivieren. Außerdem können Sie die gespeicherten Ablaufverfolgungsdateien auch wieder einspielen, sofern Sie mindestens die in der Vorlage T*SQL_Replay* enthaltenen Ereignisse und Ereignisspalten in Ihr Protokoll mit aufgenommen haben.

Erstellen und Ändern von Vorlagen

Falls die in Tabelle 4.1 aufgezählten Standardvorlagen für Ihre Zwecke nicht ausreichend sind, können Sie eigene Vorlagen hinzufügen oder auch bestehende Vorlagen verändern. Ich empfehle Ihnen, die Standardvorlagen nicht zu verändern. Erzeugen Sie lieber eigene Vorlagen. Dies funktioniert zum Beispiel über DATEI • VORLAGEN • NEUE VORLAGE. Anschließend können Sie die Ereignisse und Ereignisspalten konfigurieren. Eine weitaus einfachere Möglichkeit der Vorlagenerstellung ist die Verwendung einer bestehenden Vorlage als Muster. Starten Sie hierzu einfach die Konfiguration einer neuen Ablaufverfolgung unter Verwendung einer bestehenden Vorlage. Verändern Sie dann die Ereignisse und Ereignisspalten nach Ihren Wünschen. Sobald Sie hiermit fertig sind, starten Sie die Ablaufverfolgung. Über den Menüeintrag DATEI • SPEICHERN UNTER/ABLAUFVERFOLGUNGSVORLAGE... können Sie aus dieser konfigurierten Ablaufverfolgung die Vorlage herausziehen und abspeichern.

4.3.4 Serverseitige Ablaufverfolgungen

Wie bereits weiter oben erwähnt, sollten Sie die grafische Oberfläche des Profilers nach Möglichkeit nicht verwenden. Die hierfür erforderliche Verbindung zum SQL Server und das Übertragen der Ablaufverfolgungsdaten für die Darstellung im Profiler benötigen Serverressourcen, die ihrerseits den Server zusätzlich belasten. Viel unangenehmer wirkt sich hier allerdings die Tatsache aus, dass diese Ressourcen die Ergebnisse der Messungen ganz erheblich verfälschen können.

Wenn Sie eine Ablaufverfolgung aus dem Profiler heraus starten, dann wird auf dem Server immer eine neue Ablaufverfolgung angelegt. Sie können dies überprüfen, indem Sie die Systemsicht `sys.traces` abfragen, die alle konfigurierten Ablaufverfolgungen zurückgibt. Wundern Sie sich hierbei bitte nicht, dass immer eine Ablaufverfolgung mit der *ID 1* existiert: Dies ist eine Ablaufverfolgung, die stets beim Start von SQL Server automatisch ausgeführt wird. Diese Ablaufverfolgung benötigt nur sehr wenige Ressourcen.

Eine Ablaufverfolgung benötigt also generell nicht den Profiler für die Protokollierung. Das Protokoll kann durchaus in eine Datei geschrieben werden, die dann später ausgewertet wird. Leider ist es aber nicht möglich, den Profiler zu beenden und durch den Profiler gestartete und noch laufende Ablaufverfolgungen weiterhin ausführen zu lassen. Der Profiler sorgt in diesem Fall dafür, dass alle noch in seiner Umgebung laufenden Ablaufverfolgungen beendet werden.

Sie können eine serverseitige Ablaufverfolgung, die ohne den Profiler auskommt, unter Verwendung von T-SQL konfigurieren. Eine solche Ablaufverfolgung kann das Protokoll in Dateien ablegen. Diese Dateien lassen sich später mit dem Profiler oder auch mit T-SQL auswerten.

Für die Konfiguration einer serverseitigen Ablaufverfolgung mittels T-SQL stehen Ihnen drei gespeicherte Prozeduren zur Verfügung:

- `sp_trace_create`. Mit dieser Prozedur legen Sie eine neue Ablaufverfolgung an. Hierbei geben Sie den Dateinamen für die Speicherung sowie eine maximale Dateigröße an. Weiter können Sie festlegen, dass ein Dateirollover (Ich bleibe hier bei der von Microsoft eingeführten Terminologie, um Sie nicht zu verwirren.) durchgeführt werden soll, dass also nach Erreichen der maximalen Dateigröße eine neue Datei begonnen wird. Diese Prozedur gibt eine *TraceId* zurück, die eine Ablaufverfolgung innerhalb des Servers eindeutig kennzeichnet. Sie benötigen diese ID für die beiden anderen Prozeduren zur Konfiguration der Ereignisse und des Status. Das Erzeugen einer Ablaufverfolgung über diese Prozedur bewirkt nicht, dass die Ablaufverfolgung gestartet wird.
- `sp_trace_setevent`. Mit dieser Prozedur legen Sie die in die Ablaufverfolgung aufzunehmenden Ereignisse und Ereignisspalten fest. Für die einzelnen Ereignisse und Ereignisspalten existiert in der Dokumentation eine Integer-Codierung. Die Prozedur erwartet also nicht etwa die Namen der Ereignisse und Ereignisspalten, sondern die entsprechend codierten Integer-Werte der zu protokollierenden Ereignisse und Ereignisspalten.
- `sp_trace_setstatus`. Diese Prozedur ermöglicht das Starten, Anhalten und Beenden einer Ablaufverfolgung.

Die manuelle Konfiguration ist recht umständlich, da Sie sicherlich nicht die Codierungen für alle Ereignisse und Ereignisspalten im Kopf behalten werden. Diese Codes müssen Sie aus der Dokumentation heraussuchen. Glücklicherweise existiert eine bequemere Methode, ein T-SQL-Skript für eine Ablaufverfolgung zu erzeugen, die gleichzeitig auch weniger fehleranfällig ist. Hier kommt wieder der Profiler ins Spiel.

Eine im Profiler konfigurierte Ablaufverfolgung kann als T-SQL-Skript für die serverseitige Erzeugung dieser Ablaufverfolgung gespeichert werden. Den entsprechenden Menüpunkt finden Sie unter DATEI • EXPORTIEREN • SKRIPT FÜR ABLAUFVERFOLGUNGSDEFINITION ERSTELLEN. Das folgende Listing zeigt ein Beispiel ein so erzeugten Skripts, unter Verwendung der Vorlage *Tuning*:

```
-- Create a Queue
declare @rc int
declare @TraceID int
declare @maxfilesize bigint
set @maxfilesize = 5

exec @rc = sp_trace_create @TraceID output, 0
                           ,N'InsertFileNameHere', @maxfilesize, NULL
if (@rc != 0) goto error

-- Set the events
declare @on bit
set @on = 1
exec sp_trace_setevent @TraceID, 10, 1, @on
exec sp_trace_setevent @TraceID, 10, 3, @on
exec sp_trace_setevent @TraceID, 10, 11, @on
exec sp_trace_setevent @TraceID, 10, 35, @on
exec sp_trace_setevent @TraceID, 10, 12, @on
exec sp_trace_setevent @TraceID, 10, 13, @on
```

```
exec sp_trace_setevent @TraceID, 45, 1, @on
exec sp_trace_setevent @TraceID, 45, 3, @on
exec sp_trace_setevent @TraceID, 45, 11, @on
exec sp_trace_setevent @TraceID, 45, 35, @on
exec sp_trace_setevent @TraceID, 45, 12, @on
exec sp_trace_setevent @TraceID, 45, 28, @on
exec sp_trace_setevent @TraceID, 45, 13, @on
exec sp_trace_setevent @TraceID, 12, 1, @on
exec sp_trace_setevent @TraceID, 12, 3, @on
exec sp_trace_setevent @TraceID, 12, 11, @on
exec sp_trace_setevent @TraceID, 12, 35, @on
exec sp_trace_setevent @TraceID, 12, 12, @on
exec sp_trace_setevent @TraceID, 12, 13, @on

-- Set the trace status to start
exec sp_trace_setstatus @TraceID, 1

-- display trace id for future references
select TraceID=@TraceID
goto finish

error:
select ErrorCode=@rc

finish:
go
```

Aus Gründen der Übersichtlichkeit wurden aus dem Skript einige Kommentare sowie nicht benötigte Bereiche entfernt.

Sie erkennen im Skript sehr schön, dass der Profiler Ihnen die Codierung der Ereignisse und Ereignisspalten abnimmt. Der fett gedruckte Bereich muss von Ihnen angepasst werden. Hier geben Sie den Namen der Ablaufverfolgungsdatei an, wobei die Dateiendung *.trc* automatisch angefügt wird. Was Sie dort spezifizieren, sind die maximale Größe der Datei und ob Sie ein Dateirollover wünschen. Das Skript konfiguriert die Ablaufverfolgung und startet sie zugleich. Falls Sie die Ablaufverfolgung anhalten oder beenden möchten, müssen Sie die Prozedur `sp_trace_setstatus` aufrufen. Für diese Prozedur benötigen Sie in jedem Fall die *TraceId* als Parameter. Deshalb gibt das Skript die ID der erzeugten Ablaufverfolgung im Ergebnisbereich aus. In der Regel wird diese *TraceId* den Wert 2 haben. Die Prozedur `sp_trace_setstatus` erwartet die *TraceId* als ersten Parameter. Mit Hilfe des zweiten Parameters geben Sie an, was Sie mit der Ablaufverfolgung machen möchten:

```
-- Ablaufverfolgung anhalten
exec sp_trace_setstatus 2, 0

-- Ablaufverfolgung beenden und benötigte Ressourcen entfernen
exec sp_trace_setstatus 2, 2
```

Falls Sie die *TraceId* vergessen, können Sie die Systemsicht `sys.traces` verwenden, um die Informationen über alle derzeit konfigurierten Ablaufverfolgungen zu erhalten.

Durch die Möglichkeit, eine Ablaufverfolgung per Skript zu konfigurieren, eröffnen sich Ihnen interessante Perspektiven. So können Sie zum Beispiel mehrere Skripte erstellen und diese als eigene Vorlagen in den Vorlagen-Explorer des Management Studios aufnehmen. Dadurch müssen Sie mit dem Profiler nicht jedesmal von vorne beginnen, wenn Sie eine serverseitige Ablaufverfolgung ausführen möchten. Außerdem ist es natürlich möglich, T-SQL-Skripte in Aufträge des SQL Server Agents einzufügen. Dadurch können Sie Ablaufverfolgungen zeitgesteuert starten und beenden. Darüber hinaus kann der Agent auch bestimmte Ressourcen überwachen und beim Über- oder Unterschreiten konfigurierter Schwellwerte Aufträge starten. Dadurch ist es möglich, Ablaufverfolgungen auszuführen, sobald bestimmte Systemparameter als kritisch eingestufte Werte erreichen.

4.3.5 Arbeiten mit Ablaufverfolgungsdateien

Nehmen wir nun also an, Sie haben eine oder mehrere Ablaufverfolgungsdateien mit relevanten Informationen erzeugt. Wie können Sie anschließend vorgehen, um diese Informationen auszuwerten?

Für kleinere Dateien bietet sich eine Analyse direkt im Profiler an. Wie bereits gesagt, können Sie Ablaufverfolgungsdateien einfach per Doppelklick im Profiler öffnen. Dort können Sie über BEARBEITEN • SUCHEN... nach Ereignissen oder Ereignisspalten zu suchen.

Wesentlich interessanter ist jedoch die Möglichkeit, die enthaltenen Informationen über T-SQL zu analysieren. Hierzu verwenden Sie die Funktion `fn_trace_gettable()`, die den Inhalt einer Ablaufverfolgungsdatei in eine Tabelle umwandelt. Diese Funktion erwartet als ersten Parameter den Namen der zu untersuchenden Ablaufverfolgungsdatei. Hier muss stets der Name des ursprünglich bei `sp_trace_create` angegebenen Dateinamens verwendet werden. Es können ja unter Umständen auch mehrere Ablaufverfolgungsdateien existieren, nämlich dann, wenn Sie den Dateirollover eingestellt haben. Und dies bringt uns auch schon zum zweiten Parameter. Hier geben Sie die Anzahl der Dateien, die eingelesen werden sollen, an. Für diesen Parameter können Sie einfach den Wert `DEFAULT` angeben. Es werden dann alle Dateien verarbeitet.

Die folgende Anweisung überführt die angegebene Ablaufverfolgung in eine Tabelle und stellt das Ergebnis im Abfrageeditor dar:

```
select * from fn_trace_gettable('c:\traces\Prblog.trc', default)
```

Die von `fn_trace_gettable()` zurückgegebene Tabelle enthält immer alle Ereignisspalten, unabhängig davon, ob Sie die entsprechende Spalte in die Ablaufverfolgung eingeschlossen haben oder nicht. Für alle nicht eingeschlossenen Spalten wird einfach der Wert NULL zurückgeliefert. falls Sie nur bestimmte Ereignisspalten sehen möchten, können Sie diese Spalten einfach in der `SELECT`-Anweisung angeben.

Natürlich ist es auch möglich, unter Verwendung von `fn_trace_gettable()` eine Tabelle zu erzeugen, in der die Daten gespeichert werden. Dies funktioniert zum Beispiel so:

```
select cast(TextData as nvarchar(max)) as TextData
      ,StartTime
      ,Duration
  into TraceData
  from fn_trace_gettable('c:\traces\Prblog.trc', default)
 where Duration is not null
```

Hier werden nur die Spalten *TextData*, *StartTime* und *Duration* in eine Tabelle *TraceData* übertragen, wobei lediglich Zeilen herausgefiltert werden, die überhaupt einen Wert in der Spalte *Duration* besitzen.

Sobald Sie Ihre Ablaufverfolgungsdaten in eine Tabelle kopiert haben, stehen Ihnen alle Möglichkeiten von T-SQL für eine Auswertung zur Verfügung. So können Sie zum Beispiel Indizes für Spalten dieser Tabelle anlegen, damit Ihre Auswertungen entsprechend beschleunigt werden. Wenn Sie beispielsweise nur die zehn längsten Abfragen in einem bestimmten Zeitraum oder die Abfragen, die am häufigsten verwendet werden, herausfinden möchten, so ist dies sehr einfach möglich. Insgesamt steht Ihnen auf diese Weise eine hervorragende Möglichkeit zur Analyse der ausgeführten Abfragen zur Verfügung.

4.4 Der Windows-Systemmonitor

Da der SQL Server-Dienst im Rahmen des Windows-Betriebssystems ausgeführt wird, kann er normalerweise nur im Zusammenhang mit dem Betriebssystemzustand analysiert werden. Der Windows-Systemmonitor ist hierfür ein geeignetes Werkzeug, zumal mit der Installation einer SQL Server-Instanz zusätzliche Leistungsindikatoren zum Systemmonitor hinzugefügt werden, die eine Überwachung von SQL Server ermöglichen. Insgesamt sind dies ca. 800 Leistungsindikatoren. Wenn Sie zu einem Sammlungssatz Indikatoren hinzufügen, werden Sie feststellen, dass Ihnen auch SQL Server-spezifische Leistungsindikatoren angeboten werden. Abbildung 4.12 zeigt ein Beispiel.

Da es nahezu unmöglich ist, die Bedeutung aller verfügbaren Indikatoren zu kennen, ist es sehr nützlich, dass Sie durch die Auswahl der Option BESCHREIBUNG ANZEIGEN (siehe Abbildung 4.12) eine schnelle Übersicht über den gerade im Dialog ausgewählten Leistungsindikator erhalten. Das ist wirklich eine große Hilfe.

Falls Sie an dieser Stelle eine detaillierte Beschreibung der Handhabung des Systemmonitors erwarten, muss ich Sie enttäuschen. Sie erhalten hier nur eine kurze aber ausreichende Einführung in die Benutzung des Systemmonitors. Sie erfahren, wie Sie den Systemmonitor zur Überwachung von SQL Server einsetzen, und welche Möglichkeiten Ihnen hierfür zur Verfügung stehen. Eine umfassende Hilfe finden Sie in der Windows-Dokumentation.

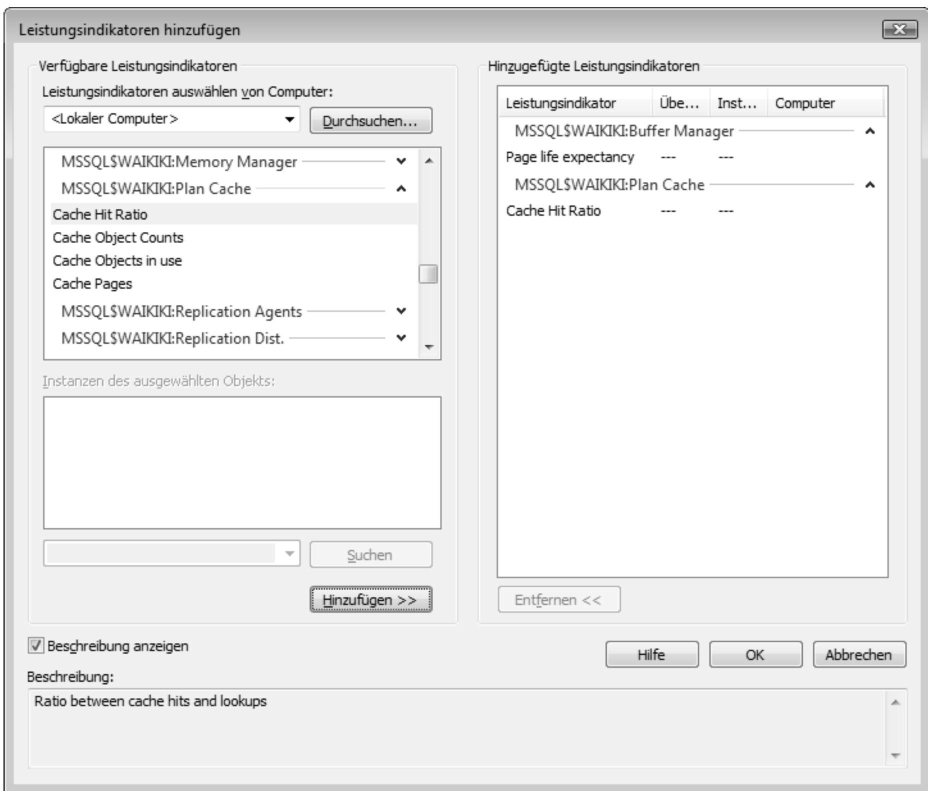


Abbildung 4.12: *SQL Server-Leistungsindikatoren des Systemmonitors*

4.4.1 Wichtige Leistungsindikatoren

Natürlich ist es nicht möglich, an dieser Stelle alle kanpp 800 SQL Server-Leistungsindikatoren aufzuzählen und zu erläutern. Auf einige wichtige Indikatoren und deren Bedeutung möchte ich Sie jedoch hinweisen. Oft reicht es aus, bei Problemen diese Indikatoren zu beobachten, um dann die Analyse in eine bestimmte Richtung zu vertiefen. Auf meinem PC habe ich übrigens nach der Installation von SQL Server ein sprachliches Durcheinander vorgefunden. Obwohl ich sowohl ein deutsches Betriebssystem als auch eine deutschsprachige SQL Server-Version installiert habe, wurden die Indikatoren des Betriebssystems in deutscher und die von SQL Server in englischer Sprache angezeigt. Da die Übersetzung der Bezeichnungen nicht immer einfach ist, finden Sie in der folgenden Aufstellung für alle Leistungsindikatoren sowohl die deutschen als auch (jeweils in Klammern) die englischen Bezeichnungen.

Wichtige Leistungsindikatoren des Betriebssystems

- **Physikalischer Datenträger → Durchschnittl. Warteschlangenlänge des Datenträgers (Physical Disk → Avg. Disk Queue Length)**. Dieser Indikator zeigt an, wie viele E/A-Vorgänge in der Warteschlange warten abgearbeitet zu werden. Falls dieser Wert über einen längeren Zeitraum größer als 2 für einen physikalischen Datenträger in einem Array ist, dann ist Ihr E/A-System sehr wahrscheinlich nicht leistungsfähig genug. Sie sollten dann zusätzliche Festplatten hinzufügen.
- **Logischer Datenträger → Zeit (%) (Logical Disk → % Disk Time)**. Dieser Indikator informiert über die Auslastung eines gesamten Festplatten-Arrays. Er enthält den prozentualen Zeitanteil, den das Array insgesamt für Lese- und Schreiboperationen benötigt hat. Auf Dauer sollte dieser Wert unter 60 % liegen. Falls der Wert über einen längeren Zeitraum zu hoch ist, sollten Sie darüber nachdenken, wie Sie den E/A-Durchsatz auf Ihrem System erhöhen können. Mögliche Maßnahmen hierfür wären etwa das Hinzufügen weiterer Festplatten zum Array oder der Austausch existierender Festplatten durch leistungsfähigere. Eventuell kann auch der Tausch des gesamten Arrays oder/und des Controllers erforderlich sein.
- **Prozessor → Prozessorzeit (%) (Processor → % Processor Time)**. Dies ist ein Indikator für den Zeitraum, in dem der Prozessor arbeitet, sich also nicht im Leerlauf befindet. Die durchschnittliche Auslastung des Prozessors sollte 50 % nicht übersteigen. Dieser Indikator kann die Belastung jedes einzelnen Prozessors, aber auch die Gesamtbelastung aller Prozessoren messen. Wenn Sie über einen längeren Zeitraum mehr als 80 % Belastung für alle Prozessoren beobachten, dann benötigen Sie wahrscheinlich mehr oder leistungsfähigere Prozessoren. Möglicherweise ist dies aber auch ein Anzeichen für einen zu kleinen Hauptspeicher, was dazu führen kann, dass der Plancache nicht genügend Ausführungspläne speichern kann, sodass Re-Kompilierungen erforderlich sind.
- **Speicher → Seiten/s (Memory → Pages/sec)**. Dieser Indikator misst die Anzahl der Ein- und Auslagerungsvorgänge von Speicherseiten je Sekunde. Jede Operation, die eine Speicherseite von der Festplatte liest oder eine Seite auf die Festplatte schreibt, erhöht den Wert dieses Indikators. Der Wert sollte unter 20 liegen, andernfalls ist dies ein Beleg für zu viele Auslagerungsvorgänge, also eine zu starke Verwendung des virtuellen Speichers. Falls Sie SQL Server eine dedizierte Maschine zugewiesen haben (was, nebenbei bemerkt, eine gute Idee ist), also außer SQL Server nichts weiter auf der Hardware ausgeführt wird, dann sollte dieser Wert null sein, vielleicht mit Ausnahme von einigen kurzeitigen Ausreißern nach oben. Ist der Wert zu hoch (im Mittel größer als 20 über einen Tag), so benötigen Sie mehr RAM. Falls außer SQL Server noch andere Applikationen auf dem Server laufen, hilft es eventuell auch, diese Applikationen auf einen anderen Server auszulagern und SQL Server einen dedizierten Server zu spendieren.
- **Speicher → Verfügbare Bytes (Memory → Available Bytes)**. Der Indikator zeigt den verfügbaren freien Hauptspeicher an. Dieser Wert sollte dauerhaft etwa 5 % größer als der des verfügbaren Hauptspeichers sein. Falls der Wert zu gering ist, werden Auslagerungen durchgeführt, was die Leistung ganz entscheidend verschlechtert. In diesem Fall benötigen Sie sehr wahrscheinlich mehr RAM. Möglich ist auch, dass die SQL Server-Konfigurationsparameter dahingehend verändert wurden, dass der für SQL Server maximal verfügbare Speicher auf einen zu kleinen Wert eingestellt wurde. Überprüfen Sie in diesem Fall bitte die Einstellung (zum Beispiel mittels `sp_configure`) und ändern Sie sie gegebenenfalls.

- **System → Prozessor Warteschlangenlänge (System → Processor Queue Length)**. Die Prozessorwarteschlangenlänge ist ein Indikator für Prozesse, die nicht ausgeführt werden können, weil sie auf die Zuteilung von CPU-Zeit warten müssen. Hierbei müssen Sie folgendes beachten: Nicht jeder Prozessor verwendet seinen eigenen Indikator. Die Warteschlangenlänge bezieht sich also auf alle Prozessoren in einem Computer. Falls Sie die durchschnittliche Warteschlangenlänge je Prozessor interessiert, müssen Sie diesen Wert durch die Anzahl der Prozessoren teilen. Generell sollten auf einer SQL Server-Maschine dauerhaft nicht mehr als zwei Prozesse je Prozessor in der Warteschlange stehen. Bei vier Prozessoren ist demnach eine Warteschlangenlänge, die über einen längeren Zeitraum größer als acht ist, ein Indiz dafür, dass Sie zu wenige oder zu schwache Prozessoren verwenden.

Wichtige SQL Server-Leistungsindikatoren

- **SQL Server:Allgemeine Statistik → Benutzerverbindungen (SQL Server:General Statistics → User Connections)**. Dieser Indikator informiert über die Anzahl an Verbindungen zum SQL Server. Ist dieser Wert größer als die maximal zulässige Anzahl von Worker Threads, dann müssen sich Verbindungen Worker Threads teilen. Dies kann negative Auswirkungen auf die Leistung haben. Falls Sie also über einen längeren Zeitraum mehr als 255 Benutzerverbindungen beobachten, dann sollten Sie den Konfigurationsparameter *Maximum Worker Threads* überprüfen und erhöhen. Standardmäßig wird dieser Wert dynamisch vergeben, was normalerweise die richtige Wahl ist, da Sie sich dann nicht selbst um die Konfiguration kümmern müssen. Immerhin kann es aber sein, dass jemand die maximale zulässige Anzahl Worker Threads reduziert hat; daher sollten Sie diese Einstellung überprüfen.
- **SQL Server:Datenbanken → Ausstehende Protokollleerungen/Sekunde (SQL Server: Databases → Log Flush Waits/sec)**. Dieser Indikator steht Ihnen pro Datenbank zur Verfügung, aber auch als Summen-Indikator für alle vorhandenen Datenbanken gemeinsam. Er zeigt die Anzahl der durch `COMMIT` abgeschlossenen (impliziten oder expliziten) Transaktionen an, die auf das Schreiben des Transaktionsprotokolls warten. Wie Sie bereits in Kapitel 2 erfahren haben, wird das Transaktionsprotokoll synchron geschrieben. Eine Transaktion ist somit erst dann abgeschlossen, wenn die zur Transaktion gehörenden Protokolleinträge aus dem Hauptspeicher in die Protokolldatei übertragen wurden. Falls der Wert für diesen Indikator im normalen Betrieb einer OLTP-Datenbank (also nicht etwa während der Ausführung von Massenimporten) dauerhaft größer als eins ist, ist dies ein Indiz dafür, dass Sie an dieser Stelle ein Problem mit dem E/A-System haben. Möglicherweise müssen Sie für das Transaktionsprotokoll schnellere Festplatten installieren oder lediglich den Schreibcache einschalten (siehe das Beispiel in Kapitel 2). Oftmals kommt es auch vor, dass die Protokoll- und Datendateien nebeneinander auf demselben physikalischen Datenträger existieren. Hier also noch einmal der Hinweis: Trennen Sie Protokoll- und Datendateien voneinander und installieren Sie diese beiden Dateitypen auf unterschiedlichen Datenträgern.

  Eventuell können Sie auch Ihre SQL-Anweisungen dahingehend anpassen, dass Sie statt vieler kleiner Transaktionen wenige größere Transaktionen absetzen. Erinnern Sie sich hierzu bitte nochmals an das in Kapitel 2 gezeigte Beispiel.

- **SQL Server:Datenbanken → Transaktionen/Sekunde (SQL Server:Databases → Transaction/sec)**. Auch dieser Indikator kann je Datenbank oder als Summenwert über alle Datenbanken konfiguriert werden. Er steht für die Anzahl von Transaktionen, die SQL Server je Sekunde verarbeitet. Da dieser Wert von der Größe einer Transaktion abhängt, kann hierfür kein Richtwert angegeben werden. OLTP-Systeme verarbeiten normalerweise viele kleine Transaktionen. Sie sollten diesen Wert beobachten, falls Sie feststellen, dass Ihr SQL Server-System sporadisch langsamer reagiert als normal. So kann zum Beispiel ein sprunghafter Anstieg zu einem bestimmten Zeitpunkt, der vielleicht auch noch zyklisch (also etwa täglich) wiederkehrt, die Analyse der Ursache für eine plötzliche Verschlechterung der Abfrageleistung des Gesamtsystems erleichtern.
- **SQL Server:Plan Cache → Cachetrefferqoute (SQL Server:Plan Cache → Cache Hit Ratio)**. Kompilierte Ausführungspläne werden nach Möglichkeit im sogenannten Plancache gespeichert und wiederverwendet. Das Ziel ist eine Vermeidung von erneuten Kompilierungen, da dies ein CPU-intensiver Vorgang ist. Jedesmal, wenn der Abfrageprozessor einen Plan erstellt, wird er zunächst versuchen, ein Muster für einen entsprechenden Plan im Plancache zu finden, bevor er die Erstellung des Plans in Angriff nimmt. Falls diese Suche erfolgreich war, wird der vorhandene Plan verwendet. In Kapitel 9 werden wir uns mit dieser Thematik eingehend befassen. In OLTP-Systemen sollte die Trefferquote so hoch wie möglich sein. Eine kleine Cache-Trefferquote bedeutet eine hohe Kompilierungs- bzw. Re-Kompilierungsquote, was zu einer übermäßigen Belastung der CPU führen kann. Betrachten Sie diesen Indikator also im Zusammenhang mit dem Indikator zur Messung der Prozessorzeit. Falls die Prozessorauslastung hoch und die Trefferquote im Plancache gering ist, und gleichzeitig auch noch der Wert für die Transaktionen pro Sekunde signalisiert, dass SQL Server Transaktionen verarbeitet, dann sollten Sie prüfen, in welcher Weise Ihre Anwendungen Abfragen an den SQL Server absetzen. Kapitel 9 untersucht diese Thematik eingehend und zeigt, was Sie beachten müssen, um eine angemessene Plancache-Trefferquote zu erzielen.
- **SQL Server:Puffer-Manager → Lebenserwartung von Seiten (SQL Server:Buffer Manager → Page Life Expectancy)**. Aus Kapitel 2 wissen Sie bereits, dass der Datencache eine entscheidende Komponente für die E/A-Performance – und somit für die gesamte Leistung – des Systems ist. Immer dann, wenn eine Lese- bzw. Schreiboperation direkt den Datencache verwendet, funktioniert die entsprechende Operation um ein Vielfaches schneller als eine entsprechende physische Operation direkt auf dem Datenträger. Da der Datencache nicht unbegrenzt groß ist, werden allerdings auch Seiten, auf die über einen längeren Zeitraum nicht zugegriffen wurde, aus dem Cache entfernt; nämlich dann, wenn der Cache voll ist und auf Datenseiten zugegriffen wird, die bislang nicht im Cache stehen. In OLAP-Systemen, die naturgemäß mit großen Lesevorgängen arbeiten, ist dieser Indikator weniger von Bedeutung. Hier ist es durchaus normal, dass Leseoperationen viele Daten lesen, die nicht alle in den Datencache passen. In OLTP-Systemen sieht dies allerdings anders aus. Das E/A-Muster einer OLTP-Anwendung besteht in der Regel aus vielen kleinen E/A-Operationen. Für diese Art von Anwendung sollte die in Sekunden angegebene durchschnittliche Lebenserwartung einer Seite nicht unter 300 liegen. Andernfalls besitzen Sie möglicherweise zu wenig Hauptspeicher.

  Der Indikator kann auch ein anderes Problem aufzeigen. Plötzliche Änderungen der durchschnittlichen Lebenserwartung einer Datenseite sind ein Indiz dafür, dass viele

Daten in den Datencache übertragen werden. In einer OLTP-Datenbank stellt eine derartige Situation ein Alarmsignal dar. Die Ursache für ein solches Verhalten kann zum Beispiel ein fehlender Index sein, wodurch eine sequenzielle Suche, also beispielsweise ein Tabellen-Scan, erforderlich ist. Es werden also möglicherweise zu viele Leseoperationen ausgeführt. Eine weitere Ursache hierfür kann sein, dass Ihr OLTP-System für Reporting-Zwecke verwendet wird. Auch dann ist es möglich, dass durch große Lesevorgänge während eines bestimmten Zeitraumes die durchschnittliche Lebenserwartung einer Seite im Datencache sprunghaft absinkt oder ansteigt. Denken Sie nur an den Assistenten der Geschäftsführung, der mit einem Reporting-Werkzeug ausgefeilte Berichte erstellt, um diese dem Geschäftsführer vorzulegen, der dies während der normalen Arbeitszeit erledigt, in der das System also gerade produktiv ist und gleichzeitig sehr viele Benutzer bedienen soll.

- **SQL Server:SQL Statistik → Batchanforderungen/Sekunde (SQL Server:SQL Statistics → Batch Requests/sec)**. Dieser Indikator gibt Auskunft darüber, wie viele T-SQL-Stapel pro Sekunde zur Abarbeitung an den Server gesendet wurden. Sie können diesen Indikator für zwei Analysen verwenden: Zum einen sollten Sie beobachten, ob der Wert während eines bestimmten Zeitraumes Ausreißer aufweist, die eventuell sogar zyklisch auftreten. Dies könnte zum Beispiel darauf hindeuten, dass zu bestimmten Zeiten Prozesse laufen, die die Serverlast insgesamt erhöhen. Möglicherweise können Sie solche Prozesse auf Zeiten geringerer sonstiger Aktivitäten verschieben. Beobachten Sie hierfür auch andere Indikatoren, die Auskunft über die Auslastung der CPU- oder des E/A-Systems geben.

  Eine weitere Möglichkeit ist die Beobachtung dieses Indikators im Zusammenhang mit den erforderlichen SQL-Kompilierungen je Sekunde (siehe nächster Punkt). Falls in einem OLTP-System die Anzahl der Kompilierungen je Sekunde nicht wesentlich geringer ist als die Batch-Anforderungen je Sekunde, dann erfordern die meisten Ihrer T-SQL-Stapel eine Kompilierung, etwa weil im Plancache kein geeigneter Abfrageplan gefunden wurde. Normalerweise ist dies ein Indiz dafür, dass Ihre Anwendungen keinen Gebrauch von parametrisierten Abfragen machen. Auch dies kann die Leistung negativ beeinflussen. Es ist gut möglich, dass Sie in diesem Fall auch eine erhöhte Auslastung der CPU feststellen. Mehr hierzu in Kapitel 9.

- **SQL Server:SQL Statistik → SQL Kompilierungen/Sekunde (SQL Server:SQL Statistics → SQL Compilations/sec)**. Dieser Indikator zeigt die insgesamt erforderlichen Kompilierungen von T-SQL-Stapeln an. Dies betrifft sowohl die initialen Kompilierungen für den Fall, dass im Plancache kein geeigneter Plan gefunden werden kann, als auch erneute Kompilierungen für Abfragen, für die zwar ein Plan gefunden wird, der aber wegen eines zu stark geänderten Kontextes nicht verwendet werden kann (siehe »Erneute Kompilierungen/Sekunde« im nächsten Punkt). Wenn Sie beobachten, dass dieser Indikator einen hohen Wert aufweist, werden Sie in den meisten Fällen auch eine erhöhte CPU-Last feststellen. Sollte dies der Fall sein, dann ist möglicherweise die Art und Weise, in der Ihre Anwendungen Anfragen an den SQL Server richten, die Ursache. Sie sollten untersuchen, ob Ihre Abfragen parametrisiert werden können, um erneute Kompilierungen zu vermeiden. In Kapitel 9 erfahren Sie mehr zum Thema »Parametrisierung von Abfragen«.

- **SQL Server:SQL Statistik → Erneute SQL-Kompilierungen/Sekunde (SQL Server: SQL Statistics → SQL Re-Compilations/sec)**. Der Indikator gibt die Anzahl der erforderlichen Re-Kompilierungen von Abfrageplänen an. Es ist durchaus möglich, dass für eine auszuführende Abfrage bereits ein entsprechender Plan im Plancache existiert, dieser Plan aber nicht verwendet werden kann, weil etwa die aktuelle Ausführungsumgebung eine Wiederverwendung nicht zulässt. Dieser Fall kann eintreten, wenn eine identische Abfrage mit unterschiedlichen SET-Optionen ausgeführt wird. (Siehe ebenfalls Kapitel 9). In diesem Fall muss die Abfrage erneut kompiliert werden. Normalerweise sollte dieser Indikator einen niedrigen Wert anzeigen. Falls die Anzahl der Re-Kompilierungen groß ist, ist dies eventuell ein Anzeichen dafür, dass Benutzerverbindungen mit unterschiedlichen Einstellungen aufgebaut werden. Eine hohe Anzahl von Re-Kompilierungen wirkt sich auch auf die CPU-Belastung aus. Beachten Sie also diesen Indikator im Zusammenhang mit der CPU-Nutzung und auch im Zusammenhang mit der Plancache-Trefferquote. Für die Anzahl der initialen Kompilierungen müssen Sie den Wert des Indikators vom Wert des Indikators *SQL-Kompilierungen/Sekunde* subtrahieren.

Der wohl am häufigsten missverstandene Wert ist die Trefferquote im Datencache, gemessen in Prozent durch den Indikator *SQL Server:Puffer Manager → Puffercache-Trefferquote (SQL Server:Buffer Manager → Buffer Cache Hit Ratio)*. Sie werden häufig den Hinweis finden, dass Ihr SQL Server mehr Hauptspeicher benötigt, sofern dieser Wert nicht deutlich über 90 % liegt. Im Prinzip ist diese Aussage absolut richtig. Bitte ziehen Sie daraus aber nicht die Schlussfolgerung, dass Ihr Hauptspeicher ausreichend bemessen ist, wenn der Wert nahe 100 % liegt. Der Wert wird durch Read Ahead-Operationen, die ja massenweise Daten von der Festplatte in den Pufferspeicher übertragen, verfälscht. Die durch Read Aheads in den Datencache übertragenen Seiten werden offensichtlich zur Trefferquote hinzugezählt. So haben Sie zum Beispiel unmittelbar nach einem Neustart von SQL Server sofort eine Cache-Trefferquote von nahezu 100 Prozent – und das, obwohl zu diesem Zeitpunkt definitiv keine einzige Datenseite im Cache war. Ich habe noch auf keiner SQL Server-Instanz einen Wert von weniger als 95 % beobachten können. Verfolgen Sie also lieber die durchschnittliche Verweildauer einer Datenseite im Cache, falls Sie feststellen möchten, ob Ihr System über ausreichend Hauptspeicherkapazität verfügt.

Wenn Sie Probleme mit der SQL Server-Performance feststellen, dann ist es meist ausreichend, zunächst mit einigen der oben genannten Indikatoren zu beginnen und dann tiefer zu forschen. Hierzu bieten sich benutzerdefinierte Sammlungen im Systemmonitor an, die Sie nach Ihren eigenen Vorstellungen zusammenstellen können.

Ich möchte Ihnen hierzu ein Beispiel präsentieren, damit Sie eine Vorstellung davon bekommen, wie Sie eine solche Zusammenstellung konfigurieren. Wir erstellen einen neuen benutzerdefinierten Sammlungssatz, dem zwei Sammlungen hinzugefügt werden. Die beiden Sammlungen sollen insgesamt alle oben aufgelisteten Indikatoren enthalten, wobei für die Systemindikatoren und die SQL Server-spezifischen Indikatoren jeweils eine eigene Sammlung angelegt werden soll.

Starten Sie den Systemmonitor und wählen dort in der Konsolenstruktur aus dem Kontextmenü für SAMMLUNGSSÄTZE • BENUTZERDEFINIERT den Eintrag NEU • SAMMLUNGSSATZ (Abbildung 4.13).

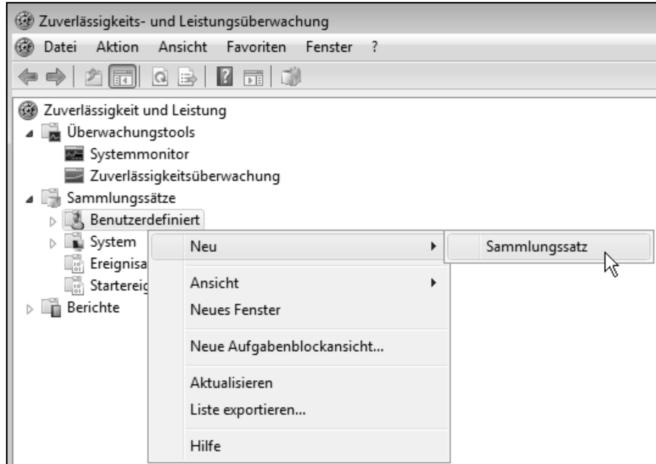


Abbildung 4.13: *Einen benutzerdefinierten Sammlungssatz erstellen*

Es wird ein Assistent gestartet, der Sie durch die Konfiguration eines Sammlungssatzes und – falls Sie dies wünschen – auch gleich durch die Konfiguration einer Sammlung für den neu angelegten Sammlungssatz führt. Bei der Konfiguration müssen Sie sorgfältig vorgehen, da der Assistent keine Möglichkeit bietet, rückwärts zu navigieren. Wenn Sie also in einem Schritt etwas übersehen und zu schnell zur nächsten Seite wechseln, bleibt Ihnen leider nur die Möglichkeit, die Konfiguration abzubrechen und von vorne zu beginnen. Wählen Sie auf der ersten Seite des Assistenten die Option MANUELL ERSTELLEN (ERWEITERT) und geben Sie dem Sammlungssatz einen Namen, also zum Beispiel »SQL Server Diagnose«, so wie in Abbildung 4.14 gezeigt.

Neuen Sammlungssatz erstellen
Wie soll dieser neue Sammlungssatz erstellt werden?
Name:
SQL Server Diagnose
Aus Vorlage erstellen (empfohlen)
Wie funktionieren Vorlagen?
Manuell erstellen (Erweitert)
Wie werden Sammlungssätze manuell ausgewählt?
Weiter
Fertig stellen
Abbrechen

Abbildung 4.14: *Einen neuen Sammlungssatz konfigurieren*

Klicken Sie auf WEITER und wählen Sie im anschließenden Dialog als einzuschließenden Datentyp DATENPROTOKOLLE ERSTELLEN und LEISTUNGSINDIKATOREN (Abbildung 4.15) aus.

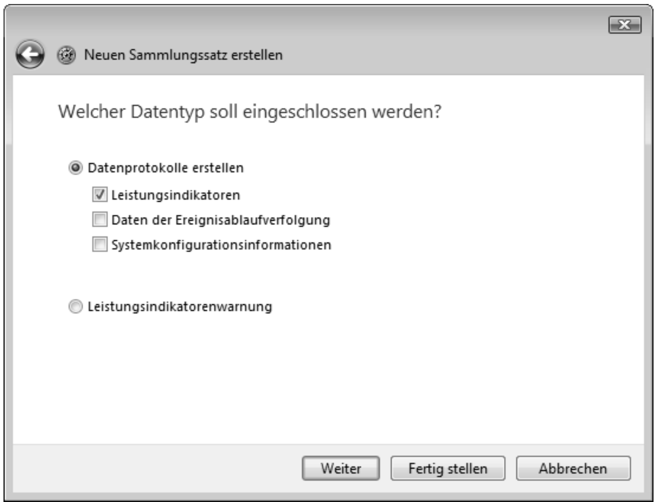


Abbildung 4.15: *Datentyp, der in den Sammlungssatz aufgenommen werden soll*

Sie können nun die Konfiguration des Sammlungssatzes durch Betätigen der Schaltfläche FERTIG STELLEN abschließen. Dadurch wird dem Sammlungssatz eine Standardsammlung mit Namen »DataCollector01« hinzugefügt, die wir allerdings nicht benötigen. Löschen Sie bitte diese Sammlung und fügen Sie dann über das Kontextmenü NEU • SAMMLUNG eine neue Sammlung hinzu. Diese Sammlung soll alle SQL Server-spezifischen Indikatoren enthalten. Geben Sie der Sammlung also einen entsprechenden Namen, zum Beispiel »SQL Server Indikatoren«. Geben Sie auch an, dass die Sammlung Leistungsindikatoren enthalten soll (Abbildung 4.16).

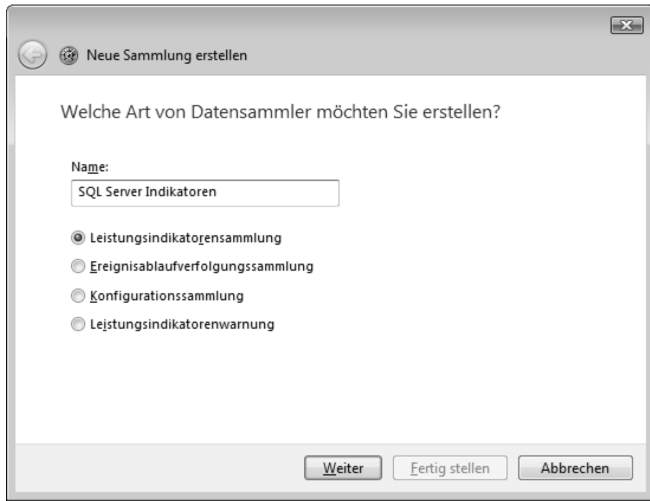


Abbildung 4.16: *Erstellen einer Sammlung für SQL Server-Indikatoren*

Auf der nächsten Seite geben Sie durch ein Abtastintervall an, wie häufig die Ereignisse für diese Sammlung aufgezeichnet werden sollen. Für diesen Test kann der Standardwert von 15 Sekunden beibehalten werden.

Sie können nun die erforderlichen Leistungsindikatoren hinzufügen. Nehmen Sie hier bitte alle Indikatoren auf, die im vorangegangenen Abschnitt unter dem Punkt »Wichtige SQL Server-Leistungsindikatoren« genannt wurden. Sobald Sie die Konfiguration abgeschlossen haben, wiederholen Sie bitte die Schritte zum Hinzufügen einer neuen Sammlung. Diesmal nennen Sie die Sammlung beispielsweise »System Leistungsindikatoren« und nehmen alle im letzten Abschnitt aufgezählten Indikatoren des Betriebssystems in die Sammlung auf. Es existiert nun ein Sammlungssatz mit zwei Datensammlungen, so wie in Abbildung 4.17 zu sehen.

Starten den Sammlungssatz zum Test, führen Sie einige SQL-Anweisungen aus und beenden Sie den Sammlungssatz anschließend wieder.

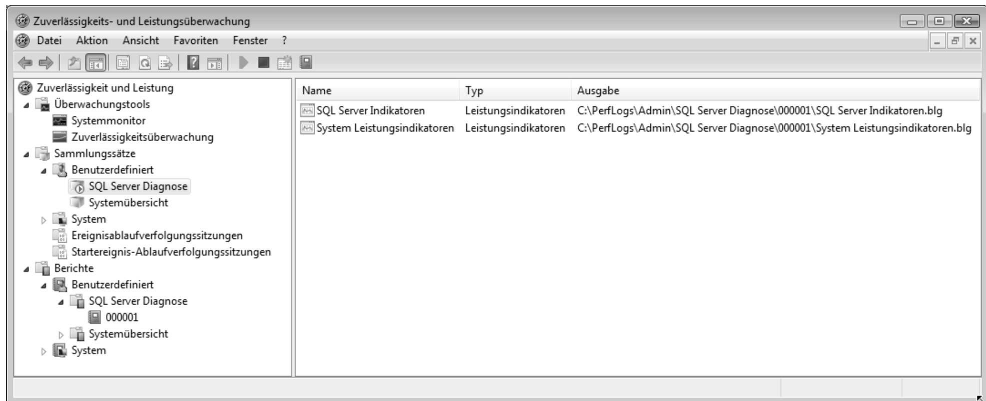


Abbildung 4.17: *Der angelegte Sammlungssatz mit den beiden Sammlungen*

Für einen Sammlungssatz gibt es eine Reihe von Einstellmöglichkeiten, die Sie über den Eigenschaftendialog erreichen. So ist es zum Beispiel möglich, den Sammlungssatz zeitgesteuert zu starten bzw. zu beenden oder beim Beenden des Sammlungssatzes zusätzliche Aufgaben auszuführen.

Außerdem wird für jeden neuen Sammlungssatz auch ein entsprechender Bericht angelegt, sobald Sie den zugehörigen Sammlungssatz starten. Diesen Bericht können Sie später öffnen und somit die in der Datensammlung aufgezeichneten Werte darstellen und auswerten. Für unseren Sammlungssatz sehen Sie in Abbildung 4.17 eine Berichtsvorlage mit dem gleichen Namen wie der Bericht sowie einen konkreten Bericht mit dem Namen »000001«.

Für einen erstellten Bericht gibt es unterschiedliche Darstellungsmöglichkeiten, aus denen Sie über das Menü ANSICHT eine auswählen können. Die beliebteste und bekannteste Darstellung ist sicherlich das Liniendiagramm des Systemmonitors, so wie in Abbildung 4.18 zu sehen.

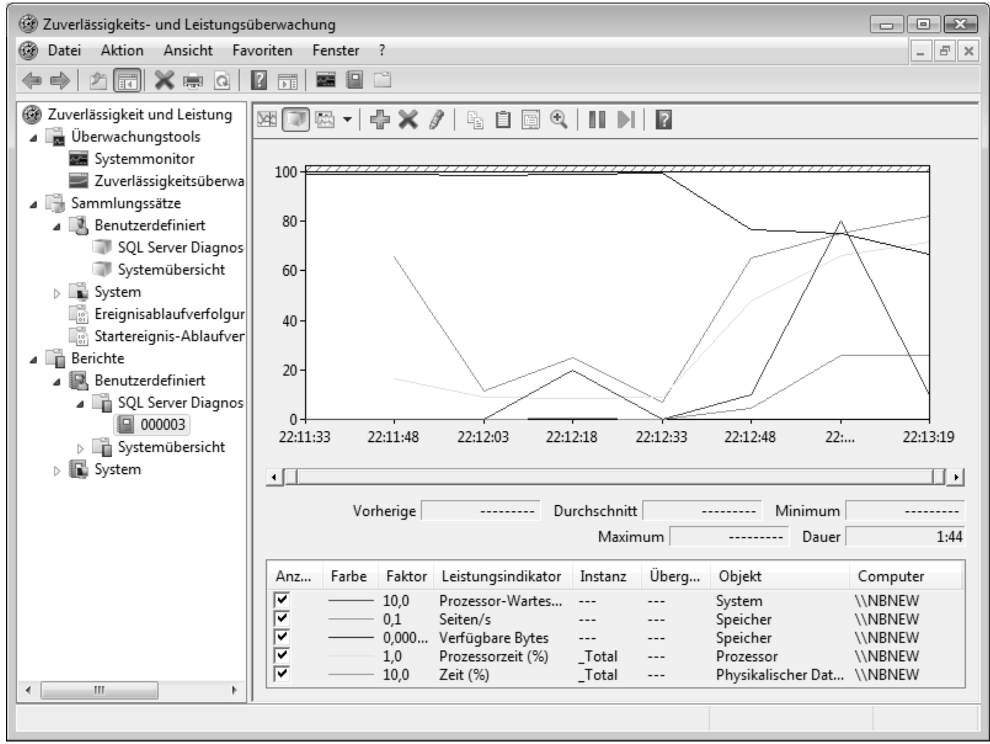


Abbildung 4.18: *Der erstellte Bericht als Liniendiagramm*

Den für jede Datensammlung automatisch erzeugten Standardbericht können Sie ebenfalls konfigurieren. Wenn Sie in den Eigenschaften die Option DATENVERWALTUNG UND BERICHTERSTELLUNG AKTIVIEREN auswählen, können Sie für diesen Bericht auch eine Übersichtsdarstellung auswählen. Hierbei handelt es sich um eine HTML-Datei, die auch sehr bequem zur Archivierung oder zum Versand des Berichts verwendet werden kann. Abbildung 4.19 zeigt ein Beispiel.

Die protokollierten Sammlungen werden in Dateien abgelegt, deren Speicherort und -format Sie bei der Konfiguration der Sammlung festgelegt haben. Diese Sammlungen sind mit dem Systemmonitor verknüpft und können daher einfach per Doppelklick geöffnet und ausgewertet werden. Außerdem existiert eine weitere Möglichkeit, die Protokolle auszuwerten. Hierzu kommen wir nun im nächsten Abschnitt.

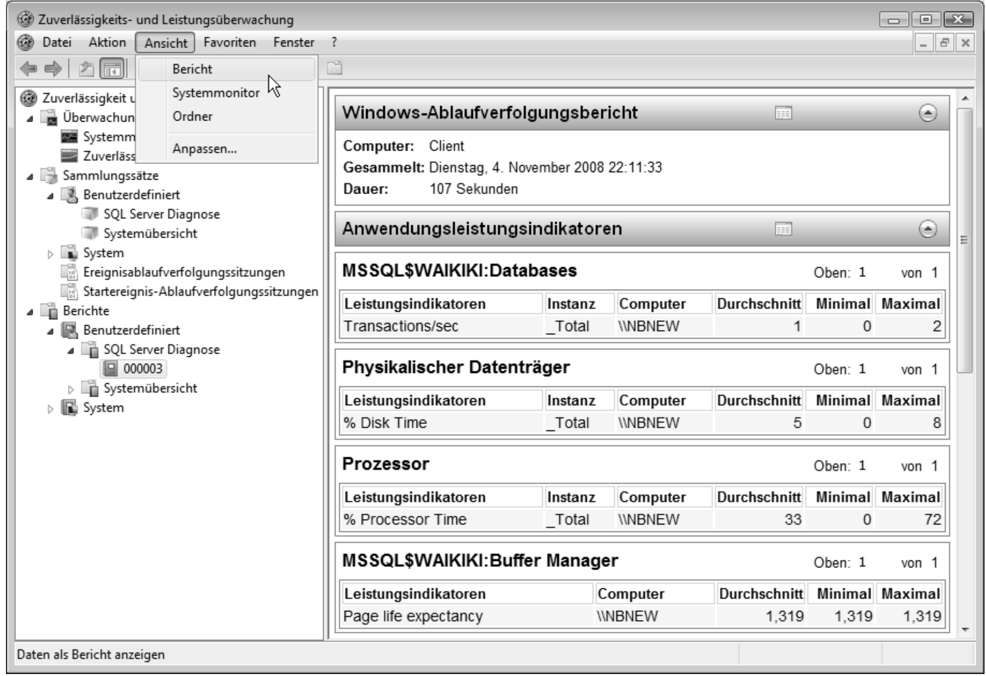


Abbildung 4.19: *Übersichtsdarstellung eines protokollierten Sammlungssatzes*

4.5 Verbindung von Systemmonitor-Berichten mit Ablaufverfolgungen

Seit der Version 2005 bietet der SQL Server Profiler die gleichermaßen faszinierende und doch nur wenig bekannte Möglichkeit, gleichzeitig eine gespeicherte Ablaufverfolgung und ein vom Systemmonitor gespeichertes Protokoll zu öffnen und anzuzeigen. Voraussetzung hierfür ist, dass beide Aufzeichnungen denselben Zeitraum umfassen und dass die SQL Server-Ablaufverfolgung zumindest die Ereignisspalte *StartTime* enthält.

Angenommen, Sie haben eine serverseitige Ablaufverfolgung erstellt und das Ergebnis liegt als Ablaufverfolgungsdatei vor. Öffnen Sie einfach diese Ablaufverfolgung durch einen Doppelklick im SQL Server Profiler. Falls Sie ebenfalls eine Systemmonitor-Datensammlung gespeichert haben, können Sie diese Datensammlung über DATEI • LEISTUNGSDATEN IMPORTIEREN... zur Anzeige hinzufügen. Beide Protokolle werden in einem Fenster angezeigt (Abbildung 4.20).

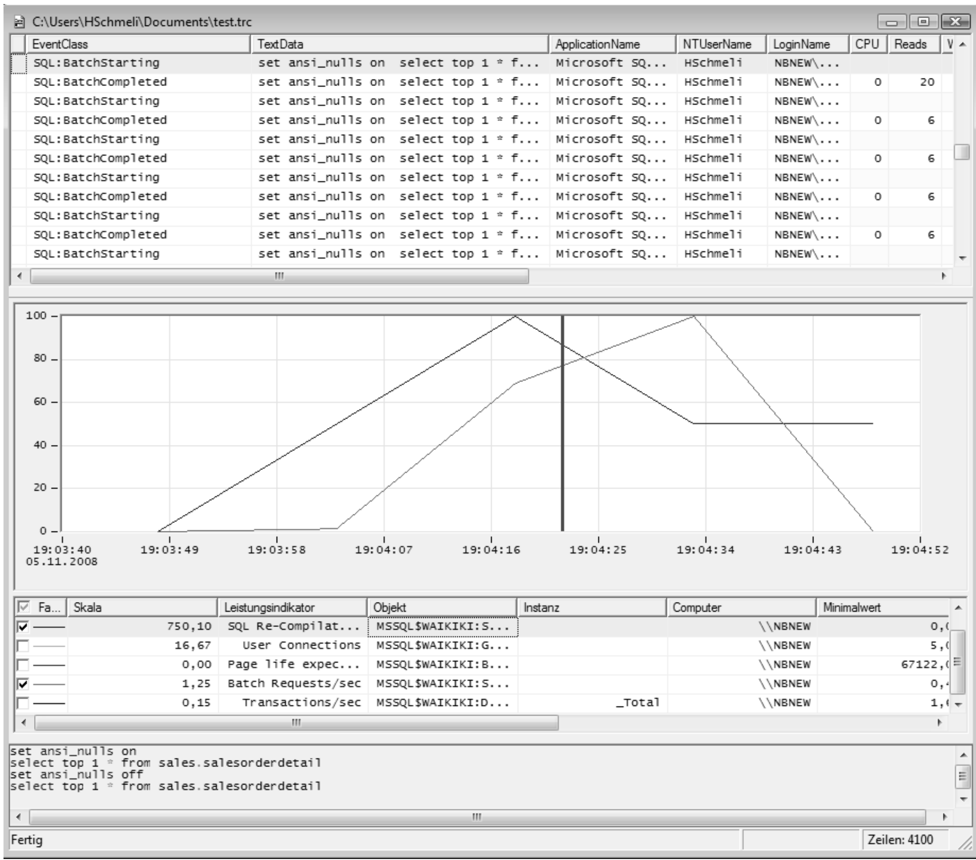


Abbildung 4.20: *Verbindung von Ablaufverfolgung und Systemmonitor-Sammlung*

Hierbei werden die beiden Protokolle gleichzeitig angezeigt. Wann immer Sie also in einem der beiden Protokolle einen Bereich anklicken, wird auch der entsprechende Zeitpunkt im anderen Protokoll ausgewählt.

4.6 Dynamische Verwaltungssichten

SQL Server protokolliert eine Vielzahl ausgeführter Aktionen. Die entsprechenden Protokolle sind dabei kein Geheimnis. Ab der Version 2005 gibt es für die Abfrage der protokollierten Ereignisse und Indikatoren dynamische Verwaltungssichten (Dynamic Management Views). Durch diese Systemsichten können Sie zum Beispiel Informationen über Abfrageausführungen mit zugehöriger Nutzung von Systemressourcen, aktuell laufende Transaktionen oder auch Blockierungen und Wartezustände erhalten. SQL Server ist zum Glück sehr »gesprächig«, wenn Sie ihn nach internen Abläufen »ausfragen« möchten. Es liegen tatsächlich fast alle Interna offen. Sie müssen lediglich wissen, welche dynamische Verwaltungssicht Sie wofür verwenden, und wie Sie die Inhalte interpretieren. Insgesamt verfügt

SQL Server über etwa 140 dynamische Verwaltungssichten, von denen einige in Wirklichkeit allerdings keine Sichten, sondern Funktionen sind.

An dieser Stelle werden Sie keine Auflistung aller Sichten finden. Ich möchte Ihnen hier zunächst nur einige der aus meiner Sicht wichtigsten vorstellen. Im weiteren Verlauf dieses Buches werden Ihnen dann zu gegebener Zeit weitere dynamische Verwaltungssichten begegnen, und zwar immer dann, wenn Sie im entsprechenden Kontext benötigt werden.

Bevor wir nun aber einige der Sichten und Funktionen betrachten, möchte ich noch Folgendes erwähnen:

1. Die dynamischen Verwaltungssichten heißen deshalb dynamisch, weil die zurückgelieferten Informationen sich ständig ändern. Sie bekommen also bei einer Abfrage immer die momentan gültigen Spaltenwerte. Hierbei ist es oftmals so, dass diese Werte kumulierte Daten seit dem letzten SQL Server-Start enthalten. Dies bedeutet, dass die entsprechenden Werte bei einem Neustart von SQL Server komplett zurückgesetzt werden. Die Kumulierung beginnt also bei jedem Start von SQL Server erneut.
2. SQL Server schützt sich auch vor sich selber. In diesem Zusammenhang bedeutet dies, dass darauf geachtet wird, die Systemlast nicht durch die Protokollierung der über die dynamischen Verwaltungssichten zugänglichen Werte übermäßig zu vergrößern. Die Protokollierung benötigt natürlich ebenfalls Systemressourcen, beispielsweise Hauptspeicher. Falls die Protokollierung die Ressourcen zu sehr belastet, kann es zum Beispiel auch passieren, dass ältere Einträge aus Protokollen entfernt werden. Dadurch kann es vorkommen, dass die verfügbare Historie nicht in jedem Fall bis zum letzten SQL Server-Start zurückreicht.
3. Falls Sie mehr über dynamische Verwaltungssichten erfahren möchten, so empfehle ich Ihnen, dass Sie einfach mit dem Profiler eine Ablaufverfolgung starten und ein paar integrierte Berichte für Systeminformationen generieren. (Mit Berichten befassen wir uns etwas weiter unten.) In der Ablaufverfolgung können Sie dann sehr schön verfolgen, welche dynamischen Systemsichten verwendet werden, um die Berichtsdaten abzufragen. Diese Vorgehensweise ist sehr einfach und doch effizient. Aus meiner Erfahrung kann ich Ihnen versichern, dass Sie sich als Administrator oder Entwickler über kurz oder lang mit dynamischen Verwaltungssichten auseinandersetzen müssen. Dann kann das »Ausspionieren« der existierenden Berichte tatsächlich eine wertvolle Hilfe sein.

Die Namensgebung der dynamischen Verwaltungssichten hat System, das heißt, sie folgt einem bestimmten Muster. Alle dynamischen Verwaltungssichten sind im Schema `sys` abgelegt – und zwar in jeder existierenden Datenbank. Rein physikalisch existieren die Sichten natürlich nur einmal. Sie können jedoch aus jeder Datenbank auf diese Sichten zugreifen. Der Name für jede dynamische Verwaltungssicht beginnt mit den Zeichen `dm_`, was für »dynamic management« steht. Nach dem Unterstrich folgt ein Kürzel für den Bereich, über den die Sicht Informationen liefert. So steht zum Beispiel das Kürzel `db` für Datenbank, `os` für Betriebssystem, `exec` für die Ausführungsumgebung bzw. Abfrageausführung, `tran` für Transaktionen und `broker` für Sichten, die Informationen zum SQL Server Service Broker liefern. Dann folgt der eigentliche Name der Sicht, der beschreibt, welche Daten die entsprechende Sicht zurückliefert.

Sie werden im weiteren Verlauf dieses Buches – und auch bereits in diesem Kapitel – viele der wichtigsten Verwaltungssichten kennenlernen. Ich möchte Ihnen aber bereits an dieser Stelle einige elementare Sichten vorstellen, die wir immer wieder benötigen werden, und Ihnen zeigen, welche Informationen Sie mit diesen Sichten erhalten können. Dabei konzentrieren wir uns zunächst auf die drei Bereiche Wartezustände, E/A-Vorgänge und aktuelle Aktivität.

4.6.1 Abfrage der aktuellen Aktivität

Die dynamische Verwaltungssicht `sys.dm_exec_requests` liefert Informationen über die gerade laufenden Prozesse. Die Sicht enthält auch eine Spalte mit einem `sql_handle`, die verwendet werden kann, um den Text der gerade laufenden Abfragen zurückzuliefern. Die Umwandlung eines `sql_handle` in den zugehörigen Text erledigt die dynamische Verwaltungsfunktion `sys.dm_exec_sql_text`. Wenn Sie die beiden Sichten bzw. die Sicht und die Funktion verbinden, erhalten Sie die folgende Abfrage:

```
select t.text,start_time,status
      ,command,wait_type,wait_time,last_wait_type,wait_resource
      ,percent_complete,estimated_completion_time,cpu_time
      ,total_elapsed_time,scheduler_id
      ,task_address,reads,writes,logical_reads
  from sys.dm_exec_requests as r
       cross apply sys.dm_exec_sql_text(r.sql_handle) as t
 -- Nur für eine Session
 where session_id=65
```

Das Beispiel zeigt auch, wie Sie die Daten nur für eine bestimmte Session (65) abfragen. Möchten Sie die Daten für alle aktuell laufenden Abfragen sehen, dann lassen Sie die `WHERE`-Bedingung einfach weg.

Die Abfrage gibt Ihnen nur eine Momentaufnahme zurück und nicht etwa eine Historie der in der letzten Zeit ausgeführten Aktionen. Es werden also nur die gerade aktuellen Abfragen berücksichtigt.

4.6.2 Abfrage der E/A-Vorgänge

Für das Aufspüren von Performance-Problemen ist es oftmals hilfreich, zunächst die E/A-Operationen zu untersuchen und festzustellen, welche Datenbanken die meiste E/A-Last erzeugen. Noch einmal zur Erinnerung: Physikalische E/A-Operationen sind in vielen Fällen der Flaschenhals, da Lese- und Schreibvorgänge von bzw. auf die Festplatten um Größenordnungen langsamer sind als beispielsweise das Lesen oder Schreiben direkt im Speicher. Die dynamische Verwaltungsfunktion `sys.dm_os_virtual_file_stats` liefert Informationen über den Datendurchsatz. Die Funktion erwartet zwei Parameter: Der erste Parameter spezifiziert die *DatenbankId*, der zweite Parameter die *DateiId*, für die Sie die Information abfragen möchten. Beide Parameter dürfen auch `NULL` sein; in diesem Fall erhalten Sie die E/A-Vorgänge für alle Dateien in sämtlichen Datenbanken. Ein kleines Problem dabei ist, dass die Funktion für Dateien und Datenbanken nur die Ids und nicht die Namen zurückliefert. Es ist aber leicht möglich, die Systemsicht `sys.master_files` mit

der Funktion `sys.dm_os_virtual_file_stats` so zu verbinden, dass Sie genaue Informationen zum Datendurchsatz je Datenbank und Datei erhalten. Eine entsprechende Abfrage sähe wie folgt aus:

```
select db_name(vfs.database_id)                   as Datenbank
      ,case
         when mf.type = 1
           then 'Protokoll'
         else 'Daten'
       end                                        as Typ
      ,cast(vfs.num_of_bytes_read
           / 1024.0/1024.0 as decimal(10,2))      as [Gelesen/MB]
      ,cast(vfs.num_of_bytes_written
           / 1024.0/1024.0 as decimal(10,2))      as [Geschrieben/MB]
      ,cast((vfs.num_of_bytes_read
             + vfs.num_of_bytes_written)
           / 1024.0/1024.0 as decimal(10,2))      as [EA Summe/MB]
      ,cast(vfs.io_stall/1000.0 as decimal(10,2)) as [Wartezeit (sek)]
      ,row_number() over(order by io_stall desc)  as Rang
  from sys.dm_io_virtual_file_stats(null,null) vfs
       join sys.master_files mf
         on mf.database_id = vfs.database_id
        and mf.file_id = vfs.file_id
```

In Abbildung 4.21 sehen Sie ein beispielhaftes Ergebnis der obigen Abfrage. In den letzten beiden Spalten finden Sie die Gesamtwartezeit für E/A-Operationen sowie die Rangfolge nach eben dieser Wartezeit.

Ergebnisse | Meldungen

	Datenbank	Typ	Gelesen/MB	GeSchrieben/MB	EA Summe/MB	Wartezeit (sek)	Rang
1	QueryTest	Protokoll	1.43	1.43	1.45	18.06	1
2	PerfDWH	Protokoll	1.19	1.19	1.20	17.19	2
3	AdventureWorks2008	Daten	13.54	13.54	13.55	13.71	3
4	msdb	Protokoll	0.37	0.37	0.44	7.06	4
5	AdventureWorks2008	Protokoll	0.44	0.44	0.46	4.66	5

Abbildung 4.21: *E/A-Operationen je Datenbank und Datei*

Hier erhalten Sie stets die kumulierten Werte seit dem letzten Start von SQL Server. Es ist also zum Beispiel nicht ohne Weiteres möglich, die Werte für die letzten zwei Stunden zu ermitteln. Hierzu müssten Sie zu definierten Zeitpunkten die Ergebnisse der Abfrage protokollieren und dann die Differenz zwischen zwei Zeitpunkten berechnen. In Kapitel 11 finden Sie ein Beispiel für eine derartige Verfahrensweise.

4.6.3 Abfrage der insgesamt aufgetretenen Wartezustände

Es ist eine weitverbreitete Erkenntnis, dass sich die Zeit, die eine Anforderung für die Abarbeitung insgesamt benötigt, so zusammensetzt:

Gesamtzeit = Arbeitszeit + Wartezeit

Wie auch im richtigen Leben gibt es also einen Zeitanteil, in dem tatsächlich gearbeitet wird, und einen Zeitanteil, in dem darauf gewartet wird, dass die eigentliche Arbeit fortgesetzt werden kann. Die Wartezeit resultiert hierbei im Wesentlichen daraus, dass gemeinsam genutzte Ressourcen nicht alle Anforderungen gleichzeitig bearbeiten können. Solche Ressourcen können Systemressourcen wie zum Beispiel CPU-Zeit oder auch Hauptspeicher oder E/A-Kanäle für physikalische Lese- und Schreiboperationen sein. Darüber hinaus entstehen durch das Sperren auf Datenbereichen Blockierungen (also Wartezustände), wenn inkompatible Sperren angefordert werden. In diesem Fall kann eine Sperre erst dann erteilt werden, sobald die bereits von einem anderen Prozess gesperrten Daten wieder freigegeben werden.

Es ist möglich, die aufgetretenen Wartezeiten abzufragen, und zwar unterteilt nach ihrer Kategorie. Hierzu bietet SQL Server die dynamische Verwaltungssicht `sys.dm_os_wait_stats`, die Ihnen bereits in Kapitel 2 begegnet ist. Diese Sicht gibt alle seit dem letzten Start von SQL Server aufgetretenen Wartezustände zurück, enthält also wiederum nur die kumulierten Werte. Beachten Sie bitte, dass die Sicht nur bereits abgeschlossene Wartevorgänge berücksichtigt; gerade aktuell bestehende Wartezustände werden nicht angezeigt.

Insgesamt gibt es immerhin fast 500 unterschiedliche Wartezustände, die durch die Sicht zurückgegeben werden. Eine mehr oder weniger genaue Erklärung finden Sie in der Online-Dokumentation. Wir werden es auch hier wieder so handhaben, dass an dieser Stelle keine Auflistung der möglichen Wartezustände gegeben wird. Stattdessen werden Sie in den weiteren Kapiteln einige besonders wichtige Wartezustände kennenlernen, sofern dies für die dort jeweils präsentierte Thematik erforderlich und sinnvoll ist.

Einige Wartezustände betreffen SQL Server-interne Prozesse, die per Definition warten, wie zum Beispiel der *Lazy Writer*-Prozess, der geänderte Datenseiten asynchron auf die Festplatte schreibt. Der Prozess *Lazy Writer* ist die meiste Zeit mit Warten beschäftigt. Deshalb können Sie diesen Wartezustand getrost außer Acht lassen. Dies trifft auch auf eine Reihe anderer Wartezustände zu, bei denen es oft so ist, dass hohe Wartezeiten gewünscht sind, und die daher kein Problem darstellen. Hierzu zählt zum Beispiel der Zustand `LOGMGR_QUEUE`, der aussagt, dass die Warteschlange für das Schreiben des Protokolls auf Arbeit wartet. Die Dokumentation enthält für jeden Wartezustand auch die Information, ob das Auftreten von Wartezeiten dieser Kategorie problematisch ist oder nicht. Leider kann die in der Dokumentation enthaltene Auflistung der Kategorien aber nicht nach dieser Information sortiert oder gefiltert werden, was das Herausfinden der tatsächlich relevanten Zustände nicht gerade erleichtert.

Die folgende Abfrage ermittelt die seit dem letzten Start von SQL Server aufgetretenen Wartezustände und -zeiten, wobei die nicht wesentlichen Zustände herausgefiltert wurden:

```
with WaitStats as
 (
  select wait_type, waiting_tasks_count
        ,wait_time_ms, max_wait_time_ms
        ,signal_wait_time_ms
        ,sum(wait_time_ms) over() as WaitTimeTotal
    from sys.dm_os_wait_stats
```

In der Anweisung wird noch ein wenig gerechnet, um für jeden Wartezustand den prozentualen Anteil an der Gesamtwartezeit zu ermitteln.

Wenn Sie die Abfrage ausführen, wird auch eine Spalte mit Namen `signal_wait_time_ms` zurückgeliefert, die einer kurzen Erklärung bedarf: Dies ist die Zeit, die vergeht, bevor der entsprechende Thread nach seiner Aktivierung tatsächlich mit der Arbeit beginnt. Es kann durchaus vorkommen, dass diese Zeitspanne relevant ist. Immerhin ist es vorstellbar, dass ein Prozess prinzipiell starten könnte, weil alle angeforderten Ressourcen verfügbar sind. In diesem Fall wird also der Start initialisiert. Bevor der Prozess aber tatsächlich startet, vergeht noch eine gewisse Zeitspanne. Dieses Intervall wird im Wesentlichen durch die Auslastung der vorhandenen Prozessoren bestimmt. Aus dem Wert für `signal_wait_time_ms` können Sie daher in gewisser Weise auf Ihre CPU-Auslastung schließen. Wenn der Zeitanteil von `signal_wait_time_ms` größer als 25 Prozent des Wertes für `wait_time_ms` ist, ist dies ein Indiz dafür, dass Ihre CPUs die eingehenden Anforderungen nicht schnell genug verarbeiten können. Die Spalte `wait_time_ms` enthält übrigens auch den Anteil für `signal_wait_time_ms`.

4.6.4 Abfrage der SQL Server-Leistungsindikatoren

Die Werte aller SQL Server-Leistungsindikatoren können durch die dynamische Verwaltungssicht `sys.dm_os_performance_counters` abgefragt werden. In Abbildung 4.22 sehen Sie einen Auszug des Ergebnisses.

object_name	counter_name	instance_name
MSSQL$WAIKIKI:Plan Cache	Cache Hit Ratio Base	SQL Plans
MSSQL$WAIKIKI:Plan Cache	Cache Pages	SQL Plans
MSSQL$WAIKIKI:Plan Cache	Cache Object Counts	SQL Plans
MSSQL$WAIKIKI:Plan Cache	Cache Objects in use	SQL Plans
MSSQL$WAIKIKI:Plan Cache	Cache Hit Ratio	Object Plans
MSSQL$WAIKIKI:Plan Cache	Cache Hit Ratio Base	Object Plans
MSSQL$WAIKIKI:Plan Cache	Cache Pages	Object Plans
MSSQL$WAIKIKI:Plan Cache	Cache Object Counts	Object Plans
MSSQL$WAIKIKI:Plan Cache	Cache Objects in use	Object Plans
MSSQL$WAIKIKI:Plan Cache	Cache Hit Ratio	_Total
MSSQL$WAIKIKI:Plan Cache	Cache Hit Ratio Base	_Total
MSSQL$WAIKIKI:Plan Cache	Cache Pages	_Total
MSSQL$WAIKIKI:Plan Cache	Cache Object Counts	_Total
MSSQL$WAIKIKI:Plan Cache	Cache Objects in use	_Total

Abbildung 4.22: *Auszug aus der Rückgabe von »sys.dm_os_performance_counters«*

Für jede SQL Server-Instanz erhalten Sie etwa 820 Zeilen und dann noch einmal 36 je Datenbank, die Auskunft über die seit dem letzten Start von SQL Server kumulierten Werte der SQL Server-Leistungsindikatoren geben. Es ist also auch hier nicht ohne Weiteres möglich, Änderungen zwischen zwei Zeitpunkten zu beobachten. Falls Sie diese Information benötigen, müssen Sie die Ergebnisse der Sicht zu beiden Zeitpunkten speichern und anschließend voneinander subtrahieren. Ich möchte Sie an dieser Stelle noch einmal darauf hinweisen, dass Sie hierzu mehr in Kapitel 11 erfahren.

4.7 Statistische Systemfunktionen

SQL Server bietet eine Hand voll statistischer Systemfunktionen an, die für eine Performance-Analyse hilfreich sind. Diese statistischen Systemfunktionen sind insbesondere dann nützlich, wenn es gilt, schnell sich einen Überblick über den Zustand der laufenden SQL Server-Instanz zu verschaffen. Die Anzahl dieser Funktionen ist bei Weitem nicht so groß wie die der dynamischen Verwaltungssichten. Außerdem sind die Namen nicht ganz so komplex, wie dies bei den dynamischen Verwaltungssichten der Fall ist. Aus diesen beiden Gründen ist es leichter, sich die Namen dieser Funktionen zu merken. Außerdem hat man sie dadurch im Falle eines Falles schnell zur Hand. Sie finden eine Übersicht dieser Funktionen im Objekt-Explorer, dort unter jeder Datenbank im Ordner *Programmierbarkeit\Funktionen\Systemfunktionen\Statistische Systemfunktionen*. Für eine Analyse der Auslastung Ihrer SQL Server-Instanz sind hierbei insbesondere die in der folgenden Auflistung beschriebenen Funktionen interessant. Alle aufgeführten Funktionen geben jeweils nur einen einzigen Wert zurück. Dieser Wert enthält die kumulierte Information seit dem letzten Start der SQL Server-Instanz (und natürlich für diese Instanz).

- `@@Connections`. Dieser Wert enthält die Anzahl der Verbindungsversuche zum SQL Server. Hierbei ist es unerheblich, ob ein Verbindungsversuch erfolgreich war oder nicht – es zählt alleine der Versuch.
- `@@Cpu_Busy`. Gibt die Zeit zurück, die alle CPUs zusammen für SQL Server aufgewendet haben. Das Ergebnis wird in CPU-Zeitinkrementen (Timer Ticks) zurückgegeben (siehe unten).
- `@@Idle`. Während dieser Zeit hat die SQL Server-Instanz sich im Leerlauf befunden. Dieser Wert wird ebenfalls für alle CPUs kumuliert. Die Zeiteinheit ist auch hier Timer Ticks.
- `@@IO_Busy`. Dies ist die Zeit, die SQL Server mit E/A-Operationen verbracht hat, ebenfalls wieder in Timer Ticks.
- `@@Pack_Received`. Dies ist die Anzahl der Pakete, die SQL Server über das Netzwerk empfangen hat.
- `@@Pack_Sent`. Dieser Wert steht für die Anzahl Pakete, die SQL Server über das Netzwerk versandt hat.
- `@@Timeticks`. Dieser Wert gibt die in einem CPU-Zeitinkrement (Timer Tick) enthaltene Anzahl Mikrosekunden zurück. Der Wert ist systemabhängig und wird zur Umwandlung von CPU-Zeitinkrementen in eine Zeiteinheit benötigt (siehe unten).
- `@@Total_Read`. Der Rückgabewert dieser Funktion enthält die Anzahl der physikalischen Lesevorgänge. Hier wird nur die Anzahl der Leseoperationen gezählt; eine Angabe zur gelesenen Datenmengen erhalten Sie nicht.
- `@@Total_Write`. Die Funktion gibt Auskunft über die physikalischen Schreibvorgänge. Ebenso wie bei `@@Total_Read` erhalten Sie auch hier nur die Anzahl der Schreibvorgänge und nicht die Angabe zur tatsächlich geschriebenen Datenmenge.

Alle aufgelisteten Funktionen geben einen Wert vom Typ `INTEGER` zurück. Da stets die seit dem letzen SQL Server-Start kumulierten Werte ermittelt werden, kann es daher vorkommen, dass ein arithmetischer Überlauf auftritt, und Sie Werte erhalten, die nicht korrekt sind. Dies gilt insbesondere für die Funktionen `@@Cpu_Busy` und `@@IO_Busy`, die ja in einer sehr kleinen Zeiteinheit angegeben werden und daher sehr große Werte aufweisen können. Das Maximum für diese beiden Werte beträgt deshalb jeweils ca. 49 Tage. Wenn der Wert darüber hinaus anwächst, erhalten Sie beim Aufruf der Funktion eine Warnung. Die Funktionswerte sind in diesem Fall nicht korrekt.

Die Funktionen, die eine Zeit zurückgeben, verwenden CPU-Zeitinkremente als Zeiteinheit. Zur Umrechnung dieser Zeitinkremente in Sekunden muss die Funktion `@@Timeticks` verwendet werden, die angibt, wie viele Mikrosekunden ein Zeitinkrement beinhaltet. Für eine Umrechnung von Zeitinkrementen in Sekunden multiplizieren Sie den von der Funktion erhaltenen Wert mit dem Wert von `@@Timeticks` und teilen ihn dann durch 1.000.000. Also zum Beispiel so:

```
select 0.000001 * @@io_busy * @@timeticks as [EA in Sekunden]
```

Heraus kommt die Anzahl Sekunden, die Ihre SQL Server-Instanz seit dem letzten Start mit E/A- Operationen verbracht hat.

Oftmals ist der absolute Wert der statistischen Systemfunktionen alleine nicht aussagekräftig genug. In vielen Fällen ist zum Beispiel die Information über die durchschnittlichen E/A-Operationen je Sekunde interessanter als die Information, wie viele E/A-Operationen insgesamt ausgeführt wurden. Für Aussagen dieser Art muss die Laufzeit der SQL Server-Instanz bestimmt werden. Wir können dann zum Beispiel den Wert von `@@Total_Read` einfach durch die Gesamtlaufzeit der Instanz teilen, um herauszufinden, wie viele Lese-Operationen bislang im Schnitt je Sekunde ausgeführt wurden.

Das Ermitteln der Laufzeit einer SQL Server-Instanz ist leider nur über einen kleinen Umweg möglich. Es gibt hierfür keine direkte Systemfunktion; allerdings helfen die Systemprozesse, die bei jedem Start von SQL Server automatisch gestartet werden, hier weiter. In der dynamischen Verwaltungssicht `sys.dm_exec_sessions` finden Sie für alle Verbindungen auch eine Log-in-Zeit. Für Systemprozesse (dies sind diejenigen Prozesse mit einer *spid* kleiner als 51), ist diese Zeit ziemlich genau mit der Startzeit der SQL Server-Instanz identisch.

Das folgende Skript ermittelt relative Werte für E/A-Operationen, bezogen auf die Gesamtlaufzeit der SQL Server Instanz, sowie den prozentualen Zeitanteil, für die Nutzung der CPUs und des E/A-Systems.

```
-- Bestimme die Anzahl von Sekunden seit dem Start von SQL Server
-- Wir verwenden den Anmeldezeitpunkt des Prozesses mit der spid 1
declare @sekundenSeitStart float
select @sekundenSeitStart = datediff(s, login_time, current_timestamp)
  from sys.dm_exec_sessions
```

4.8 Gespeicherte Systemprozeduren

SQL Server bietet insgesamt etwa 1.350 gespeicherte Systemprozeduren an, die sowohl zur Konfiguration als auch zur Abfrage von SQL Server-Zuständen verwendet werden können. Natürlich können an dieser Stelle nicht alle vorhandenen Prozeduren aufgezählt und erläutert werden. Ich möchte Sie also auch hier wieder einmal auf die Online-Dokumentation verweisen. Viele der Prozeduren werden Sie wahrscheinlich nie oder nur sehr selten benötigen, weil die entsprechenden Aktionen über die grafische Oberfläche des SQL Server Management Studios sehr viel bequemer erledigt werden können. – Wobei im Hintergrund oftmals gespeicherte Systemprozeduren verwendet werden, um die angeforderten Aktionen auszuführen, was Sie mit dem Profiler übrigens auch sehr schön beobachten können.

Etwas weiter oben haben Sie bereits gespeicherte Systemprozeduren zur Verwaltung von SQL Server-Ablaufverfolgungen kennengelernt. Auf einige weitere Prozeduren werden wir im Verlauf des Buches noch näher eingehen. Das sind natürlich genau diejenigen Prozeduren, die für eine Performance-Analyse relevant sind. An dieser Stelle möchte ich Ihnen zunächst nur eine Handvoll weiterer nützlicher Prozeduren vorstellen.

- **`sp_configure`**. Mit dieser Prozedur können Sie die aktuellen Konfigurationsparameter Ihrer SQL Server-Instanz abfragen oder auch verändern. Da die Werte dieser Parameter unter Umständen ganz erhebliche Auswirkung auf die Ausführung von Abfragen und damit auch auf die Anfrageleistung haben, ist es von Bedeutung, dass Sie bei auftretenden Problemen die aktuellen Parameterwerte analysieren. Für die Abfrage der aktuellen Parameter können Sie auch die Systemsicht `sys.configurations` oder das weiter unten im Abschnitt 4.13 gezeigte Server Dashboard verwenden. Das Ändern dieser Werte funktioniert nur über den Aufruf der Prozedur `sp_configure` oder den Eigenschaften-Dialog der SQL Server-Instanz im Objekt Explorer des Management Studios.
- **`sp_lock`**. Wie Sie sicherlich bereits anhand des Namens der Prozedur vermuten, liefert `sp_lock` Informationen über aktuell existierende Blockierungen. Hierbei werden für jede existierende Verbindung unter anderem der Typ der aktiven Sperre, der Sperr-Modus sowie der Status der Sperre zurückgegeben. Ein Sperr-Typ ist hierbei zum Beispiel *PAG* für eine Seitensperre, *TAB* für die Sperre einer Tabelle oder *DB* für eine Sperre der gesamten Datenbank. Der Sperr-Modus kann zum Beispiel *X* für eine exklusive Sperre oder *S* für eine gemeinsame Sperre sein. In der Spalte *Status* sehen Sie, ob die Sperre bereits erteilt wurde (*GRANT*), oder ob auf die Erteilung der Sperre gewartet wird (*WAIT*). Die Prozedur kann auch ein oder zwei Prozess-IDs als Parameter verarbeiten. In diesem Fall werden nur Informationen über Sperren dieser Prozesse ausgegeben. `sp_lock` ermöglicht einen einfachen und schnellen Überblick über existierende Blockierungen und ist daher gut geeignet, um eine Analyse zu starten, falls Sie Blockierungsprobleme vermuten.
- **`sp_monitor`**. Diese Prozedur ruft die in Abschnitt 4.7 behandelten statistischen Systemfunktionen auf und gibt die ermittelten Werte in vier Ergebnissen zurück. Die Prozedur protokolliert hierbei stets die gerade (also beim Aufruf) aktuellen Rückgabewerte der Funktionen in der Tabelle `master.dbo.spt_monitor`. Ein Aufruf von `sp_monitor` ermittelt vor dem Speichern der aktuellen Werte zunächst die Differenz der aktuellen Werte zu den bereits gespeicherten Werten und gibt dann die Differenzwerte zurück. In Abbildung 4.23 sehen Sie ein Beispiel für die Rückgabe der vier Ergebnisse von `sp_`

monitor. Sie werden im Ergebnis der Ausgabe sicherlich sofort die in Abschnitt 4.7 erläuterten statistischen Systemfunktionen wiedererkennen.

last_run	current_run	seconds
2008-11-06 21:57:40.583	2008-11-19 20:29:00.030	1117880

cpu_busy	io_busy	idle
14(7)-0%	2(1)-0%	343167(317754)-28%

packets_received	packets_sent	packet_errors
3010(2280)	4164(3052)	0(0)

total_read	total_write	total_errors	connections
2631(823)	662(499)	0(0)	6306(3686)

Abbildung 4.23:
Rückgabe von »sp_monitor«

Die Werte in Klammern geben dabei jeweils die Veränderung zwischen zwei Aufrufen von sp_monitor an. Die Prozentangaben in den Spalten cpu_busy, io_busy und idle beziehen sich auf den prozentualen Anteil des Wertes seit der letzten Ausführung von sp_monitor. So bedeutet zum Beispiel die Angabe 343167(317754)-28% für idle, dass SQL Server insgesamt 343167 Timer Ticks unbeschäftigt war, davon 317754 allein seit dem letzten Aufruf von sp_monitor. Dies entspricht 28 % der Gesamtzeit seit dem letzten Aufruf von sp_monitor.

sp_monitor ist recht gut geeignet, wenn Sie einen schnellen Eindruck davon bekommen möchten, wie sehr Ihre SQL Server-Instanz momentan ausgelastet ist. Rufen Sie die Prozedur einfach zweimal im Abstand von 20 Sekunden auf und Sie können so ungefähr abschätzen, was gerade insgesamt abläuft. Genau hierfür ist sp_monitor gut geeignet; für weiterführende Analysen müssen Sie sicherlich andere Möglichkeiten in Betracht ziehen. Was Sie auch bedenken sollten, ist der Umstand, dass sp_monitor nur die Differenz zum letzten Aufruf berechnet. Hierbei ist es unerheblich, von welcher Verbindung dieser Aufruf abgesetzt wurde. Wenn also gerade von mehreren Verbindungen aus mit sp_monitor gearbeitet wird, dann kann dies einige Verwirrung erzeugen.

- **sp_spaceused**. Diese Prozedur liefert Informationen über den verwendeten Speicherplatz einer Datenbank, einer Tabelle, einer indizierten Sicht, einer Service Broker-Warteschlange oder eines Index zurück. Dadurch erhalten Sie einen schnellen Überblick über die Speicherverwendung bestimmter Datenbankobjekte.
- **sp_who**. Diese Prozedur liefert Informationen zu gerade aktiven Verbindungen. Hierbei sehen Sie zum Beispiel, welche Blockierungen gerade aktiv sind, welche Wartezustände momentan existieren und welcher Abfragetyp gerade von welcher Verbindung ausgeführt wird. Möglich ist hierbei sowohl eine Ausgabe aller aktuellen Verbindungen als auch eine Auswahl nach Anmeldenamen oder Sitzungs-IDs.
- **sp_who2**. Diese Prozedur ist undokumentiert. Wie der Name bereits vermuten lässt, liefert die Prozedur, ähnlich wie sp_who, Informationen über die Aktivitäten der momentan offenen Verbindungen. Zusätzlich zu den von sp_who zurückgelieferten Spalten erhalten Sie durch sp_who2 beispielsweise auch noch Informationen über die verwendete CPU-Zeit, E/A-Operationen und die letzte Ausführung einer Abfrage je Verbindung. Diese Informationen sind in vielen Fällen sehr hilfreich, wenn Sie aktuellen Problemen auf den Grund gehen. sp_who2 bietet tatsächlich eine sehr bequeme Möglichkeit, schnell und einfach die wichtigsten Informationen für jede offene Verbin-

dung abzufragen. Bitte bedenken Sie aber, dass die gespeicherte Prozedur `sp_who2` undokumentiert ist. Daher kann sich das Verhalten der Prozedur natürlich jederzeit und ohne Vorankündigung ändern. Betrachten Sie bitte die oben angegebenen Informationen unter diesem Aspekt und wundern Sie sich bitte nicht, wenn zum Beispiel Spalten wegfallen oder die gesamte Prozedur nicht mehr verfügbar ist.

4.9 DBCC

Der **D**ata**B**ase **C**onsistency **C**hecker wurde – betrachtet man den Namen – ganz offensichtlich ursprünglich entwickelt, um die Konsistenz von Datenbanken zu überprüfen. Das kann `DBCC` natürlich auch heute noch (`DBCC CHECKDB`). Darüber hinaus wurde `DBCC` aber erweitert und hat sich so nach und nach zu einem Universalwerkzeug entwickelt. Mittlerweile kann `DBCC` eine Menge mehr als Konsistenzprüfungen, unter anderem auch das Abfragen von SQL Server-Parametern und -Daten. Zu diesen Daten gehören auch solche, die Auskunft über Einstellungen und Zustände geben, welche für eine Suche nach Performance-Problemen interessant sind. `DBCC` wird Ihnen im Verlaufe dieses Buches noch an vielen Stellen begegnen, wenn solche Einstellungen abgefragt werden. An dieser Stelle möchte ich Ihnen zunächst zwei DBCC Kommandos vorstellen, die für eine Anfrageanalyse sehr wichtig sind:

- **`DBCC DROPCLEANBUFFERS`**. Dieses Kommando löscht den gesamten SQL Server-Datencache. Alle nachfolgenden Lesevorgänge müssen die Daten also zunächst von der Festplatte lesen.
- **`DBCC FREEPROCCACHE`**. Mit diesem Kommando löschen Sie den Plan Cache von SQL Server. Dadurch muss der Optimierer für alle nachfolgenden Abfragen einen entsprechenden Plan erstellen, da kein geeigneter Plan im Plan Cache vorhanden ist.

Beide Kommandos benötigen Sie für Vergleichsmessungen. Wenn Sie zum Beispiel eine bestimmte Abfrage tunen und mehrfach ausführen, wobei Sie jeweils die Ausführungszeit messen, werden die von der Abfrage benötigten Daten bei der ersten Ausführung in den Datencache übertragen, sofern sie dort noch nicht existieren. Alle nachfolgenden Ausführungen der Abfrage können dann die Daten aus dem Cache lesen und sind dadurch erheblich schneller. Wenn Sie also Vergleichsmessungen vornehmen, müssen Sie dafür sorgen, dass die Randbedingungen für alle Messungen weitestgehend identisch sind. Hierzu können Sie entweder einen »warmen« oder aber »kalten« Cache verwenden. Über `DBCC DROPCLEANBUFFERS` leeren Sie den Datencache und stellen somit sicher, dass bei jeder Messung physische Leseoperationen ausgeführt werden.

Bitte setzen Sie `DBCC DROPCLEANBUFFERS` und `DBCC FREEPROCCACHE` in Produktivumgebungen mit sehr viel Vorsicht ein. Das Leeren des Pufferspeichers hat erhebliche negative Konsequenzen auf die Gesamt-Performance. Bedenken Sie auch, dass beide Kommandos stets die Caches für den *gesamten SQL Server* leeren.

4.10 SQLdiag

SQLdiag ist ein Kommandozeilenprogramm zur Analyse und Protokollierung von SQL Server- und Windows-Leistungsdaten. Das Programm kann sowohl als einfache Konsolenanwendung als auch als Windows-Dienst ausgeführt werden. SQLdiag ist vor allem für den Microsoft-Support eine große Hilfe bei der Fehleranalyse. Falls Sie jemals in die Verlegenheit kommen sollten, mit Hilfe des Microsoft-Supports ein spezielles Problem zu bearbeiten, dann ist es sehr wahrscheinlich, dass Sie gebeten werden, die von SQLdiag erstellten Protokolle zur Verfügung zu stellen. Aber SQLdiag kann auch bei der ganz normalen Überwachung von SQL Server sehr nützlich sein. Es ist also weit mehr als nur ein Werkzeug zur Fehlersuche.

Über eine Reihe von Kommandozeilenparametern steuern Sie das Verhalten des Programms. Insbesondere geben Sie über die Kommandozeile eine Konfigurationsdatei an, in der festgelegt wird, welche SQL Server-Instanz überwacht werden und welche Leistungsparameter in das Ausgabeprotokoll eingeschlossen werden sollen. Die zur Verfügung stehenden Leistungsparameter lassen sich dabei in die folgenden Kategorien unterteilen:

- Informationen über die Konfiguration der SQL Server-Instanz
- Ablaufverfolgungen des SQL Server Profilers
- Informationen über aufgetretene Blockierungen
- Windows-Leistungsindikatoren
- Windows-Ereignisprotokolle

Ebenfalls über die Kommandozeile geben Sie das Ausgabeverzeichnis für die erzeugten Protokolldateien an. (Es werden tatsächlich mehrere Protokolldateien erzeugt.)

Es ist auch möglich, SQLdiag gänzlich ohne Parameter aufzurufen. In diesem Fall arbeitet das Programm im Standardmodus. Dabei werden die in Abbildung 4.23 aufgelisteten Optionen verwendet.

Parameter	Wert
Konfigurationsdatei	*SQLDiag.xml*
Computer	Der Computer, auf dem SQLdiag gestartet wird
SQL Server Instanz	Alle SQL Server-Instanzen auf dem Computer, die erreicht werden können
Ausgabeverzeichnis	Das Unterverzeichnis *100\Tools\Binn\SQLDIAG* Ihrer SQL Server-Installation, also zum Beispiel *C:\Program Files\Microsoft SQL Server\100\Tools\Binn\SQLDIAG*
Informationen im Ausgabeverzeichnis	*computername_MSINFO32.TXT*: Die Systeminformationen des Betriebssystems *computername_instanzname_sp_sqldiag_Shutdown.OUT*: Diese Datei wird für jede SQL Server-Instanz erstellt. Hier finden Sie zum Beispiel Ereignisse und Ereignisdaten der SQL Server-Ablaufverfolgung, Fehlerprotokolle, Konfigurationsoptionen und die Ergebnisse der statistischen Systemfunktionen (die Sie in Abschnitt 4.7 kennengelernt haben)

Tabelle 4.3: *Standardparameter für SQLdiag*

Im Allgemeinen wird diese Standardkonfiguration für Ihre Bedürfnisse nicht geeignet sein und Sie werden eine spezielle Konfigurationsdatei sowie weitere Kommandozeilenparameter verwenden, um eine auf Ihr Problem angepasste Protokollierung zu konfigurieren. Um einen schnellen Überblick über den Gesundheitszustand Ihrer SQL Server-Instanzen zu erhalten, können Sie durch einen einfachen Aufruf von SQLdiag sehr leicht die erforderlichen Informationen zusammentragen. Sie erhalten eine Ausgabe, wie in Abbildung 4.24 gezeigt.

```
SQLDIAG
2008/11/21 08:30:52.86 SQLDIAG Collector version
2008/11/21 08:30:52.86 SQLDIAG

IMPORTANT:  Please wait until you see "Collection started" before attempting to
reproduce your issue

2008/11/21 08:30:52.86 SQLDIAG Output path: C:\Program Files\Microsoft SQL Serve
r\100\Tools\Binn\SQLDIAG\
2008/11/21 08:30:52.92 SQLDIAG Collecting from 1 logical machine(s)
2008/11/21 08:30:52.93 NBNEW\* SQL Server version: 10
2008/11/21 08:30:52.93 NBNEW\* Machine name: NBNEW (this machine)
2008/11/21 08:30:52.93 NBNEW\* Target machine is not a cluster
2008/11/21 08:30:52.93 NBNEW\* Instance: WAIKIKI (32-bit)
2008/11/21 08:30:53.92 SQLDIAG Initialization starting...
2008/11/21 08:30:55.07 NBNEW\* MsInfo: Get MSINFO32
2008/11/21 08:30:55.28 NBNEW\* MsInfo: Get default traces
2008/11/21 08:30:55.57 NBNEW\* MsInfo: Get SQLDumper log
2008/11/21 08:30:55.93 SQLDIAG Initialization complete
2008/11/21 08:30:55.93 NBNEW\* Collecting diagnostic data

2008/11/21 08:30:56.93 SQLDIAG Collection started.  Press Ctrl+C to stop.
```

Abbildung 4.24: *Beispielausgabe für SQLdiag*

Die Überwachung und Protokollierung durch SQLdiag unterstützt die folgenden Anwendungsszenarien:

- Manuelles Starten und Beenden. Wenn Sie SQLdiag im normalen Modus starten, beginnt die Protokollierung unmittelbar nach dem Start. Das Beenden der Protokollierung müssen Sie manuell durch Betätigen der Tastenkombination Strg + C vornehmen.
- Erstellen einer Momentaufnahme. Über den Parameter */X* legen Sie fest, dass die Protokollierung nach dem Start sofort beginnt und SQLdiag nach der Sammlung aller Informationen automatisch beendet wird.
- Fortlaufender Modus. Durch den Parameter */L* startet SQLdiag im Schlafmodus. Die Protokollierung beginnt also nicht sofort, sondern zu einem festgelegten Zeitpunkt, den Sie über den Parameter */B* angeben. Auch die Beendigung erfolgt automatisch zu dem über den Parameter */E* angegebenen Zeitpunkt.

Die detaillierte Konfiguration der in die Überwachung und Protokollierung aufzunehmenden Daten wird über eine XML-Konfigurationsdatei vorgenommen, die Sie an SQLdiag über den Parameter */I* übergeben. In der Online-Dokumentation sind leider keinerlei Hinweise zum Aufbau der Konfigurationsdatei zu finden, sodass Sie sich dort alleine zurechtfinden müssen. Glücklicherweise werden aber Vorlagen mitgeliefert, die Sie entsprechend anpassen können. Diese Vorlagen finden Sie im Verzeichnis *100\Tools\Binn*. Die folgenden drei Vorlagen stehen Ihnen zur Verfügung:

- *SD_General.xml*. Diese Vorlage enthält die Einstellungen für die Protokollierung von Windows-Ereignissen, Datensammlungen des Windows-Systemmonitors und minimalen SQL Server-Ablaufverfolgungsereignissen.

- *SD_Detailed.xml*. Auch in dieser Vorlage sind die Einstellungen für Protokollierungen der Windows-Ereignisse und des Systemmonitors enthalten, allerdings ausführlicher. Insbesondere erhalten Sie bei Verwendung dieser Vorlage sehr umfangreiche SQL Server Profiler-Ablaufverfolgungsereignisse.
- *SQLDiag.xml*. Diese Vorlage wird von SQLdiag verwendet, falls Sie über den Kommandozeilenparameter /I keine explizite Vorlage angeben.

In vielen Fällen enthalten diese Vorlagen bereits die geeigneten Einstellungen für eine Überwachung, sodass Sie einfach mit einer Kopie einer solchen Standardvorlage arbeiten können. Wichtig ist, dass Sie in der Konfigurationsdatei den Computernamen und den Namen der zu überwachenden SQL Server-Instanz sowie auch die Verbindungsinformationen anpassen. Hierzu verwenden Sie die folgenden beiden Bereiche:

- Anpassen des Computernamens

  ```
  <Machine name="Z1">
  ```

- Anpassen der SQL Server-Instanz und der Verbindungsinformationen

  ```
  <Instance name="HAL9000" windowsauth="true"
     ssver="10" user="">
  ```

Die anderen erforderlichen Anpassungen der Konfigurationsdatei sind relativ einfach. Für alle Ereignisse, die in das Protokoll aufgenommen werden sollen, gibt es jeweils ein Attribut `enabled`, dass Sie einfach entsprechend auf `true` oder `false` setzen können. Eine entsprechende Sektion in der Konfigurationsdatei sieht zum Beispiel so aus:

```
<EventType name="Database" enabled="true">
  <Event id="92" name="Data File Auto Grow"  enabled="true" .../>
  <Event id="94" name="Data File Auto Shrink"  enabled="true" .../>
...
</EventType>
```

In allen Standardvorlagen ist die Überwachung und Protokollierung für Blockierungen ausgeschaltet. In vielen Fällen sind aber gerade Blockierungen eine wesentliche Ursache für Probleme bzw. eine mangelhafte Leistung. Sie müssen diese Überwachungsoption manuell einschalten, falls Sie die Blockierungsereignisse in die Protokolle mit aufnehmen möchten. Hierzu ändern Sie die folgende Sektion:

```
<BlockingCollector enabled="true"
        pollinginterval="5" maxfilesize="350" filecount="1"/>
```

Wenn sie sich einmal durch die Konfiguration durchgearbeitet haben und mit einer erstellten Konfigurationsdatei eine Diagnose starten und beenden, finden Sie letztlich im Ausgabeverzeichnis von SQLdiag eine Reihe von Dateien, in denen die konfigurierte Information gespeichert ist. Teilweise ist das Dateiformat einfacher Text, was eine manuelle Analyse erfordert und daher etwas mühsam ist. Diese Textprotokolle sind wahrscheinlich eher für den Microsoft-Support geeignet, um wirklich systeminterne Informationen abzufragen und festzustellen, unter welchen Umständen und mit welchen Parametern SQLdiag ausgeführt wurde.

Wirklich interessant ist die Möglichkeit, die ebenfalls erzeugte SQL Server-Ablaufverfolgung im Profiler zu öffnen und anschließend die Leistungsdaten des Windows Systemmonitors, die Sie ebenfalls im Ausgabeverzeichnis finden, zu importieren. Diese Vorgehensweise haben Sie in Abschnitt 4.5 bereits kennengelernt. Betrachten Sie hierzu bitte noch einmal Abbildung 4.20. Hier spielt SQLdiag wirklich seine Stärken aus – immer vorausgesetzt, dass Sie sich einmal die Arbeit gemacht und eine entsprechende Konfigurationsdatei erstellt haben. Über den fortlaufenden Modus (Parameter */L*), in dem Sie über die Parameter */B* und */E* Zeitpunkte für den den Start und das Ende der Protokollierung angeben müssen, können Sie die Protokollierung einschränken und SQLdiag so ausführen, dass die Protokollierung automatisch nur den Sie interessierenden Zeitraum umfasst.

Bei aller Nützlichkeit von SQLdiag und der äußerst bequemen Art und Weise, sehr einfach detaillierte Informationen über den Zustand einer SQL Server-Instanz und des Betriebssystems zu erhalten, bedenken Sie bitte auch hier, dass sich die Protokollierung drastisch auf die Systemleistung auswirken kann, sofern das Protokoll zu ausführlich wird. Insbesondere die in der Datei *SD_Detailed.xml* enthaltene Konfiguration führt zu einer sehr umfangreichen Protokollierung, die nur in seltenen Fällen wirklich sinnvoll sein dürfte. Es gilt also, wie so oft, einen Mittelweg zu finden, bei dem Sie nur die wirklich benötigten Informationen in die Protokollierung einschließen.

4.11 Ausführungspläne

In Kapitel 3 haben Sie bereits eine kurze Einführung in die Darstellung von Abfrageplänen erhalten. An dieser Stelle soll diese Einführung nun etwas erweitert werden und zwar dahingehend, dass Sie in die Lage versetzt werden, mögliche Probleme durch die Auswertung des vom Optimierer erzeugten Abfrageplans zu erkennen.

Wir betrachten hier nur den grafischen Ausführungsplan für die Analyse. Dieser Plan enthält, wie bereits in Kapitel 3 erwähnt, Operatoren (Knoten) und Pfeile (Kanten), die den Datenfluss zwischen den Operatoren verdeutlichen. Für komplexe Abfragen kann der grafische Ausführungsplan ebenfalls sehr komplex werden. Abbildung 4.25 zeigt ein weiteres Beispiel für einen einfachen grafischen Ausführungsplan.

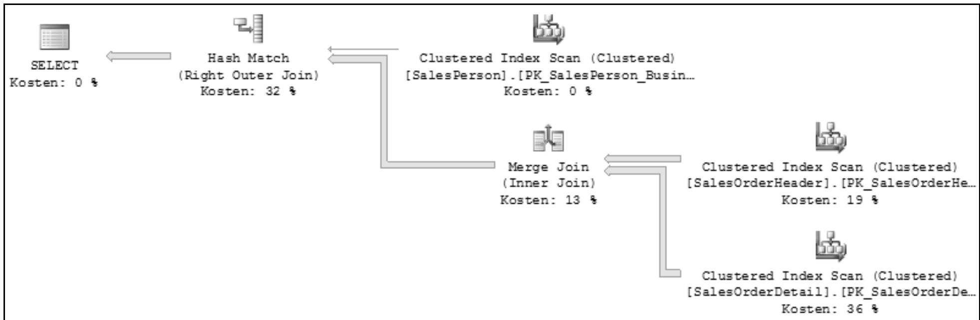


Abbildung 4.25: *Beispiel für einen grafischen Ausführungsplan*

Sie werden im weiteren Verlauf dieses Buches eine Vielzahl solcher Pläne sehen und anhand dieser Pläne ein Verständnis dafür entwickeln, auf welche Weise der Optimierer arbeitet. Der Ausführungsplan ist auch geeignet, um problematische Abfragen zu analysieren bzw. herauszufinden, worin genau das Problem liegt.

In Kapitel 3 haben Sie bereits eine allgemeine Einführung in die Arbeit mit Ausführungsplanen erhalten. An dieser Stelle soll nun die Einführung fortgesetzt werden. Zunächst möchte ich Ihnen die wichtigsten Operatoren vorstellen, die Ihnen in Ausführungsplänen immer wieder begegnen werden. Anschließend erhalten Sie einige Tipps zur Analyse von Ausführungsplänen und erfahren, worauf Sie in einem Ausführungsplan achten sollten, um mögliche Probleme zu entdecken.

4.11.1 Wichtige Operatoren in Ausführungsplänen

Eine Aufzählung aller verfügbaren, insgesamt über 100 Operatoren ist an dieser Stelle nicht angebracht, dafür ist die Online-Dokumentation wiederum der geeignetere Ort. Einige Operatoren, die Sie in den meisten Abfrageplänen finden, sollten Sie jedoch in jedem Fall mehr oder weniger beherrschen. Diese Operatoren möchte ich Ihnen in diesem Abschnitt kurz vorstellen. Im weiteren Verlaufe dieses Buches werden Ihnen dann noch eine Reihe anderer Operatoren begegnen.

Eines noch vorweg: Leider ist die deutsche Namensgebung für die Operatoren alles andere als gelungen. Manche Namen für die Operatoren wurden gar nicht übersetzt, andere nur teilweise. Heraus kommt auch hier wieder einmal eine mehr oder weniger »denglishe« Bezeichnung, also eine Mischung aus Deutsch und Englisch. Wundern Sie sich also bitte nicht über die manchmal etwas seltsam anmutenden Namen.

Ich habe die Operatoren in unterschiedliche Kategorien eingeteilt, um etwas Ordnung in die Aufzählung zu bringen. Jede Kategorie wird in den folgenden Abschnitten in einer eigenen Tabelle dargestellt. Leider ist es so, dass in den Erklärungen für die Operatoren ein wenig vorgegriffen werden muss, weil viele Konzepte erst in späteren Kapiteln behandelt werden. In den Erklärungen finden Sie jeweils einen Hinweis auf die weiterführenden Kapitel.

Such-Operatoren

Die wichtigsten Operatoren zur Suche von Daten in Tabellen und Indizes zeigt Abbildung 4.25. Die gegebenen Erklärungen nehmen hier schon auf Indexkonzepte Bezug, die wir erst in den Kapiteln 5 und 6 behandeln werden. Mehr zur Indexoptimierung erfahren Sie in den Kapiteln 11 und 13.

Operator	Symbol	Erklärung
Table Scan		Dieser Operator steht für die sequenzielle Suche in einer Tabelle, die über keinen gruppierten Index verfügt. Bei einer solchen Operation müssen alle Datenseiten der Tabelle gelesen werden.
Clustered Index Scan		Auch dieser Operator durchsucht alle Datenseiten einer Tabelle sequenziell. Der Unterschied zum Table Scan ist lediglich, dass für diese Tabelle ein gruppierter Index existiert.
Nonclustered Index Scan		Dieser Operator liest alle Daten in einem nicht gruppierten Index. Auch diese Suche wird sequenziell durchgeführt, es werden also wirklich alle Indexseiten gelesen.
Nonclustered Index Seek		In einem nicht gruppierten Index wird eine Suche über den Indexbaum durchgeführt.
Clustered Index Seek		Auch hier wird über den Indexbaum gesucht; aber in diesem Fall über den gruppierten Index einer Tabelle.
Schlüsselsuche (Clustered)		Wenn über eine Suchoperation nicht alle erforderlichen Spalten geholt werden konnten, dann müssen die fehlenden Spaltenwerte über eine Suche im gruppierten Index sozusagen nachgeladen werden, sofern für die Tabelle ein gruppierter Index existiert. Mehr hierzu erfahren Sie in Kapitel 5.

Tabelle 4.4: *Such Operatoren*

JOIN-Operatoren

JOIN-Operatoren verarbeiten Eingaben auf zwei Kanälen und erzeugen einen Ausgabestrom (siehe zum Beispiel Abbildung 4.25 und dort den Hash Match-Operator). Die in Tabelle 4.5 aufgeführten Operatoren werden dabei nicht nur für die Verknüpfung von Tabellen verwendet. Wenn eine Suche über einen Index durchgeführt wird, aber die Indexdaten nicht alle im weiteren Verlauf benötigten Daten enthalten, dann müssen diese Daten über zusätzliche Suchen ermittelt werden. In solch einem Fall dient ein JOIN-Operator dazu, die Daten der beiden Suchoperationen miteinander zu verknüpfen. Wenn Ihnen das im Moment noch zu kompliziert erscheint, dann gedulden Sie sich bitte bis Kapitel 5, dort erfahren Sie mehr über Indizes und Indexsuchen. In Kapitel 9 werden wir uns dann ausführlich mit den unterschiedlichen physikalischen JOIN-Operatoren auseinandersetzen. Verstehen Sie daher bitte Tabelle 4.5 nur als eine erste Einführung in die Thematik.

Operator	Symbol	Erklärung
Merge Join		Ein Merge Join verarbeitet sortierte Eingabedaten. Dadurch ist dieser Operator vor allem bei der Verknüpfung großer Datenmengen, die bereits (etwa durch einen geeigneten Index) sortiert vorliegen, eine gute Wahl. Möglicherweise ist es auch sinnvoll, die Daten vorher durch einen Sort-Operator (siehe unten) zu sortieren. In Abbildung 4.25 sehen Sie ein Beispiel für einen Merge Join. Bei einer Merge Join-Operation werden die Eingabeströme prinzipiell wechselseitig verarbeitet, wohingegen die anderen Join-Operatoren als äußere und innere Schleife abgearbeitet werden.
Hash Match		Wenn größere Datenmengen, die nicht entsprechend der Verknüpfungsbedingung sortiert sind, verknüpft werden sollen, ist es oftmals sinnvoll, eine Hash-Tabelle als Hilfskonstrukt zu erzeugen. Über diese Hash-Tabelle werden dann Suchoperationen und Verknüpfungen durchgeführt. In Abbildung 4.25 sehen Sie den Hash Match-Operator, der dort für den Right Outer Join verwendet wird. Sie können sich den Hash Match-Operator wie eine Schleife vorstellen. Der obere Eingabezweig ist die äußere Schleife, über die die Hash-Werte bestimmt werden. Für jeden Wert in der äußeren Schleife wird dann eine Suchoperation für den ermittelten Hash-Wert entsprechend des unteren Zweiges durchgeführt. Wird der Wert gefunden, so stimmen beide Zeilen überein, und die Zeile wird in die Hash-Tabelle aufgenommen.
Nested Loops		Auch der Nested Loop-Operator wird als innere und äußere Schleife ausgeführt. Die Suche wird hier aber nicht durch eine zusätzliche Hash-Tabelle unterstützt. Dieser Operator ist deshalb in den meisten Fällen dann sinnvoll, wenn in der äußeren Schleife nur wenige Zeilen enthalten sind und die innere Schleife durch einen unterstützenden Index durchsucht werden kann. Möglicherweise wird der Optimierer auch einen temporären Index erstellen, um die innere Suche zu beschleunigen (siehe Spool-Operator weiter unten).

Tabelle 4.5: *JOIN-Operatoren*

Alle JOIN-Operatoren kennen verschiedene Unterarten, wie zum Beispiel den SEMI JOIN, den INNER JOIN und die verschiedenen OUTER JOIN-Typen. Hierauf wird etwas weiter unten und dann später in Kapitel 9 nochmals eingegangen.

Sonstige Operatoren

Die in Tabelle 4.6 aufgeführten Operatoren werden Sie ebenfalls recht häufig in Ihren Abfrageplänen finden.

Operator	Symbol	Erklärung
Sort		Der Sort-Operator sortiert die Eingabe entsprechend der Sortierbedingung.
Spool		Ein Spool-Operator erstellt ein temporäres Objekt in der Datenbank *tempdb*. In manchen Fällen kann das Speichern eines Zwischenergebnisses in dieser Form zu einer Verbesserung der Abfrageleistung beitragen. Die Beschleunigung kommt dadurch zustande, dass der entsprechende Abfragezweig nicht mehrfach durchlaufen werden muss, sondern stattdessen auf das Zwischenergebnis zurückgegriffen werden kann. Das temporäre Objekt kann zum Beispiel ein Index oder eine Tabelle sein. Ein Spool kann hierbei in zwei unterschiedlichen Formen auftreten: Eager und Lazy. Der Eager Spool lädt alle benötigten Zeilen, sobald die erste Zeile aus dem Spool benötigt wird. Bei einem Lazy Spool erfolgt das Laden der Zeilen auf eine Anforderung hin. Eine Zeile wird erst dann im Spool gespeichert, wenn sie angefordert wird.
Filter		Der Filter-Operator ist ein Scan-Operator. Er durchsucht die Eingabe sequenziell und erstellt eine Ausgabe entsprechend der Filterbedingung.

Tabelle 4.6: *Weitere wichtige Operatoren*

Wenn Sie einen grafischen Ausführungsplan analysieren, dann kommen Sie mit den oben aufgeführten Operatoren bereits ein gutes Stück weiter. Auf längere Sicht allerdings werden Sie nicht umhinkommen, neben diesen Basisoperatoren auch noch einige weitere zu kennenzulernen. Es wird Ihnen vor allem in der ersten Zeit immer wieder passieren, dass Sie Operatoren in einem Plan entdecken, die Ihnen unbekannt sind. Glücklicherweise finden Sie im grafischen Ausführungsplan zu jedem Operator weitere Informationen, wobei auch eine Kurzbeschreibung über die Funktion des Operators ausgegeben wird. Welche Informationen dies sind, und wie Sie diese Informationen finden, erklärt der folgende Abschnitt.

4.11.2 Eigenschaften von Operatoren

Schauen Sie sich bitte noch einmal Abbildung 4.25 an. Dort finden Sie für jeden Operator eine Angabe über die relativen Kosten. Allein mittels dieser Angabe können Sie sehr schnell einen Überblick über die »Zeitfresser« in einer Abfrage erhalten.

Sobald Sie den Mauszeiger über einen Operator bewegen, öffnet sich ein Fenster mit zusätzlichen Informationen zu diesem Operator. Ein Beispiel hierfür sehen Sie in Abbildung 4.26.

Hash Match

Jede Zeile der oberen Eingabe zum Erstellen einer Hashtabelle und jede Zeile der unteren Eingabe zum Einfügen in die Hashtabelle verwenden, wobei alle übereinstimmenden Zeilen ausgegeben werden.

Physischer Vorgang	Hash Match
Logischer Vorgang	Inner Join
Tatsächliche Anzahl von Zeilen	296
Geschätzte E/A-Kosten	0
Geschätzte CPU-Kosten	0,0201919
Geschätzte Anzahl von Ausführungen	1
Anzahl von Ausführungen	1
Geschätzte Operatorkosten	0,0201954 (7 %)
Geschätzte Unterstrukturkosten	0,261817
Geschätzte Anzahl von Zeilen	266,028
Geschätzte Zeilengröße	303 B
Tatsächliche erneute Bindungen	0
Tatsächliche Zurückspulvorgänge	0
Knoten-ID	3

Einfügungshashschlüssel

[AdventureWorks2008].[HumanResources].[EmployeeDepartmentHistory].DepartmentID

Abbildung 4.26:
Beispiel für zusätzliche Informationen über einen Operator

Im oberen Teil des Fensters sehen Sie zunächst den Namen des Operators. Dort finden Sie auch eine Kurzbeschreibung über die Arbeitsweise des Operators. Sofern Sie den beschriebenen Operator noch nicht kennen, werden Sie aus dieser Kurzbeschreibung wahrscheinlich nicht sehr schlau. Die Information ist jedoch nützlich, wenn Sie die Arbeitsweise des Operators bereits einmal untersucht haben – zum Beispiel anhand der Online-Dokumentation. Unterhalb der Kurzbeschreibung finden Sie eine Tabelle mit den folgenden Informationen:

- **Physischer Vorgang**. Dies ist die tatsächliche physikalische Operation, die von der Query Engine ausgeführt wird.
- **Logischer Vorgang**. Der logische Vorgang spezifiziert die auszuführende Operation aus konzeptioneller Sicht. Mit anderen Worten: An dieser Stelle steht die tatsächlich auszuführende Operation. So finden Sie hier zum Beispiel die Aussage, dass ein INNER JOIN durchgeführt wird. Auf welche Art und Weise der JOIN dann tatsächlich physikalisch ausgeführt wird, bestimmt der physische Vorgang. Ein INNER JOIN kann zum Beispiel durch drei verschiedene physische Operatoren implementiert werden. Manche Operatoren kennen keine Unterscheidung zwischen dem logischen und physikalischen Vorgang; zum Beispiel der Table- oder der Clustered Index Scan. In diesem Fall ist der logische Vorgang gleich dem physikalischen Vorgang. Der Optimierer kann dann also nicht aus unterschiedlichen Operatoren zur Implementierung der Operation auswählen.
- **Tatsächliche Anzahl von Zeilen**. Hier finden Sie die Anzahl Zeilen, die der Operator verarbeitet hat. Diese Information gibt es nur im aktuellen Plan.
- **Geschätzte E/A-Kosten**. Dieser Wert enthält die voraussichtlichen (d.h. geschätzten) E/A-Kosten des Operators.

- **Geschätzte CPU-Kosten**. Der Wert gibt die geschätzten CPU-Kosten an, die der Operator verursacht. Die Summe aus CPU- und E/A-Kosten ergibt die gesamten Operatorkosten. Die Werte für CPU- und E/A-Kosten geben Ihnen auch einen Hinweis darauf, ob der entsprechende Operator hauptsächlich CPU- oder E/A-Last erzeugt. Der in Abbildung 4.26 dargestellte Hash Match-Operator erzeugt keinerlei E/A-, sondern ausschließlich CPU-Kosten.
- **Geschätzte Anzahl von Ausführungen**. An dieser Stelle wird angegeben, wie oft der Operator voraussichtlich ausgeführt wird.
- **Anzahl von Ausführungen**. Hier steht die tatsächliche Anzahl der Operatorausführungen, die bei der Ausführung der Abfrage bestimmt wurde. Dieser Wert ist nur im aktuellen Plan enthalten. Leider fehlt hier das Wort »tatsächlich«, so wie in allen anderen Informationen, die nur im aktuellen Plan vorhanden sind.
- **Geschätzte Operatorkosten**. Dieser Wert enthält die geschätzten Kosten des Operators. Wie bereits erwähnt, ergibt sich dieser Wert aus der Summe der E/A- und CPU-Kosten. Im grafischen Ausführungsplan finden Sie auch die relativen Kosten des Operators in Bezug auf die Gesamtkosten der Abfrage. Dieser prozentuale Anteil der Operatorkosten steht in Klammern unterhalb des Operators (siehe Abbildung 4.25).
- **Geschätzte Unterstrukturkosten**. An dieser Stelle finden Sie die kumulierten Kosten für den Operator selber und alle untergeordneten (also weiter rechts stehenden) Zweige. Insbesondere enthält also der Operator oben links die geschätzten Gesamtkosten der Abfrage. Bei SELECT-Abfragen ist dies immer der SELECT-Operator (siehe nochmals Abbildung 4.25).
- **Geschätzte Anzahl von Zeilen**. Bei der Erstellung des AbfragePlans stützt sich der Optimierer auf Kardinalitätsschätzungen, um den für eine Operation optimalen Operator auszuwählen. Wie dies genau funktioniert, erfahren Sie in Kapitel 9. Die erwartete Zeilenanzahl hat zum Beispiel entscheidenden Einfluss auf die Wahl eines Such- oder JOIN-Operators. Die vom Optimierer geschätzte Zeilenanzahl finden Sie an dieser Stelle.
- **Geschätzte Zeilengröße**. Hier finden Sie die ungefähre Zeilenlänge. Im Zusammenhang mit der Zeilenanzahl und der Anzahl Ausführungen können Sie so zum Beispiel abschätzen, wie viele Bytes der Operator insgesamt verarbeitet.
- **Tatsächliche erneute Bindungen und tatsächliche Zurückspulvorgänge**. Diese Informationen finden Sie nur im aktuellen Plan. Die beiden Werte sind allein für Nested Loop Joins interessant. In allen anderen Fällen ist der Wert für die erneuten Bindungen stets 1 und der Wert für die Zurückspulvorgänge 0. Jeder physische Vorgang wird durch den Aufruf einer internen *Init*-Methode wenigstens einmal initialisiert. Für einen Operator in der inneren Schleife eines Nested Loop Joins kann diese Initialisierung mehrfach erforderlich sein. Sobald die Initialisierung erneut erfolgt, werden die Zähler für die erneuten Bindungen und die Zurückspulvorgänge hochgesetzt. Eine erneute Bindung ist immer dann erforderlich, wenn sich die Parameter in der äußeren Schleife derart geändert haben, dass die innere Seite des JOIN-Operators komplett neu berechnet werden muss. Sofern sich die Parameter für die innere Suchoperation nicht geändert haben, kann das innere Ergebnis wieder verwendet werden. In diesem Fall ist nur ein Zurückspulen erforderlich; eine erneute Bindung wird dann nicht durchgeführt. Sie Summe von erneuten Bindungen und Zurückspulvorgängen sollte der Anzahl von Zeilen entsprechen, die in der äußeren Schleife des Nested

Loop-Operators verarbeitet werden. (Dies ist im grafischen Plan der obere Zweig.) Es ist allerdings auch möglich, dass beide Werte überhaupt nicht vorhanden sind, weil für den entsprechenden Operator weder erneute Bindungen noch Zurückspulvorgänge durchgeführt werden können. In diesem Fall hat der grafische Ausführungsplan ein Problem, weil dann der Wert 0 angezeigt wird.

- **Unterer Teil des Fensters**. Hier stehen weitere Informationen über die Ausführung der Operation. Insbesondere finden Sie hier die Liste der Ausgabeelemente sowie die Argumente des Operators.

Nach dieser kurzen Einführung wollen wir nun untersuchen, wie ein Abfrageplan dabei helfen kann, Probleme zu ermitteln.

4.11.3 Analyse von Ausführungsplänen

Beim Betrachten eines grafischen Abfrageplans können Sie mit etwas Übung Rückschlüsse auf mögliche Probleme ziehen. Diese Verfahrensweise zur Abfrageanalyse ist wirklich ausgezeichnet dafür geeignet, recht schnell die Schwachpunkte einer Abfrage aufzuspüren. Wir werden hiervon im weiteren Verlauf des Buches regen Gebrauch machen. Allerdings kann die Analyse größerer Pläne für komplizierte Abfragen auch schon einmal eine echte Herausforderung darstellen, ganz einfach deshalb, weil die Darstellung des Baumes für einen solchen Plan sehr viel Platz in Anspruch nimmt, und es daher schwierig ist, die Analyse systematisch durchzuführen.

Wenn Sie einen solch umfangreichen Plan betrachten, dann versuchen Sie, den Plan in verschiedene Teilpläne zu zerlegen, die Sie anschließend getrennt betrachten. Dies gelingt in der Regel immer. Eine große Hilfe bei der Navigation in umfangreichen Plänen ist dabei der in der unteren rechten Ecke des Ausführungsplans vorhandene Navigator (das [+]-Zeichen). Sobald Sie diese Schaltfläche betätigen, können Sie durch einfaches Ziehen der Maus im Ausführungsplan navigieren.

Betrachten Sie einen physikalischen Abfrageplan, so sollten Sie sich zu Beginn auf die folgenden Anhaltspunkte konzentrieren:

- **Prozentuale Operatorkosten**. Wie bereits erwähnt, steht unter jedem Operator der prozentuale Anteil der Kosten für den Operator. Die Summe aller dieser Kosten sollte also stets 100 % betragen. Sie können alleine durch das Inspizieren dieses Wertes herausfinden, ob die Kosten einer Abfrage im Wesentlichen durch einige Operatoren bestimmt werden und sich dann im Weiteren auf diese Operatoren konzentrieren. Natürlich finden Sie auf diese Weise auch die unkritischen Operatoren heraus, also diejenigen, die für die Gesamtabfragekosten nicht weiter ins Gewicht fallen.
- **Dicke Pfeile weit links oben**. Erinnern Sie sich bitte daran, dass die Stärke der Pfeile die Anzahl der übergebenen Zeilen repräsentiert. Der Optimierer wird stets versuchen, die Zeilenanzahl so früh wie möglich zu begrenzen, damit alle nachfolgenden Operatoren weniger Zeilen verarbeiten müssen. Dadurch sollten dicke Pfeile möglichst nur am Beginn der Ausführung, also weit rechts zu finden sein. Wenn Sie in Ihrem Plan stärkere Pfeile weit links oben finden, bedeutet dies, dass alle vorhergehenden Operatoren eine Vielzahl von Zeilen verarbeiten mussten. Dies ist möglicherweise ein Indiz dafür, dass eine Abfrage zu viele Zeilen liefert, etwa weil die Filterung des Ergebnisses erst auf dem Client erfolgt.

- **Scan-Operationen**. Scan-Operatorationen können ein Indiz für fehlende Indizes sein. Generell sind Scan-Operationen nicht schlecht an sich. Es ist sehr gut möglich, dass ein Table Scan oder ein Clustered Index Scan tatsächlich die effizienteste Möglichkeit ist, eine Suchoperation auszuführen. Beispielweise deshalb, weil sehr viele Zeilen die Filterbedingung passieren. Möglicherweise gibt es auch gar keine Filterbedingung; auch dann ist eine Scan-Operation oftmals absolut in Ordnung. Sie sollten jedoch verstehen, warum in Ihrer Abfrage Scan-Operationen ausgeführt werden, damit Sie entscheiden können, ob Sie etwas unternehmen sollten oder den Scan akzeptieren.
- **Schlüsselsuche (Clustered)**. Dies ist fast immer ein Indiz für einen fehlenden Index. Eine Schlüsselsuche im gruppierten Index ist dann erforderlich, wenn die initiale Suche von Zeilen, zum Beispiel über den Indexbaum eines nicht gruppierten Index, erfolgt, aber diese Suche nicht alle erforderlichen Daten zurückliefern kann. In diesem Fall ist ein »Nachladen« dieser fehlenden Daten über den gruppierten Index erforderlich. Die zusätzliche Schlüsselsuche im gruppierten Index ist oftmals ein erheblicher Kostenfaktor. Sie können diese Kosten in vielen Fällen durch das Erstellen geeigneter Indizes einsparen, wenn Ihnen die Kosten zu hoch erscheinen. In den Kapiteln 5, 9 und 11 werden wir hierauf noch näher eingehen.
- **Filter-Operatoren**. Ein Filter-Operator arbeitet sequenziell, ist also auch ein Scan-Operator. Wenn Sie in Ihrer Abfrage Filter-Operatoren mit relevanten Kosten vorfinden, kann dies ein Hinweis auf fehlende Indizes sein. Prüfen Sie bitte auch, ob Sie in Ihren Filterbedingungen in der `WHERE`-Klausel Funktionen verwenden. Dadurch wird in den meisten Fällen die Verwendung von Indizes unterbunden.
- **Sort-Operatoren**. Sort-Operationen sind in der Regel sehr kostspielig. Erinnern Sie sich bitte daran, dass der Sort-Operator ein *Stop And Go*-Operator ist, der also erst dann seine Arbeit beginnen kann, wenn alle Eingabezeilen vorliegen. Falls viele Zeilen sortiert werden müssen, wird der entsprechende Sort-Operator auch einen hohen Kostenanteil an den Gesamtabfragekosten aufweisen. Möglicherweise können Sie einen Index erstellen, der die Sortierung unterstützt.
- **Table Spool/Index Spool**. Das Anlegen von Spools bedeutet das Erzeugen temporärer Objekte in der Datenbank *tempdb*. Der Optimierer entscheidet sich für einen Spool, wenn er der Meinung ist, dass das Anlegen des Spools und der wiederholte Zugriff auf den Spool insgesamt günstiger sind als die wiederholte Ausführung des gesamten Zweiges, der durch den Spool abgebildet wird. Generell kann diese Verfahrensweise durchaus die Abfrage beschleunigen. Sie sollten jedoch verstehen, warum der Optimierer sich für das Anlegen eines Spools entscheidet. Es ist das Erzeugen des Spools, welches die Abfrageleistung möglicherweise negativ beeinflusst. Eventuell können Sie geeignete Indizes erstellen, um den Spool zu umgehen.
- **Parallele Ausführung von Abfragen**. Der Optimierer hat eine generelle Tendenz zur parallelen Ausführung von Abfragen. Zunächst einmal ist dagegen nichts einzuwenden. Falls mehrere Prozessoren vorhanden sind, warum dann nicht auch mehrere Prozessoren für die Abfrageausführung heranziehen? Das Problem ist hier, dass nicht alle Operatoren eine parallele Ausführung ermöglichen. Dies bedeutet, dass parallelisierte Daten auch wieder zusammengeführt werden müssen – spätestens natürlich vor dem finalen `SELECT`-Operator. Dieses Zusammenführen kann die Abfrageleistung vermindern. Generell sollte eine OLTP-Anwendung, die ja vorwiegend kleine Transaktionen verarbeitet, ohne einen merklichen Anteil an parallelen Abfragen auskommen. Wenn

große Datenmengen verarbeitet werden, also zum Beispiel in Reporting-Systemen, ist eine parallele Abfrageausführung absolut in Ordnung. In OLTP-Systemen sollten Sie herausfinden, warum eine Abfrage parallel ausgeführt wird, und dies nach Möglichkeit vermeiden. Erinnern Sie sich bitte daran, dass der Optimierer stets versucht, die Ausführungszeit zu minimieren, und dafür alle zur Verfügung stehenden Ressourcen nutzt. Eine massive parallele Ausführung von Abfragen kann Ihr System in die Knie zwingen. In OLAP-Anwendungen, bei denen in der Regel nur wenige Benutzer gleichzeitig arbeiten, ist dies sicher kein Problem. Falls Sie dagegen eine OLTP-Anwendung haben, die typischerweise viele Benutzer gleichzeitig bedient, dann kann eine gleichzeitige parallele Abfrageausführung von vielen Verbindungen das System bis an seine Grenze belasten. Wenn Sie parallele Abfragen entdecken, untersuchen Sie also bitte, ob dies wirklich erforderlich ist. Eventuell können auch hier wieder Indizes helfen.

- **Differenzen zwischen geschätzten und tatsächlichen Werten**. Der Optimierer erstellt einen Plan auf der Grundlage von Kardinalitätsschätzungen. Hierbei werden verschiedene Schätzwerte zugrunde gelegt, die Sie weiter oben bereits kennengelernt haben. Wenn Sie einen tatsächlichen Ausführungsplan anschauen, können Sie die geschätzten Werte den real aufgetretenen Werten gegenüberstellen. Sollten Sie hierbei merkliche Unterschiede entdecken, ist dies ein Indiz dafür, dass der Optimierer keinen optimalen Plan erstellen konnte. Oftmals ist dies darauf zurückzuführen, dass die existierenden Statistiken nicht aktuell sind. Hierauf kommen wir in Kapitel 9 noch einmal zurück.

  Bitte achten Sie insbesondere auf relevante Unterschiede zwischen der geschätzten Zeilenanzahl und der tatsächlichen Zeilenanzahl. Ein falscher Schätzwert kann dazu führen, dass der Optimierer einen nicht optimalen Operator für eine Operation auswählt und die Abfrageleistung sich enorm verschlechtert. In Kapitel 9 werden Sie hierfür einige Beispiele finden. Bedenken Sie bei der Untersuchung bitte, dass in der inneren Schleife eines Schleifenoperators die Zeilenanzahl nicht absolut, sondern je Ausführung angegeben wird. Hier müssen Sie also die tatsächliche Zeilenanzahl errechnen, indem Sie das Produkt aus der Anzahl der Ausführungen und der Anzahl von Zeilen je Ausführung bilden.

- **Warnungen.** Entdecken Sie bei einem Operator ein gelbes Ausrufezeichen, dann hat der Optimierer ein Problem festgestellt, auf das er Sie durch eine entsprechende Warnung hinweist. Eine solche Warnung kann zum Beispiel durch eine fehlende Statistik oder eine fehlende Verknüpfungsbedingung in einem JOIN hervorgerufen sein. Sie sollten unbedingt verstehen, warum die Warnung erzeugt wird und eventuell reagieren. Zum Beispiel durch das Aktualisieren oder Erstellen der entsprechenden Statistiken. Mehr hierzu ebenfalls in Kapitel 9.

Untersuchen Sie diese Indikatoren, dann erhalten Sie in vielen Fällen wertvolle Anhaltspunkte über Schwachstellen oder Probleme in Ihren Abfragen. Wie bereits gesagt, werden wir im weiteren Verlauf dieses Buches hiervon Gebrauch machen. Sie werden anhand von Beispielen sehen, wie Abfragen unter Verwendung des grafischen Ausführungsplans analysiert werden.

Der SQL Server Profiler bietet im Übrigen auch eine Möglichkeit, den Abfrageplan in die Ablaufverfolgung mit aufzunehmen. Hierfür existieren diverse Ereignisse für Abfragepläne im Text- oder im XML-Format, zum Beispiel *Performance:Showplan XML*. Wenn Sie

dieses Ereignis in Ihre Ablaufverfolgung einschließen, dann erhalten Sie auch den Abfrageplan im grafischen Format (siehe Abbildung 4.27).

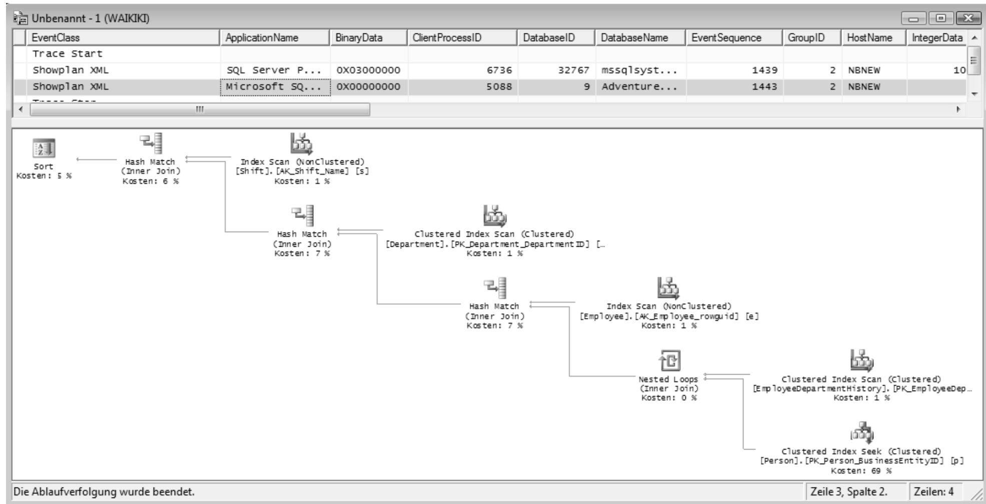


Abbildung 4.27: *Der Abfrageplan im SQL Server Profiler*

Gehen Sie aber bitte vorsichtig mit dieser Option um. Das entsprechende Ereignis benötigt eine Menge Ressourcen auf dem Server und ebenso viel Speicherplatz zum Speichern der Ablaufverfolgung.

Aus Kapitel 3 wissen Sie bereits, dass SQL Server bestrebt ist, einmal erstellte Ausführungspläne zur späteren Wiederverwendung im Plan Cache zu speichern. Dieses Verhalten können Sie ausnutzen, um die im Cache vorhanden Abfragepläne abzufragen und somit Informationen über die Ausführung bereits beendeter Abfragen zu erhalten. Verwenden Sie hierzu die dynamische Verwaltungssicht `sys.dm_exec_cached_plans`. Diese Verwaltungssicht gibt neben einigen statistischen Informationen auch einen `plan_handle` zurück. Mit diesem `plan_handle` können Sie den entsprechenden Abfrageplan durch einen Aufruf der Funktion `sys.dm_exec_query_plan` ermitteln. Eine entsprechende Abfrage sieht dann etwa so aus:

```
select cp.usecounts,qp.query_plan
  from sys.dm_exec_cached_plans as cp
       cross apply sys.dm_exec_query_plan(cp.plan_handle) as qp
```

Die Abfrage gibt den Plan im XML-Format zurück. Durch einen Klick auf diesen Plan erhalten Sie die grafische Darstellung.

Es ist nicht möglich, aus den von der obigen Abfrage zurückgelieferten XML-Verweisen direkt Rückschlüsse auf die enthaltenen Abfragen zu ziehen. Die obige Abfrage ist daher sicherlich nur sehr bedingt dafür geeignet, problematische Abfragen aufzuspüren, da jeder einzelne zurückgelieferte Plan untersucht werden muss. Normalerweise wird der Plan Cache eines Produktivsystems wenigstens einige tausend Abfragen enthalten, was das Aufspüren der problematischen Abfragen durch die obige Abfrage nahezu unmög-

lich macht. Die Verwaltungssichten `sys.dm_exec_cached_plans` und `sys.dm_exec_query_plan` enthalten einfach zu wenig Informationen für eine derartige Analyse.

Hier hilft die dynamische Verwaltungssicht `sys.dm_exec_query_stats` weiter. Diese Sicht enthält sehr nützliche Informationen zu Ausführungszeiten und zur Ressourcenverwendung von Abfragen und darüberhinaus auch Verweise zum Abfragetext und -plan. Wenn Sie `sys.dm_exec_query_stats` zusammen mit `sys.dm_exec_query_plan` und der Funktion `sys.dm_exec_sql_text` verwenden, können Sie auf einfache Art und Weise eine »Hitliste« Ihrer Abfragen erstellen und somit zum Beispiel die Abfragen mit der größten CPU-Last oder den meisten E/A-Operationen herausfinden. Eine entsprechende Abfrage zur Ermittlung der zehn Abfragen mit der längsten Ausführungsdauer sieht daher ungefähr so aus:

```
select top 10
       t.text as SqlAnweisung
      ,execution_count as AnzahlAusführungen
      ,total_physical_reads as [Leseoperationen/Disk]
      ,total_logical_reads as [Gelesene Seiten/Puffer]
      ,total_logical_writes as [Geschriebene Seiten]
      ,cast(total_worker_time/1000.0 as decimal(8,2)) as [CPU/ms]
      ,cast(total_elapsed_time/1000.0 as decimal(8,2)) as [Dauer/ms]
      ,p.query_plan as Abfrageplan
  from sys.dm_exec_query_stats
       cross apply sys.dm_exec_sql_text(sql_handle) as t
       cross apply sys.dm_exec_query_plan(plan_handle) as p
 order by [Dauer/ms] desc
```

Ein Beispiel für das Ergebnis dieser Abfrage sehen Sie in Abbildung 4.28.

SqlAnweisung	AnzahlAusführungen	Leseoperationen/Disk	Gelesene Seiten/Puffer	Geschriebene Seiten	CPU/ms	Dauer/ms	Abfrageplan
select ProductId,sum(lineT...	6	1	7440	0	1129.06	1129.06	<ShowPlanXML xmlns="http:...
select sp.* from Sales.Sale...	1	4	43	0	13.00	326.02	<ShowPlanXML xmlns="http:...
(@1 smallint)SELECT * FR...	4	0	476	0	14.00	312.02	<ShowPlanXML xmlns="http:...
select * from HumanReso...	5	9	5285	0	38.00	224.01	<ShowPlanXML xmlns="http:...
select * from SaLes.SalesO...	12	0	36	0	1.00	1.00	<ShowPlanXML xmlns="http:...

Abbildung 4.28: *Gespeicherte Ausführungspläne mit statistischen Informationen*

Auch hier können Sie über einen Klick auf den Verweis in der letzten Spalte den grafischen Ausführungsplan ansehen.

Wir werden in den Kapiteln 9 und 10 noch einmal darauf zurückkommen, wie problematische Abfragen unter Verwendung der dynamischen Verwaltungssichten gefunden werden können. An dieser Stelle möchte ich Sie nur noch einmal daran erinnern, dass die in den dynamischen Verwaltungssichten gespeicherten Informationen flüchtig sind. Es ist also keinesfalls sichergestellt, dass Sie im Plan Cache tatsächlich alle ausgeführten Abfragen wiederfinden. Der Plan Cache ist dynamisch, und selten benötigte Pläne werden wieder aus diesem Cache entfernt, sobald der Hauptspeicher knapp wird. Außerdem kann der gesamte Cache durch die Anweisung `DBCC FREEPROCCACHE` geleert werden. Auch dann gehen historische Informationen verloren.

Zum Schluss dieses Abschnitts möchte ich Sie noch einmal explizit daran erinnern, dass ein Ausführungsplan stets vor der Ausführung einer Abfrage erstellt wird. Es ist daher

nicht etwa so, dass es einen tatsächlichen Plan und einen geschätzten Plan gibt. Der Optimierer entscheidet sich für einen bestimmten Plan, nach dem die Abfrage ausgeführt wird. Im tatsächlichen Ausführungsplan erhalten Sie lediglich erweiterte Informationen, die erst durch eine Ausführung der Abfrage ermittelt werden können.

Hierbei kann es durchaus einmal vorkommen, dass der Optimierer einen nicht optimalen Plan auswählt. Die Ursachen hierfür werden wir in Kapitel 9 untersuchen. Bei sehr komplexen Abfragen ist es auch möglich, dass der Optimierer sich total »verplant«. Einen Auszug aus einem solchen Fehlplan sehen Sie in Abbildung 4.29. Betrachten Sie die für jeden Operator angegebenen prozentualen Kosten, die deutlich über 100 Prozent für die gesamte Abfrage liegen, was natürlich äußerst fragwürdig ist. In einem solchen Fall bleibt Ihnen meist nichts anderes übrig, als die komplexe Abfrage in einfachere Unterabfragen aufzuteilen und getrennt auszuführen.

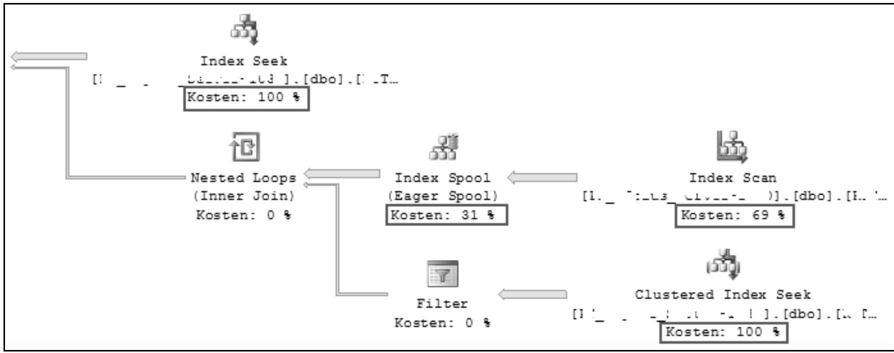


Abbildung 4.29: *Beispiel für eine Fehleinschätzung des Optimierers*

4.12 Datenauflistungen

Diese ist eine meiner Lieblingserweiterungen im SQL Server 2008, deren Name »Datenauflistungen« zunächst nicht vermuten lässt, welch mächtiges Werkzeug sich dahinter verbirgt. SQL Server erlaubt nun die Konfiguration eines sogenannten *Verwaltungs-Data Warehouse*, das früher auch als *Performance Data Warehouse* bezeichnet wurde. Dieses Data Warehouse fungiert als ein zentrales Repository für die Aufzeichnung von System- und insbesondere Leistungsdaten. Dadurch eröffnet sich Ihnen die Möglichkeit, Daten über das Systemverhalten permanent zu speichern und im Nachhinein zu untersuchen, wie das Systemverhalten zu einem bestimmten Zeitraum war. Erinnern Sie sich bitte daran, dass die dynamischen Verwaltungssichten nur Daten seit dem letzten Start von SQL Server zurückliefern. Wenn Sie diese Daten permanent speichern wollten, mussten Sie in der Vergangenheit zum Beispiel entsprechende SQL Server Agent-Aufträge anlegen, die in zyklischen Abständen Ergebnisse von dynamischen Verwaltungssichten in eigens hierfür angelegte Tabellen übertrugen. Die Inhalte dieser Tabellen konnten Sie dann später (zum Beispiel mit Excel oder auch durch eigens hierfür entwickelte Berichte) auswerten. Möglicherweise haben Sie eine solche Lösung ja bereits einmal gesehen oder auch schon selber ausprobiert.

Diese umständliche Verfahrensweise gehört nun der Vergangenheit an. Mit SQL Server 2008 können Sie ein Verwaltungs-Data Warehouse konfigurieren und die Aufzeichnung von Leistungsdaten zu bestimmten Zeitpunkten veranlassen. Dadurch erhalten gemessene Leistungsdaten eine Historie, und Sie haben die Möglichkeit, das gegenwärtige Systemverhalten mit Performance-Daten aus der Vergangenheit zu vergleichen – für die Analyse von Performance-Problemen eine oftmals erhebliche Arbeitserleichterung.

4.12.1 Konfiguration eines Verwaltungs-Data Warehouse

Die Standard SQL Server-Installation ermöglicht nicht die Konfiguration eines Verwaltungs-Data Warehouse. Wenn Sie ein Verwaltungs-Data Warehouse einrichten möchten, müssen Sie dies nach der Installation separat durchführen. Bei der Einrichtung hilft das Management Studio. Bevor Sie mit der Einrichtung eines Verwaltungs-Data Warehouse (ich werde diesen Begriff in der Folge einfach durch *VDWH* abkürzen) beginnen, sollten Sie aber noch folgende Überlegungen anstellen:

Beim VDWH handelt es sich letztlich um eine separate SQL Server-Datenbank, in die gemessene Leistungsdaten eingetragen und für Auswertungen bereitgestellt werden. Generell können Sie je SQL Server-Instanz ein eigenes VDWH anlegen. Dies widerspricht aber dem Gedanken eines gemeinsamen Repository, in dem alle erforderlichen Analysedaten zentral gespeichert werden. Die ausdrückliche Empfehlung ist daher, ein VDWH für Ihre Server-Landschaft aufzusetzen. Sofern Sie mehrere SQL Server-Instanzen im Einsatz haben, können Sie dann tatsächlich an einer zentralen Stelle Informationen sammeln und auswerten. Natürlich kosten sowohl das Sammeln von Daten als auch die späteren Auswertungen entsprechende Ressourcen. Die Empfehlung ist daher ausdrücklich, das VDWH möglichst nicht auf einem bereits stark belasteten Produktivsystem zu installieren. Richten Sie am besten für das VDWH eine eigene SQL Server-Instanz ein oder wählen Sie zumindest eine Instanz aus, die nur wenig belastet wird.

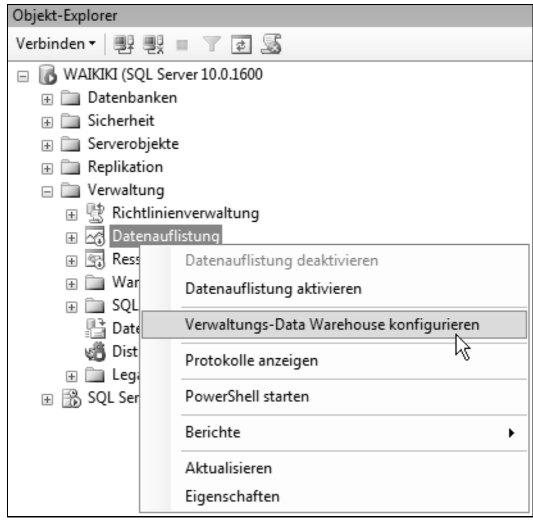


Abbildung 4.30:
Assistenten zur Konfiguration eines Verwaltungs-Data Warehouse starten

Wenn Sie sich für eine geeignete SQL Server-Instanz entschieden haben, können Sie das VDWH mit einem einfach zu bedienenden Assistenten auf dieser Instanz in Betrieb nehmen. Im ersten Schritt muss hierzu zunächst eine VDWH-Datenbank erstellt werden. Den hierfür vorhandenen Assistenten öffnen Sie einfach aus dem Kontextmenü für den Eintrag VERWALTUNG • DATENAUFLISTUNG im Objekt-Explorer (Abbildung 4.30).

Wählen Sie im Kontextmenü den Eintrag VERWALTUNGS-DATA WAREHOUSE KONFIGURIEREN aus, so wie in der Abbildung gezeigt. Auf der zweiten Seite des Assistenten wählen Sie bitte die Option VERWALTUNGS-DATA WAREHOUSE ERSTELLEN ODER AKTUALISIEREN aus (Abbildung 4.31).

Abbildung 4.31:
Das Verwaltungs-Data Warehouse konfigurieren

Anschließend müssen Sie sowohl eine SQL Server-Instanz als auch einen Datenbanknamen für die Speicherung des VDWH angeben. Sie können hierfür eine bereits vorhandene Datenbank verwenden; der Assistent ermöglicht Ihnen aber auch das Erstellen einer neuen Datenbank. Legen Sie am besten eine neue Datenbank an und nennen Sie diese Datenbank beispielsweise einfach *VDWH*. Anschließend haben Sie noch die Möglichkeit, Zugriffsrechte auf diese Datenbank – sowohl für das Einfügen von Daten als auch für die Abfrage gespeicherter Daten – zu vergeben. Hierfür existieren drei Datenbankrollen in der neu angelegten Datenbank:

- **mdw_admin**. Mitglieder dieser Rolle haben alle Berechtigungen.
- **mdw_reader**. Mitglieder dieser Rolle dürfen Daten abfragen.
- **mdw_writer**. Mitglieder dieser Rolle dürfen Daten im VDWH verändern.

Da es uns in den folgenden Experimenten und Untersuchungen lediglich darum geht, die generelle Funktionalität herauszustellen, kümmern wir uns nicht um die Vergabe von Rechten und wählen einfach eine sehr niedrige Sicherheitsstufe. Tragen Sie am besten alle Benutzer des Testsystems als *mdw_admin* ein. In Produktivsystemen ist dies selbstverständlich nicht empfehlenswert, aber für unsere Versuche mit dem VDWH ist es durchaus in Ordnung. Wenn Sie den Assistenten beenden, können Sie anschließend die erstellte Datenbank im Objekt-Explorer sehen. Die Datenbank enthält bereits eine Reihe von Tabellen, Sichten, gespeicherten Prozeduren und Funktionen, die alle verwendet werden, um Daten zu sammeln und abzufragen.

4.12.2 Konfigurieren von Datenauflistungen

Nachdem nun die VDWH-Datenbank erzeugt wurde, existiert zunächst einmal die generelle Infrastruktur für das Sammeln von Daten. Datensammlungen an sich werden allein durch die Konfiguration des VDWH noch nicht durchgeführt. Hierfür müssen zunächst noch entsprechende Datenauflistungen in einem separaten Schritt konfiguriert werden. Bevor wir dies erledigen, sind jedoch noch einige einführende Bemerkungen zu dieser Thematik nötig.

Zunächst einmal ist das VDWH sicherlich auf längere Sicht – und damit meine ich über mehrere SQL Server-Versionen hinweg – dafür ausgelegt, möglichst alle relevanten Informationen an zentraler Stelle zu speichern. Diese Informationen beziehen sich dabei nicht nur auf Leistungsdaten, sondern werden sicherlich in der Zukunft noch erweitert werden, zum Beispiel um Informationen zur Konfiguration. Daher wurde der ursprüngliche Name wohl auch noch einmal von Performance-Data Warehouse in Verwaltungs-Data Warehouse geändert. Im Augenblick allerdings sind die standardmäßig vorhandenen Möglichkeiten vor allem auf die Messung und Speicherung von Performance-Daten ausgelegt.

Wenn Sie eine Datenauflistung konfigurieren, dann muss diese Auflistung einem der in SQL Server vordefinierten Auflistertypen entsprechen. Im Moment stehen Ihnen die folgenden Auflistertypen zur Verfügung:

- **T-SQL-Abfrageauflistertyp**. Dies ist unbestritten mein Lieblings-Auflistertyp. Er ermöglicht das Ausführen einer beliebigen T-SQL-Abfrage, wobei das Ergebnis dieser Abfrage letztlich in das VDWH übertragen wird. Dadurch existiert zum Beispiel die Möglichkeit, die Ergebnisse beliebiger dynamischer Verwaltungssichten, die ja ansonsten flüchtig sind, permanent zu speichern. Da die dynamischen Verwaltungssichten sehr umfangreiche Informationen zu SQL Server-Instanzen oder auch zum Betriebssystem liefern, eröffnet dieser Auflistertyp nahezu unbegrenzte Möglichkeiten. Gleichzeitig werden Sie hierdurch auch entsprechend motiviert, sich mit den dynamischen Verwaltungssichten auseinanderzusetzen. Denn diese sollten Sie einigermaßen beherrschen, sofern Sie eigene Datenauflistungen dieses Auflistertyps konfigurieren möchten. In Kapitel 11 werden Sie hierzu ein Beispiel sehen.
- **SQL-Ablaufverfolgungs-Auflistertyp**. Wie Sie wahrscheinlich bereits anhand des Namens vermuten, können Sie diesen Auflistertyp zur Konfiguration von SQL Server-Ablaufverfolgungen verwenden. Hierbei erstellen und starten Sie im Prinzip eine serverseitige Ablaufverfolgung.
- **Leistungsindikatoren-Auflistertyp**. Dieser Auflistertyp kann zum Sammeln von Informationen über Leistungsindikatoren des Betriebssystems und auch von SQL Server eigenen Leistungsindikatoren verwendet werden.
- **Abfrageaktivitäts-Auflistertyp**. Der Auflistertyp untersucht zunächst statistische Werte ausgeführter Abfragen. Sobald eine Abfrage in einem untersuchten Bereich bestimmte Schwellwerte über- bzw. unterschreitet, werden die Daten dieser Abfrage im VDWH gespeichert. Die hierfür relevanten Messwerte sind zum Beispiel die Anzahl der aktuell offenen Transaktionen oder die benötigte CPU-Zeit. Sobald mehr als acht offene Transaktionen existieren oder eine Abfrage mehr als fünf Sekunden CPU-Zeit benötigt, werden die entsprechenden Daten der Abfrage in das Protokoll aufgenommen. Dies ist nur ein Beispiel. Es gibt noch eine Reihe weiterer Kriterien, die Sie in der Dokumentation nachlesen können.

Es ist sicherlich zu erwarten, dass diese Auflistertypen in zukünftigen SQL Server-Versionen erweitert werden, sodass sich die Möglichkeiten der Messung und Protokollierung tatsächlich dahingehend entwickeln werden, dass das VDWH möglichst die kompletten Informationen über SQL Server-Performance, Probleme, Konfigurationen und Ähnliches aufnehmen kann.

Ich hoffe, dass es dann auch komfortablere Möglichkeiten für die Konfiguration einer Datenauflistung gibt, als dies im Moment der Fall ist. Das Management Studio unterstützt nämlich derzeit das Anlegen einer benutzerdefinierten Datenauflistung nicht. Falls Sie eine solche Datenauflistung einrichten möchten, funktioniert dies momentan nur über den Aufruf einer Reihe gespeicherter Prozeduren, die allesamt in der Systemdatenbank *msdb* vorhanden sind. Die sicherlich wichtigste Prozedur ist hierbei `sp_syscollector_create_collection_item`. An diese Prozedur wird unter anderem auch die Konfiguration der zu erzeugenden Auflistung übergeben. Diese Konfiguration ist – wie könnte es wohl anders sein – ein XML-Dokument. Sie werden also nicht umhinkommen, sich mit der entsprechenden XML-Syntax zu befassen, sobald Sie eigene Auflistungen erstellen. In Kapitel 11 erfahren Sie, wie Sie hierfür vorgehen. Dort erfahren Sie auch ein wenig mehr über die anderen Prozeduren, die Ihnen für eine Konfiguration benutzerdefinierter Datensammlungen zur Verfügung stehen.

Der SQL Server Profiler bildet hier eine positive Ausnahme. In Abschnitt 4.3.4 haben Sie bereits gesehen, wie Sie mit dem Profiler ein T-SQL-Skript erstellen, das eine serverseitige Ablaufverfolgung erstellt und startet. Auf genau demselben Weg können Sie auch ein Skript für das Erstellen einer Datensammlung für die konfigurierte Ablaufverfolgung erstellen. Wählen Sie hierzu aus dem Menü DATEI den Punkt EXPORTIEREN • SKRIPT FÜR ABLAUFVERFOLGUNGSDEFINITION ERSTELLEN • FÜR DEN AUFLISTSATZ DER SQL_ABLAUFVERFOLGUNG Das so erzeugte T-SQL-Skript können Sie anschließend ausführen und so die Datensammlung starten.

Glücklicherweise müssen Sie aber nicht unbedingt einen eigenen Auflistsatz definieren, wenn Sie SQL Server-Leistungsdaten unter Verwendung einer Datenauflistung protokollieren möchten. SQL Server kommt bereits mit vordefinierten sogenannten Systemdaten-Auflistsätzen daher. Diese Auflistsätze enthalten ebenfalls vordefinierte Auflistelemente, in denen im Prinzip beschrieben ist, welche Daten im VDWH protokolliert werden sollen. Durch diese vordefinierten Vorlagen ist das Konfigurieren einer Datenauflistung relativ einfach. Im Moment bietet SQL Server die folgenden drei vordefinierten Systemdaten-Auflistsätze an:

- **Auflistsatz für Serveraktivität**. Dieser Auflistsatz erfasst Leistungsdaten zu SQL Server sowie die Verwendung von Ressourcen durch SQL Server. Darüber hinaus können Sie mit diesem Auflistsatz auch Systemressourcen außerhalb von SQL Server überwachen. Hierzu werden sowohl Leistungsindikatoren des Systems als auch von SQL Server selber abgefragt und protokolliert. Außerdem fließen die Ergebnisse diverser dynamischer Verwaltungssichten in das Protokoll ein. Hierzu zählen zum Beispiel die Sichten `sys.dm_exec_sessions`, `sys.dm_exec_requests`, `sys.dm_os_waiting_tasks` und `sys.dm_os_wait_stats`. Dieser Auflistsatz ist sehr gut geeignet, um einen schnellen Überblick über das Gesamtsystem zu erhalten.
- **Auflistsatz für Datenträgerverwendung**. Der Auflistsatz ermittelt Datenbank- und dateibezogene Größen in Bezug auf Wachstumsraten von Dateien und Datenbanken

sowie die Datenträgerverwendung. Hierzu werden ebenfalls dynamische Verwaltungssichten abgefragt, zum Beispiel `sys.dm_io_virtual_file_stats` und `sys.partitions`.

- **Auflistsatz für Abfragestatistiken**. Über diesen Auflistsatz wird vor allem die dynamische Verwaltungssicht `sys.dm_exec_query_stats` abgefragt. Abhängig vom Ergebnis dieser Abfrage werden dann in Folge sowohl die Abfragestatistiken als auch der Abfragetext nebst dem Ausführungsplan der Abfragen mit dem größten Ressourcenverbrauch in einer Kategorie ermittelt und protokolliert. Für den Ressourcenverbrauch werden hierbei beispielsweise die Ausführungszeit sowie die erforderlichen Lese- und Schreibvorgänge betrachtet.

Prinzipiell werden die in einer Datenauflistung enthaltenen Daten nach einem festgelegten Zeitplan gemessen und in das VDWH geladen. Allerdings gibt es hierbei eine kleine Besonderheit. Das Messen und Hochladen kann nach verschiedenen Zeitplänen, also letztlich mit unterschiedlicher Häufigkeit, erfolgen. Es ist möglich und auch empfehlenswert, die gemessenen Daten nicht sofort hochzuladen, sondern zunächst zwischenzuspeichern. Der verwendete Zwischenspeicher ist hierbei einfach ein Verzeichnis auf dem lokalen Server. Diese Verfahrensweise der Entkopplung von Messung und Speicherung im VDWH bewirkt letztlich, dass der SQL Server, welcher das VDWH beherbergt, nicht allzu sehr belastet wird.

Betrachten wir als Beispiel den Auflistsatz für die Abfragestatistiken. Die Messung der in diesem Auflistsatz vorhandenen Elemente erfolgt alle zehn Sekunden, das Hochladen in das VDWH aber nur alle fünfzehn Minuten. Dadurch erhalten Sie eine sehr feine granulare Auskunft, belasten aber das VDWH nicht allzu sehr. Ein Hochladen, das synchron mit der Messung, also alle zehn Sekunden erfolgen würde, möglicherweise auch noch von diversen zu beobachtenden SQL Servern, ist sicherlich nicht zu empfehlen. Auch für Datensammlungen des Typs SQL-Ablaufverfolgung ist diese Entkopplung sehr sinnvoll. Die Ablaufverfolgungsdaten werden zunächst lokal in Dateien gespeichert und dann in größeren Abständen in das VDWH übertragen.

Nach dieser kurzen Einführung wollen wir nun die systemeigenen Auflistsätze in Betrieb nehmen und dafür sorgen, dass die entsprechenden Daten in das VDWH geladen werden.

Achten Sie bitte darauf, dass der SQL Server Agent gestartet sein ist, wenn Sie die Einrichtung der Datenauflistung vornehmen.

Für das Einrichten der Datenauflistung können Sie wieder einen Assistenten verwenden, den Sie über das Kontextmenü starten, so wie bereits in Abbildung 4.30 gezeigt. Wählen Sie diesmal die Option DATENAUFLISTUNG EINRICHTEN auf der zweiten Seite des Assistenten (siehe Abbildung 4.31). Im nächsten Schritt werden Sie dann aufgefordert, ein lokales Cache-Verzeichnis für das temporäre Zwischenspeichern der gesammelten Daten anzugeben. Dieses Verzeichnis muss existieren, andernfalls erhalten Sie später beim Versuch, Daten in den Cache zu schreiben, eine Fehlermeldung. Wählen Sie also ein geeignetes Verzeichnis aus und beenden Sie den Assistenten. Der Assistent aktiviert daraufhin die Datenauflistung und startet die Systemdaten-Auflistsätze zur Messung der Datenträgerverwendung und Serveraktivität. Im Objekt-Explorer finden Sie nach erfolgreicher Einrichtung die entsprechenden Einträge (Abbildung 4.32).

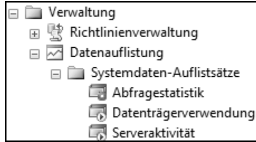


Abbildung 4.32: *Systemauflistsätze nach erfolgreicher Einrichtung der Datenauflistung*

Wenn Sie den Auflistsatz für die Abfragestatistik ebenfalls starten möchten, können Sie dies wie gewohnt über das Kontextmenü tun.

Der Assistent erledigt eine Menge weiterer Konfigurationen im Hintergrund. So werden zum Beispiel entsprechende Aufträge für den SQL Server Agent angelegt und auch SQL Server Integration Services (SSIS)-Pakete für das Speichern der Daten im Cache und das Hochladen in das VDWH erzeugt.

Wenn ein Auflistsatz einmal eingerichtet ist, können Sie die Eigenschaften durch das SQL Server Management Studio bearbeiten. Den zugehörigen Eigenschaften-Dialog öffnen Sie wie gewohnt aus dem Kontextmenü heraus. In Abbildung 4.33 sehen Sie ein Beispiel für den Auflistsatz zur Serveraktivität.

Abbildung 4.33: *Der Systemdaten-Auflistsatz »Serveraktivität«*

Über den Dialog können Sie insbesondere die Eigenschaften eines Auflistsatzes ansehen. Eine Konfiguration ist nur bedingt möglich. Die einzigen Parameter, die Sie tatsächlich ändern können, sind diejenigen zu den Zeitplänen für das Auflisten und Hochladen in das VDWH sowie die Dauer, während der die entsprechenden Daten im VDWH aufgehoben werden sollen. Nicht möglich ist hingegen das Hinzufügen, Ändern oder Entfernen von Auflistelementen. Diese Aufgabe müssen Sie momentan noch über T-SQL erledigen, indem Sie die entsprechenden gespeicherten Prozeduren aus der Systemdatenbank *msdb* aufrufen. Sie können sich jedoch (im Bereich EINGABEPARAMETER) die in den einzelnen Auflistelementen enthaltenen Parameter ansehen. In Abbildung 4.33 ist nur ein Teil dieser Informationen erkennbar; die Liste enthält jedoch jeweils alle Parameter für das ausgewählte Auflistelement.

Eine Datensammlung wird durch einen SQL Server Agent-Auftrag gestartet. Der Agent-Auftrag verwendet hierfür eine spezielle Laufzeitkomponente, die *Data Collector Run-Time Component DCEXEC.EXE*. Dieses Programm führt letztlich die SSIS-Pakete zum Zwischenspeichern und Hochladen der konfigurierten Daten aus. Wundern Sie sich also bitte nicht, wenn Sie auf Ihrem System ein Programm *DCEXEC.EXE* in der Liste der laufenden Prozesse finden.

Damit soll die Einführung in Datenauflistungen nun zunächst abgeschlossen werden. Sie werden in Kapitel 11 praktische Anwendungsbeispiele finden und auch sehen, wie benutzerdefinierte Auflistsätze konfiguriert und ausgewertet werden.

Nachdem Sie jetzt also wissen, wie Daten gesammelt werden, stellt sich die interessante Frage, wie diese gesammelten Daten ausgewertet werden können. Zunächst einmal haben Sie die Möglichkeit, die Tabellen und Sichten des VDWH abzufragen und die erhaltenen Daten zu interpretieren. Dazu müssen Sie allerdings den internen Aufbau des VDWH wenigstens teilweise kennen. Es gibt aber eine sehr viel elegantere Möglichkeit zur Auswertung der im VDWH gespeicherten Daten, und damit beschäftigen wir uns im nun folgenden Abschnitt.

4.13 Berichte

Wenn Sie sich gefragt haben, wie Sie es jemals schaffen sollen, die etwa 140 dynamischen Verwaltungssichten zu beherrschen, dann stehen Sie mit dieser Frage ganz bestimmt nicht alleine da. Sicherlich ermöglichen diese Sichten einen tieferen Einblick in die Arbeitsweise von SQL Server und erlauben eine detaillierte Analyse auftretender Probleme. Aus diesem Grund wird es Ihnen auf Dauer ganz bestimmt nicht erspart bleiben, wenigstens einige dieser Verwaltungssichten so weit kennenzulernen, dass Sie sie erfolgreich für eine Analyse einsetzen können. Warum aber nicht noch einen Schritt weiter gehen und gleich entsprechende Werkzeuge für die Auswertung der von den dynamischen Verwaltungssichten zurückgegebenen Daten erstellen und verwenden? Diese Frage hat man sich ganz offensichtlich auch bei Microsoft gestellt. Bereits seit der SQL Server-Version 2005 gibt es eine Reihe von Berichten, die über das Management Studio aufgerufen werden können. Diese Berichte sind für eine Analyse des momentanen und

des vergangenen Systemzustandes ganz hervorragend geeignet. Letztlich kommen die Daten für die Berichte größtenteils aus Systemsichten, wobei nicht nur dynamische Verwaltungssichten, sondern auch »normale«, d.h. mehr statische, Systemsichten aufgerufen werden.

Sie können beobachten, wie ein Bericht seine Daten zusammensammelt, indem Sie vor der Berichtserstellung eine SQL Server Profiler-Ablaufverfolgung starten. Dies ist eine sehr gute Möglichkeit, etwas mehr über die SQL Server-Systemsichten zu erfahren.

Im Objekt-Explorer finden Sie zu fast jedem Knoten entsprechende Berichte, die Sie über das Kontextmenü starten können. Eine Aufzählung aller verfügbaren Berichte würde den Rahmen dieses Buches gewiss sprengen. Ich möchte Ihnen daher an dieser Stelle lediglich einige ausgewählte Berichte präsentieren und Ihnen zeigen, wie Sie generell mit Berichten arbeiten. Es lohnt sich unbedingt, dass Sie sich die vorhandenen Berichte einmal ansehen und sie ausprobieren.

Eine sehr faszinierende Möglichkeit eröffnet sich Ihnen auch dadurch, dass Sie die vorhandenen Berichte um eigene Berichte ergänzen können. Dies ist gar nicht so kompliziert, wie Sie vielleicht vermuten. Sie benötigen hierfür lediglich einige zusätzliche Kenntnisse zum Berichts-Design mit Visual Studio (und natürlich entsprechendes Wissen, wie Sie an die Daten für Ihre selbst erstellten Berichte kommen; also doch wieder Einblick in die System- und Verwaltungssichten).

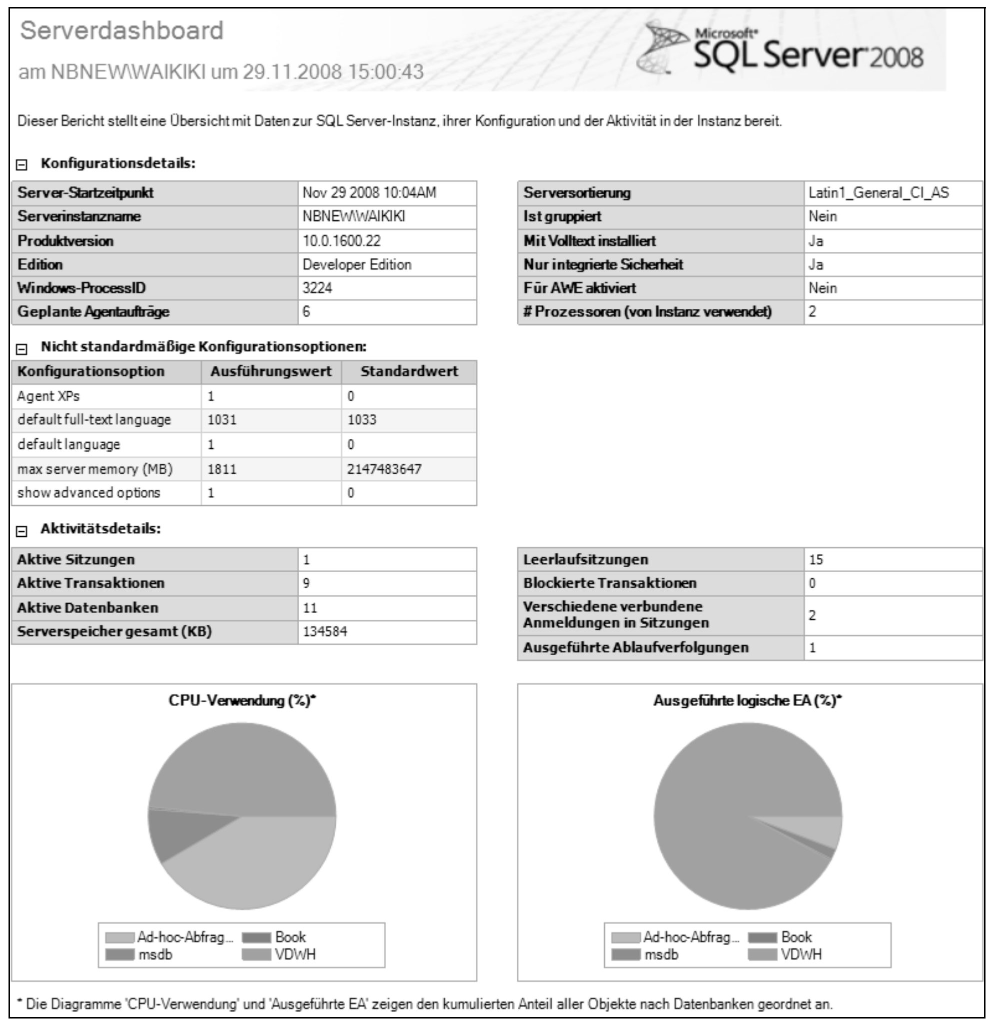
Die Berichte, welche ich Ihnen in diesem Abschnitt kurz vorstellen möchte, werden in zwei Kategorien eingeteilt. Zunächst erfahren Sie im Abschnitt 4.13.1 etwas über diverse Standardberichte, die Ihnen zur Verfügung stehen. Daran anschließend lernen Sie im Abschnitt 4.13.2 die Berichte kennen, die Ihnen der Datenauflister anbietet.

4.13.1 Allgemeine Berichte

Nach der Installation von SQL Server stehen Ihnen etwa 30 Berichte zum Ermitteln von Informationen über die SQL Server-Instanz und den SQL Server Agent zur Verfügung. Hinzu kommen noch knapp 20 Berichte für jede existierende Benutzer- und Systemdatenbank. Diese Berichte erlauben einen wirklich tiefen Einblick in die Interna von SQL Server. Sie können einen Bericht (nur) aus dem Kontextmenü für einen Ordner im Objekt-Explorer ausführen. Wählen Sie dort den Eintrag BERICHTE • STANDARDBERICHTE. Die jeweils zur Verfügung stehenden Berichte variieren dabei abhängig vom entsprechenden Knoten, für den Sie das Kontextmenü geöffnet haben. Für die SQL Server-Instanz selber werden zum Beispiel andere Berichte angeboten als für eine bestimmte Datenbank. In der folgenden Zusammenstellung finden Sie meine ganz persönlichen Favoriten für einige Bereiche.

Berichte für die SQL Server-Instanz

Serverdashboard Dieser Bericht (siehe Abbildung 4.34) ist vorzüglich für einen Gesamteindruck über den allgemeinen Gesundheitszustand Ihrer SQL Server-Instanz geeignet. Hier finden Sie wesentliche Parameter auf einen Blick, sodass Sie diesen Bericht als Ausgangspunkt für eine tiefergehende Analyse verwenden können.



Serverdashboard

am NBNEW\WAIKIKI um 29.11.2008 15:00:43

Microsoft SQL Server 2008

Dieser Bericht stellt eine Übersicht mit Daten zur SQL Server-Instanz, ihrer Konfiguration und der Aktivität in der Instanz bereit.

Konfigurationsdetails:

Server-Startzeitpunkt	Nov 29 2008 10:04AM
Serverinstanzname	NBNEW\WAIKIKI
Produktversion	10.0.1600.22
Edition	Developer Edition
Windows-ProcessID	3224
Geplante Agentaufträge	6

Serversortierung	Latin1_General_CI_AS
Ist gruppiert	Nein
Mit Volltext installiert	Ja
Nur integrierte Sicherheit	Ja
Für AWE aktiviert	Nein
# Prozessoren (von Instanz verwendet)	2

Nicht standardmäßige Konfigurationsoptionen:

Konfigurationsoption	Ausführungswert	Standardwert
Agent XPs	1	0
default full-text language	1031	1033
default language	1	0
max server memory (MB)	1811	2147483647
show advanced options	1	0

Aktivitätsdetails:

Aktive Sitzungen	1
Aktive Transaktionen	9
Aktive Datenbanken	11
Serverspeicher gesamt (KB)	134584

Leerlaufsitzungen	15
Blockierte Transaktionen	0
Verschiedene verbundene Anmeldungen in Sitzungen	2
Ausgeführte Ablaufverfolgungen	1

* Die Diagramme 'CPU-Verwendung' und 'Ausgeführte EA' zeigen den kumulierten Anteil aller Objekte nach Datenbanken geordnet an.

Abbildung 4.34: *Serverdashboard*

Ich mag besonders den Bereich *Nicht standardmäßige Konfigurationsoptionen*, in dem sofort erkennbar ist, ob wichtige Optionen verändert wurden. In Abbildung 4.34 erkennen Sie zum Beispiel sofort, dass der dem SQL Server zur Verfügung gestellte Hauptspeicher auf 1.811 MByte begrenzt wurde.

Arbeitsspeichernutzung Dieser Bericht stellt die momentane Verwendung des Arbeitsspeichers durch die verschiedenen Arbeitsspeicherclerks (dies ist tatsächlich die offizielle Übersetzung) dar. Darüber hinaus finden Sie hier auch Änderungen in der Arbeitsspeichernutzung über einen Zeitraum von sieben Tagen.

Besonders interessant ist der Bereich, der die Einordnung der im Cache vorhandenen Seiten angibt (Abbildung 4.35).

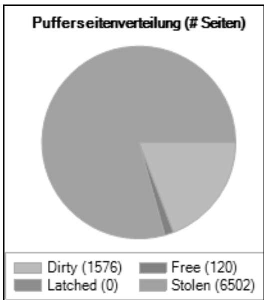


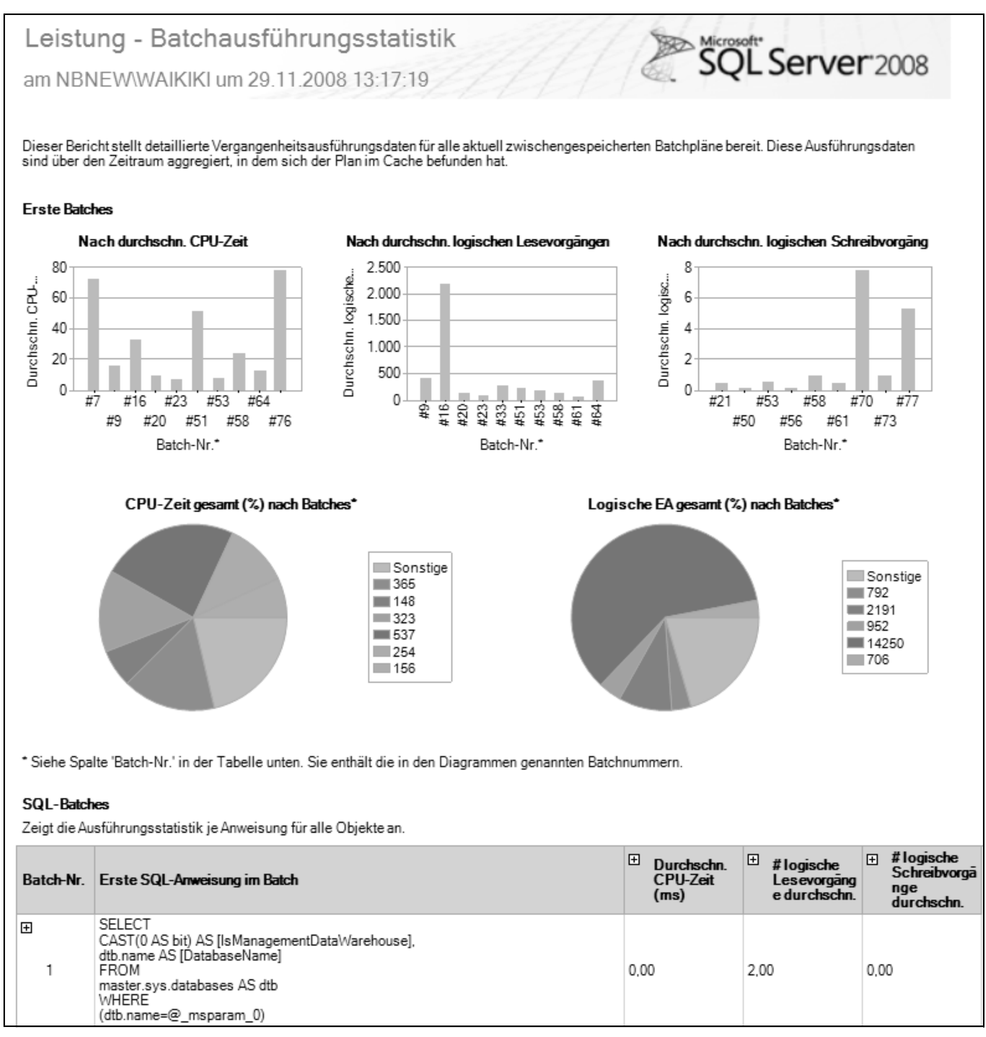
Abbildung 4.35: *Verteilung der im Cache gespeicherten Seiten*

Hinter jeder Kategorie finden Sie in Klammern die Anzahl der dieser Kategorie zugeordneten Datenseiten von jeweils 8 KByte Größe. Die dargestellten Kategorien haben folgende Bedeutung:

- **Dirty**. Diese Seiten wurden geändert, aber noch nicht permanent auf einem Datenträger gespeichert – zum Beispiel, weil noch kein `CHECKPOINT` erreicht wurde.
- **Free**. Dieser Bereich enthält Seiten, die momentan nicht verwendet werden, die also für das Speichern von Daten zur Verfügung stehen.
- **Latched**. Ein Latch ist eine Sperre auf physikalischer Ebene, also eine Sperre, welche die physikalische Datenintegrität sicherstellt. Seiten in dieser Kategorie sind momentan gesperrt, weil diese Seiten im Augenblick der Berichterstellung gerade an E/A-Operationen beteiligt sind. Ein hoher Anteil an gesperrten Seiten zeigt an, dass das System gerade stark E/A-belastet ist.
- **Stolen**. Das SQL Server-Betriebssystem (SQLOS) muss natürlich ebenso die vorhandenen Ressourcen verwalten wie jedes andere Betriebssystem auch. Dies gilt selbstverständlich gleichermaßen für den Hauptspeicher, der ja unter verschiedenen, möglicherweise konkurrierenden, internen Prozessen mehr oder weniger gerecht aufgeteilt werden muss. *Stolen Pages* kennzeichnet Seiten, die vom Daten-Cache an andere Bereiche abgegeben werden mussten, weil die zugehörigen Prozesse ebenfalls Hauptspeicher benötigen. Dies kann zum Beispiel erforderlich sein, wenn Sortier- oder Hash-Operationen ausgeführt werden, also eine Abfrage Arbeitsspeicher für die Ausführung benötigt. Der entsprechende Hauptspeicher wird dann einfach vom Daten-Cache »gestohlen«. Auch der für den Plan Cache verwendete Speicher wird in dieser Kategorie geführt. Ein hoher prozentualer Anteil an gestohlenen Seiten gegenüber freien und gepufferten Seiten ist ein klarer Beleg dafür, dass der dem SQL Server zur Verfügung stehende Hauptspeicher zu gering ist.

Dieser Bereich des Berichts hat möglicherweise noch ein kleines Problem. Was in dem Diagramm letztlich dargestellt werden soll, ist die Rückgabe des Kommandos `DBCC MEMORYSTATUS`. Wenn Sie sich einmal den Bereich *Buffer Pool* dieses Kommandos ansehen, dann finden Sie, dass deutlich mehr Informationen zurückgegeben werden, als im Bericht angezeigt werden. Insbesondere fehlt im Bericht der Bereich für den Daten-Cache mit nicht geänderten Seiten.

Leistung – Batchausführungsstatistik Dieser Bericht, für den Sie in Abbildung 4.36 ein Beispiel sehen, wertet den Plan Cache aus und gibt die Abfragen mit dem größten Verbrauch an Ressourcen zurück.



Batch-Nr.	Erste SQL-Anweisung im Batch	Durchschn. CPU-Zeit (ms)	# logische Lesevorgänge durchschn.	# logische Schreibvorgänge durchschn.
1	SELECT CAST(0 AS bit) AS [IsManagementDataWarehouse], dtb.name AS [DatabaseName] FROM master.sys.databases AS dtb WHERE (dtb.name=@_msparam_0)	0,00	2,00	0,00

Abbildung 4.36: *Serverbericht Leistung – Batchausführungsstatistik*

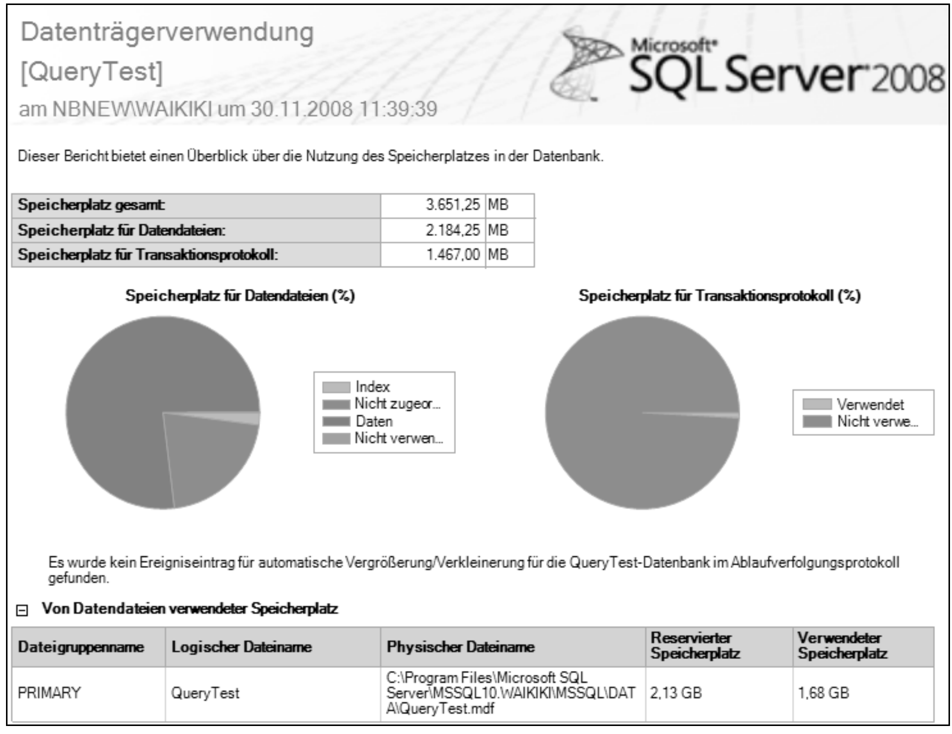
In der Tabelle im unteren Bereich erhalten Sie für die dargestellten Abfragen weitere statistische Informationen, wenn Sie den entsprechenden Bereich aufklappen. Dadurch haben Sie die Möglichkeit, die möglicherweise problematischen Abfragen detailliert zu untersuchen.

Berichte des Verwaltungs-Knotens

Anzahl von Fehlern Dieser Bericht gibt die in der SQL Server-Instanz seit dem letzten Start aufgetretenen Fehler zurück. Dadurch können Sie sehr schnell feststellen, ob in Ihrer Instanz schwerwiegende Probleme existieren.

Berichte für eine Datenbank

Datenträgerverwendung Dies ist sicherlich der Lieblingsbericht der meisten Datenbank-Administratoren. Er zeigt für eine Datenbank die physikalische Verteilung auf den zugeordneten Speicherorten (also in der Regel Festplatten) an. In Abbildung 4.37 sehen Sie ein Beispiel für den Bericht *Datenträgerverwendung*.



Datenträgerverwendung
[QueryTest]
am NBNEW\WAIKIKI um 30.11.2008 11:39:39

Dieser Bericht bietet einen Überblick über die Nutzung des Speicherplatzes in der Datenbank.

Speicherplatz gesamt:	3.651,25	MB
Speicherplatz für Datendateien:	2.184,25	MB
Speicherplatz für Transaktionsprotokoll:	1.467,00	MB

Es wurde kein Ereigniseintrag für automatische Vergrößerung/Verkleinerung für die QueryTest-Datenbank im Ablaufverfolgungsprotokoll gefunden.

⊟ **Von Datendateien verwendeter Speicherplatz**

Dateigruppenname	Logischer Dateiname	Physischer Dateiname	Reservierter Speicherplatz	Verwendeter Speicherplatz
PRIMARY	QueryTest	C:\Program Files\Microsoft SQL Server\MSSQL10.WAIKIKI\MSSQL\DATA\QueryTest.mdf	2,13 GB	1,68 GB

Abbildung 4.37: *Datenträgerverwendung durch die Datenbank »QueryTest«*

Schemaänderungsverlauf Änderungen am Datenbankschema können oftmals Ursache sowohl für ein komplettes Fehlverhalten einer Anwendung als auch für Performance-Probleme sein. Der Bericht *Schemaänderungsverlauf* listet diese Änderungen auf, sodass Sie die Möglichkeit haben festzustellen, welche Änderung(en) am Datenbankschema eventuell verantwortlich für auftretende Probleme ist (sind).

4.13.2 Berichte der Datenauflistung

Viele Standardberichte zeigen lediglich Daten an, die seit dem letzten Start von SQL Server gesammelt wurden. Dies trifft zum Beispiel auf den Bericht *Leistung-Batchausführungsstatistik* zu, der im vorigen Abschnitt vorgestellt wurde. Mit der Einführung des Verwaltungs-Data Warehouse gibt es nun auch eine standardisierte Möglichkeit, solche Daten permanent zu speichern und über vordefinierte Berichte abzufragen. Die Ihnen zur Verfügung stehenden Berichte untergliedern sich hierbei in genau die drei Bereiche, in denen auch durch die Systemdaten-Auflistsätze Daten gemessen und protokolliert werden: Serveraktivität, Abfragestatistik und Datenträgerverwendung.

Ich kann Ihnen daher nur noch einmal empfehlen, dass Sie sich mit diesen Berichten auseinandersetzen. Ich möchte Ihnen die Berichte an dieser Stelle kurz vorstellen. Hierbei ist es etwas schwierig, diese Berichte entsprechend zu beschreiben, einfach deshalb, weil die Berichte interaktiv sind, und Navigationsmöglichkeiten, wie zum Beispiel das Aufrufen von Unterberichten durch einen Mausklick, beinhalten. So etwas ist in einem Text nur schwer zu vermitteln, deshalb sollten Sie es wirklich unbedingt einmal selbst ausprobieren.

Die zur Abfrage des Verwaltungs-Data Warehouse existierenden Berichte starten Sie aus dem Kontextmenü des Knotens VERWALTUNG • DATENAUFLISTUNG.

Serveraktivität – Verlauf

Beim Aufruf dieses Berichts bekommen Sie zunächst eine Gesamtübersicht über die Ressourcenverwendung Ihrer SQL Server-Instanz, aufgetragen über die Zeit. In Abbildung 4.38 sehen Sie ein Beispiel.

Die dargestellte Information ermöglicht bereits einen sehr guten Überblick über aufgetretene Engpässe oder Probleme. Wirklich exzellent ist jedoch die Möglichkeit, weiter in die Tiefe zu gehen. So können Sie zum Beispiel durch einen Klick auf einen bestimmten Zeitpunkt in der Kurve der CPU-Verwendung an detailliertere Informationen über die Verwendung der Prozessoren gelangen. Interessant ist auch die Möglichkeit, das Zeitfenster für die Betrachtung auszuwählen. Hierzu verwenden Sie die im oberen Bereich dargestellte Zeitachse. Sobald Sie dort auf einen bestimmten Zeitbereich klicken, wird der Bericht nur für diesen Zeitabschnitt neu erstellt.

Ich vermute, dass Sie ziemlich begeistert sein werden, wenn Sie ein wenig mit dem Bericht experimentieren. Sie bekommen ein wirklich sehr mächtiges Werkzeug in die Hand, das einfach zu bedienen ist, und weitreichende Möglichkeiten für eine Analyse bietet.

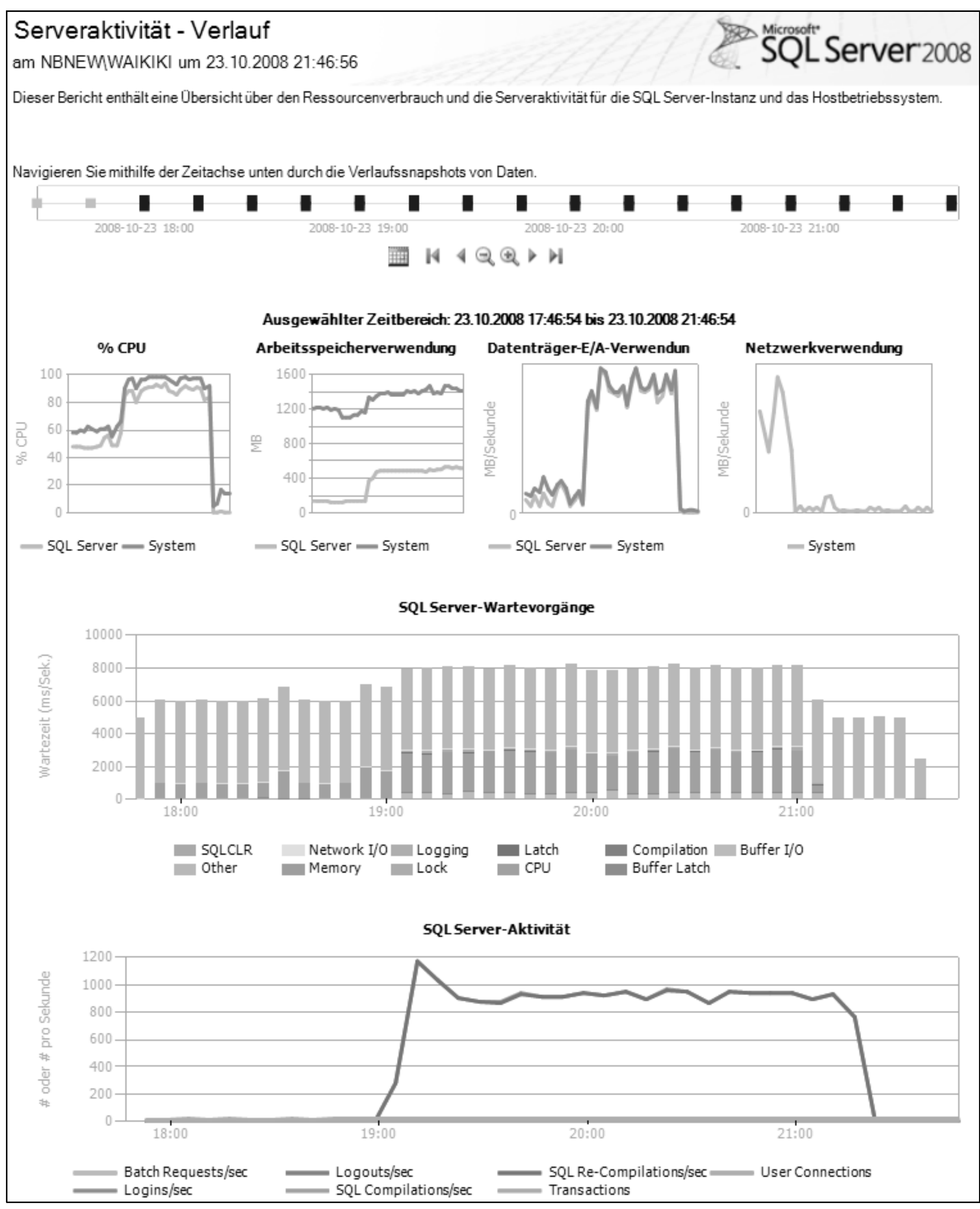


Abbildung 4.38: *Der Bericht »Serveraktivität – Verlauf«*

Datenträgerverwendung

Mit diesem Bericht können Sie für alle Datenbanken die Datenträgerverwendung über die Zeit hinweg beobachten. Ein Beispiel für den Bericht sehen Sie in Abbildung 4.39. (Beachten Sie bitte, dass der Bericht ganz offensichtlich noch nicht ins Deutsche übersetzt wurde.)

Disk Usage Collection Set

am NBNEW\WAIKIKI um 11/30/2008 12:21:23 PM

Microsoft® SQL Server 2008

This report provides an overview of the disk space used for all databases on the server and growth trends for the data file and the log file for each database for the last 2 collection points between 11/29/2008 12:00:07 PM and 11/30/2008 12:00:05 PM.

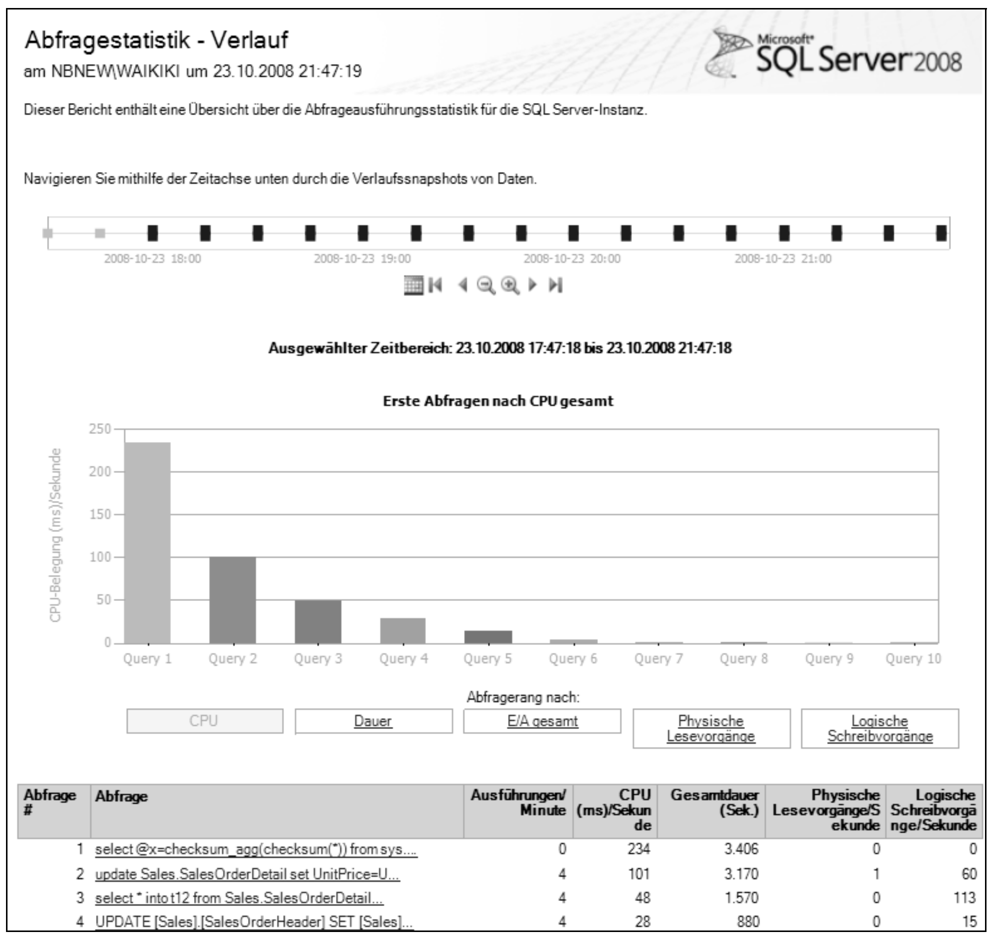
	Datenbank				Protokoll			
Datenbankname	Anfangsgröße (MB)	Trend	Aktuelle Größe (MB)	Durchschnittliche Vergrößerung (MB/Tag)	Anfangsgröße (MB)	Trend	Aktuelle Größe (MB)	Durchschnittliche Vergrößerung (MB/Tag)
AdventureWorks2008	196.06		196.06	0	66.00		66.00	0
Book	2.19		2.19	0	3.44		3.44	0
master	277.63		277.63	0	417.38		417.38	0
model	2.25		2.25	0	0.75		0.75	0
msdb	12.75		12.75	0	2.75		2.75	0
QueryTest	2,184.25		2,184.25	0	1,467.00		1,467.00	0
ReportServer$WAIKIKI	3.25		3.25	0	6.25		6.25	0
ReportServer$WAIKIKITempDB	2.25		2.25	0	0.81		0.81	0
Skripts	2.19		2.19	0	1.00		1.00	0
tempdb	14.88		14.88	0	1.00		0.75	(0.25)
VDWH	100.00		100.00	0	10.00		10.00	0

Abbildung 4.39: *Bericht über die Datenträgerverwendung*

Der Bericht zeigt sowohl die Veränderungen in den Daten- als auch in Protokolldateien an. Auch hier können Sie durch einen einfachen Mausklick in den entsprechenden Bereich detailliertere Informationen erhalten.

Abfragestatistik – Verlauf

Dieser Bericht listet die Top-Abfragen nach ihremRessourcenverbrauch auf. In Abbildung 4.40 sehen Sie ein Beispielergebnis.



Abfrage #	Abfrage	Ausführungen/ Minute	CPU (ms)/Sekunde	Gesamtdauer (Sek.)	Physische Lesevorgänge/Sekunde	Logische Schreibvorgänge/Sekunde
1	select @x=checksum_agg(checksum(*)) from sys....	0	234	3.406	0	0
2	update Sales.SalesOrderDetail set UnitPrice=U...	4	101	3.170	1	60
3	select * into t12 from Sales.SalesOrderDetail...	4	48	1.570	0	113
4	UPDATE [Sales].[SalesOrderHeader] SET [Sales]...	4	28	880	0	15

Abbildung 4.40: *Abfragestatistik – Verlauf*

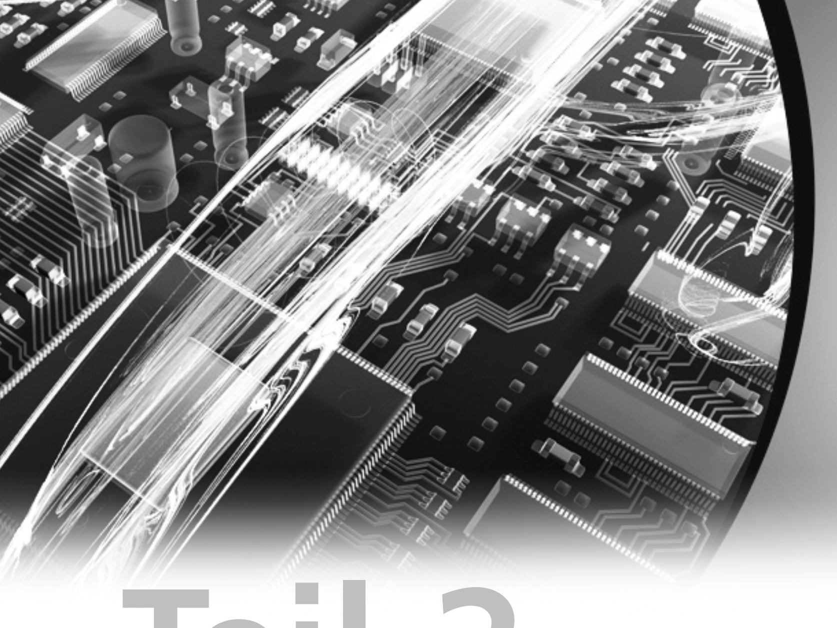
Auch hier gibt es wieder die Möglichkeit der Navigation bzw. Interaktivität. So können Sie zum Beispiel auswählen, nach welcher Ressource (CPU, Dauer, E/A) Sie Ihre Hitliste erstellen möchten. Durch einen Klick auf eine im unteren Bereich dargestellte Abfrage erhalten Sie nähere Informationen zu dieser Abfrage, darunter beispielsweise auch den grafischen Ausführungsplan. So können Sie fast mühelos untersuchen, warum die Abfrage in der Hitliste auftaucht, oder Ansatzpunkte für eine Verbesserung finden.

4.14 Zusammenfassung

In diesem Kapitel haben Sie die vielfältigen Möglichkeiten kennengelernt, die SQL Server für die Überwachung zur Verfügung stellt. Sie sollten nun in der Lage sein, aus den existierenden Möglichkeiten für die Überwachung eine geeignete auszuwählen, und so eventuellen Problemen auf den Grund zu gehen.

Falls Sie nach der Lektüre des Kapitels trotzdem noch etwas unsicher sind, welches Werkzeug Sie für welchen Einsatzfall bevorzugen sollten, dann ist das absolut verständlich. Ich kann Sie an dieser Stelle beruhigen, denn offensichtlich gibt es für zukünftige SQL Server-Versionen hier den klaren Trend, das Verwaltungs-Data Warehouse für die Überwachung und Analyse zu verwenden. Von daher sollten Sie sich auf die Beherrschung dieser Möglichkeit konzentrieren.

Sie werden in den nachfolgenden Kapiteln immer wieder die hier vorgestellten Werkzeuge antreffen und sehen, wie Sie spezifische Probleme angehen können. Mit anderen Worten: Das in diesem Kapitel geschaffene Grundwissen wird in den restlichen Kapiteln dieses Buches nach und nach vertieft werden.



Teil 2

Physische Aspekte des Datenbankentwurfs

5 Verwenden von Indizes 125

6 Verwalten von Indizes 143

7 Partitionierung 163

8 Komprimierung von Daten 171

5 Verwenden von Indizes

Bei der Abfrageoptimierung spielen Indizes eine ganz entscheidende Rolle. Die Erstellung geeigneter Indizes kann die Ausführung von Abfragen gleich um einige Größenordnungen beschleunigen und somit recht beeindruckende Auswirkungen auf die Abfrageleistung haben.

Es ist deshalb sehr wichtig, dass Sie die in diesem Kapitel vorgestellten Konzepte verstehen, um diese sinnvoll anzuwenden.

Wir werden uns zunächst damit auseinandersetzen, welche Möglichkeiten der Indizierung SQL Server zur Verfügung stellt. Hier ist etwas Theorie unerlässlich. Anschließend erfahren Sie im mehr praktisch orientierten Teil, wie Indizes erzeugt, gelöscht und reorganisiert werden. Methoden zum Auffinden geeigneter Indizes sind nicht Gegenstand dieses Kapitels. Hierfür wird noch etwas mehr Grundlagenwissen benötigt, also haben Sie bitte noch ein wenig Geduld. Die Kapitel 9 und 11 beschäftigen sich mit der Thematik, wie passende Indizes gefunden werden können.

5.1 Der Heap: Eine Tabelle ohne Index

Eine Tabelle ohne gruppierten Index wird auch als Heap (deutsch: Haufen) bezeichnet. Die Daten einer solchen Tabelle liegen in unsortierter Form, einfach als Blöcke, die nicht zueinander in Beziehung stehen, auf der Festplatte. Um auf die Tabellendaten zugreifen zu können, verwaltet SQL Server eine Zuordnungstabelle, die sogenannte Index Allocation Map (IAM). In dieser Tabelle werden Zeiger zu den einzelnen Seiten des Heap verwaltet, wobei ein solcher Zeiger die SQL Server-Datendatei sowie die Nummer der Seite innerhalb der Datendatei enthält. Ein Zeiger hat hierbei die Form `DateiID:SeitenNummer`. So bezeichnet zum Beispiel der Zeiger 1:7521 die Seite 7521 in der ersten Datendatei der entsprechenden Datenbank.

Für die weiteren Betrachtungen in diesem Kapitel verwenden wir eine Tabelle, die wir nach und nach mit Indizes ausstatten. Das folgende Skript legt diese Tabelle zunächst als Heap an:

```
use QueryTest;
-- Testtabelle erzeugen. Die dritte Spalte
-- soll hier nur simulieren, dass noch mehr Spalten
-- vorhanden sind.
if (object_id('T1') is not null)
  drop table t1
```

```
go
create table T1
 (
   Id int not null
  ,Nr int not null
  ,Platzhalter nchar(400) null default '#'
 )
go
-- Füge 200.000 Zeilen ein
insert T1 (Id,Nr)
  select n, n % 100 from Numbers
   where n <= 200000
```

Für die im Skript erzeugte Tabelle T1 werden erst einmal keine Indizes angelegt. Diese Tabelle existiert daher vorerst als Heap, das heißt, die einzelnen Datenseiten der Tabelle liegen in ungeordneter Reihenfolge auf dem Datenträger. Sie können sich dies in etwa so vorstellen, wie in Abbildung 5.1 gezeigt.

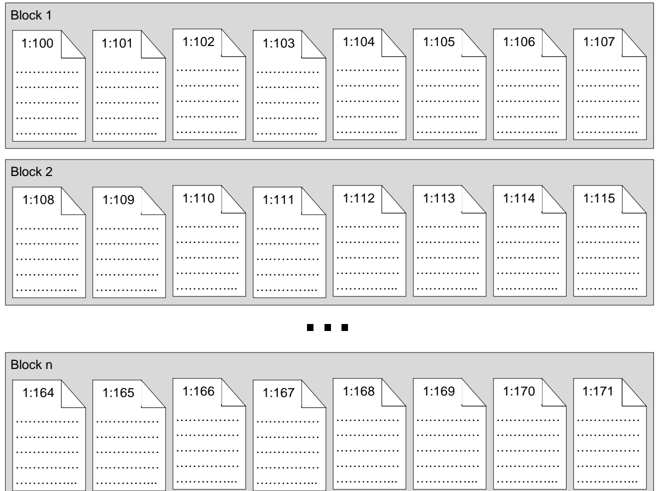


Abbildung 5.1: *Anordnung der Datenseiten in einem Heap*

Wenn nun eine Abfrage Daten aus der Tabelle T1 benötigt, dann müssen diese Daten im Heap gesucht werden. Hierzu muss – da ja keine Sortierung existiert – immer der gesamte Heap gelesen werden, ein Vorgang, der im Englischen als *Table Scan* bezeichnet wird.

Betrachten Sie als Beispiel bitte die folgende Abfrage:

```
select * from T1
  where Id=3331
```

Obwohl diese Abfrage lediglich eine Zeile zurückliefert, muss der gesamte Heap sequenziell durchsucht werden, um die in der WHERE-Klausel angegebene Bedingung zu überprüfen. Hierzu wird die Tabelle seiten- bzw. blockweise vom Datenträger oder – falls Datenseiten dort bereits existieren – auch aus dem Datencache gelesen, wobei tatsächlich alle Datenseiten (wenigstens) einmal gelesen werden müssen.

Es ist relativ einfach auszurechnen, wie viele Datenseiten letztlich für den Table Scan durchsucht werden. Jede Zeile unserer Tabelle benötigt 808 Byte (jeweils 4 Byte für jeden der beiden `INTEGER`-Werte und 800 Byte für die Spalte `Platzhalter`). Somit passen in eine Datenseite 8060 Byte / 808 Byte = 9 Zeilen der Tabelle. Bei insgesamt 200.000 Zeilen werden also 200.000 / 9 = 22.223 Seiten für die Tabelle benötigt. Das sind also insgesamt immerhin 22.223 × 8096 Byte, also etwa 170 Mbyte.

5.2 Der gruppierte Index

Ein gruppierter Index enthält in den Blattseiten des Indexbaums die Tabellendaten. Diese Daten liegen dort in sortierter Form vor, wobei die Sortierreihenfolge durch die Indexspalten festgelegt wird. Am besten lässt sich diese Aussage grafisch verdeutlichen.

Abbildung 5.2 zeigt einen Auszug aus dem Indexbaum für einen gruppierten Index auf der Spalte `ID` der Tabelle `T1`.

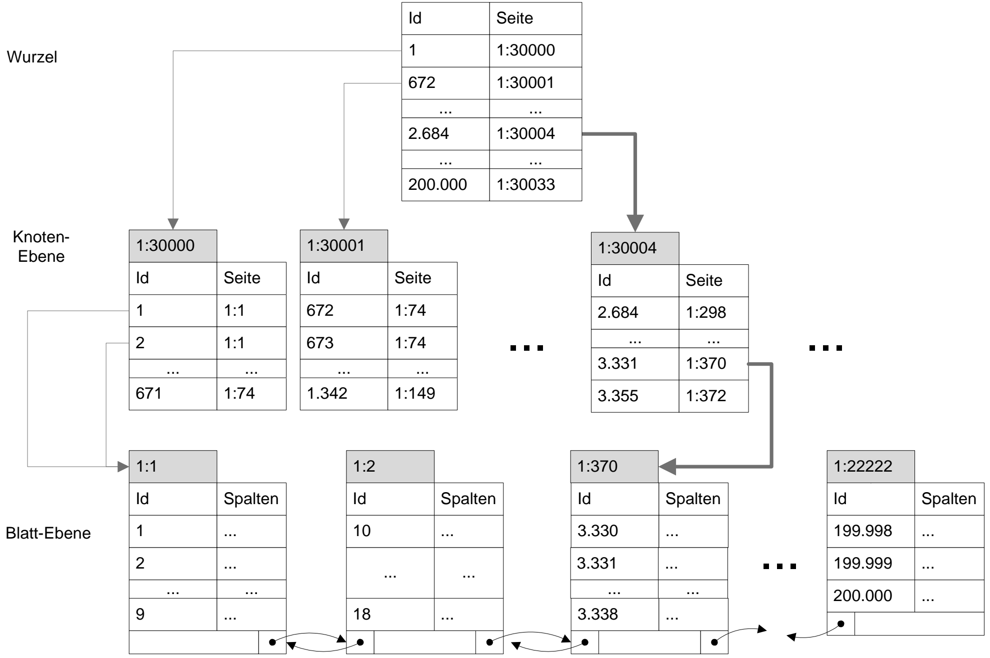


Abbildung 5.2: Gruppierter Index auf der Spalte »ID«

Jeder Index besitzt immer genau eine Datenseite, welche die Wurzel des Indexbaums verkörpert. Diese Wurzel enthält Verweise auf die erste Knotenebene, wobei am Indexwert in der Wurzel abgelesen werden kann, in welchem Knotenelement ein bestimmter Wert gesucht werden muss. Jeder Indexeintrag in der Wurzel und in der Knotenseite (also alle Einträge in Nicht-Blattseiten) enthält den Indexschlüssel und einen Verweis auf die in der untergeordneten Ebene relevante Seite. Dieser Zeiger besitzt wiederum die Form `DateiID:Seitennummer`. Die unterste Ebene des Index – also die Blattseite – enthält die eigentlichen Tabellendaten.

Die Blattseiten selber sind nochmals untereinander verkettet, und zwar in beide Richtungen. In jeder Blattseite existieren somit Verweise auf die vorhergehende sowie auf die nachfolgende Seite. Auch dies verdeutlicht Abbildung 5.2.

Ein Indexbaum ist also hierarchisch aufgebaut, was eine sehr schnelle Suche bestimmter Werte ermöglicht. In unserem Fall hat der Indexbaum drei Ebenen. Dadurch wird jeder beliebige Wert über die Suche im Indexbaum in nur drei Schritten gefunden.

Die dicken Pfeile in Abbildung 5.2 verdeutlichen eine Suche nach der ID 3331. Die Suche beginnt zunächst in der Wurzel des Index. Die Wurzel wird immer in einer einzelnen Datenseite gespeichert, die Verweise auf die Suchseiten in der nächsten Ebene enthält. Falls der Index klein ist, kann in der Wurzel auch bereits direkt auf die Blattseite verwiesen werden. In unserem Fall ist dies jedoch nicht so. Anhand der Indexwerte im Wurzelelement wird festgestellt, dass auf der Seite 1:30004 weitergesucht werden soll. Auf dieser Seite steht bereits, dass die Daten für die ID 3331 in der Blattseite 1:370 existieren – und damit ist die Suche nach dem dritten Schritt beendet.

Es ist übrigens relativ simpel, die Tiefe des Indexbaums auszurechnen. In der Blattebene benötigt der Index genau so viele Seiten, wie bereits im vorangegangenen Abschnitt berechnet, nämlich 22.223. In der darüber liegenden Ebene werden dann Verweise auf eben diese 22.223 Blattseiten benötigt. Jeder dieser Verweise speichert den eigentlichen Verweis auf die entsprechende Blattseite, aber natürlich auch den Wert des Indexschlüssels. Für den Verweis auf die Blattseite werden acht Byte benötigt. Hinzu kommt der Wert für den Index an dieser Stelle. Da unser Index nur eine INTEGER-Spalte verwendet, sind dies noch einmal vier Byte. Insgesamt sind also auf der ersten Nicht-Blattebene 12 Byte je Indexeintrag erforderlich. In einer Datenseite können somit 8060 Byte / 12 Byte = 671 Indexverweise gespeichert werden. Bei 22.223 erforderlichen Verweisen macht dies dann 34 Seiten. Auf die gleiche Art und Weise können Sie auch die weiteren benötigten Ebenen kalkulieren. In unserem Fall müssen auf der nächsten Ebene lediglich noch 34 Verweise auf die erste Nicht-Blattebene gespeichert werden, wofür eine einzelne Datenseite ausreicht. Unser Indexbaum hat also eine Tiefe von drei. Sofern dieser Index für die Suche nach einem Wert verwendet wird, kann sehr einfach über den Indexbaum navigiert werden. Somit sind lediglich noch drei Suchvorgänge erforderlich, also genau so viele, wie der Indexbaum Stufen besitzt. Vergleichen Sie dies einmal mit dem Table Scan aus dem vorherigen Beispiel. Dort waren 22.223 Leseoperationen erforderlich, mit einem geeigneten Index sind es nun lediglich noch drei; eine Verbesserung um ungefähr den Faktor 7.400.

Die Tiefe des Indexbaums bestimmt also die Anzahl der Leseoperationen. Da die Anzahl der Datenseiten je Indexebene exponentiell abnimmt, haben auch Indizes für sehr große Tabellen bzw. große Indizes mit vielen oder langen Spalten selten mehr als vier Ebenen. In unserem Fall würden zum Beispiel auch bei 2.000.000.000 Zeilen in der Tabelle `T1` noch vier Indexebenen ausreichen.

Sie müssen die Tiefe des Indexbaums übrigens nicht von Hand bestimmen. Die dynamische Systemsicht `sys.dm_db_index_physical_stats` (genauer gesagt, ist es eine Funktion) ermöglicht auch die Abfrage dieser (und noch einiger weiterer) Informationen zu Indi-

zes oder auch zu Heaps. Diese Funktion erwartet vier Parameter, mit denen bestimmt wird, welche Informationen zurückgeliefert werden sollen. Für unsere Tabelle T1 kann der Aufruf zum Beispiel so erfolgen:

```
select index_id,index_type_desc,index_depth
      ,index_level,page_count,record_count
  from sys.dm_db_index_physical_stats(db_id(),object_id('T1')
                                     ,null,null,'detailed')
```

Abbildung 5.3 zeigt ein Beispiel für die Ausgabe.

	index_id	index_type_desc	index_depth	index_level	page_count	record_count
1	1	CLUSTERED INDEX	3	0	22223	200000
2	1	CLUSTERED INDEX	3	1	36	22223
3	1	CLUSTERED INDEX	3	2	1	36

Abbildung 5.3: *Informationen zum gruppierten Index*

Die dort angezeigten Werte weichen etwas von unseren Berechnungen ab. Dies liegt einerseits daran, dass wir zusätzlichen Speicher (Overhead), der je Indexeintrag benötigt wird, vernachlässigt haben. So müssen ja zum Beispiel in den Blattseiten auch Zeiger für die Verkettung der Blattseiten selber existieren, die von uns der Einfachheit halber außer Acht gelassen wurden. Weiter sind wir bei unserer Berechnung davon ausgegangen, dass die Nicht-Blattseiten zu 100 Prozent gefüllt werden – eine Annahme, die nicht unbedingt immer korrekt ist. Prinzipiell treffen unsere Berechnungen jedoch durchaus zu.

Bereichssuchen über einen Index funktionieren ein wenig anders. Betrachten Sie bitte die folgende Abfrage:

```
select * from T1
 where id between 10001 and 10150
```

Hier für jeden Vergleich eine Suche über die komplette Tiefe des Indexbaums durchzuführen, wäre sehr ineffizient. Es gibt eine sehr viel bessere Lösung: Da die Blattseiten in sortierter Form vorliegen und außerdem aufeinander verweisen, wird einfach nur die erste Blattseite – also diejenige für ID = 10001 – über den Indexbaum ermittelt. Alle weiteren Blattseiten können dann einfach über die Navigation in der doppelt verketteten Liste gelesen werden, da diese Liste ja nach dem Schlüssel des Index sortiert ist. Sobald der Wert ID = 10151 erreicht ist, ist die Suche beendet.

Wichtig ist, dass Sie sich das Folgende einprägen:

Der gruppierte Index enthält nicht etwa eine Kopie der Tabelle, er *ist* die Tabelle. Sobald Sie einen gruppierten Index für eine Tabelle erzeugen, sind die Tabellendaten in den Blattseiten des gruppierten Index gespeichert. Aus diesem Grund ist je Tabelle auch nur ein einziger gruppierter Index erlaubt.

5.3 Der nichtgruppierte Index auf einem Heap

Eine Tabelle ohne gruppierten Index (ein Heap also) kann bis zu 249 nichtgruppierte Indizes besitzen. Auch der nichtgruppierte Index wird in Form eines Indexbaums verwaltet. Abbildung 5.4 zeigt ein Beispiel für einen nichtgruppierten Index auf der Spalte `ID`, wobei wir annehmen, dass die Tabelle keinen gruppierten Index besitzt.

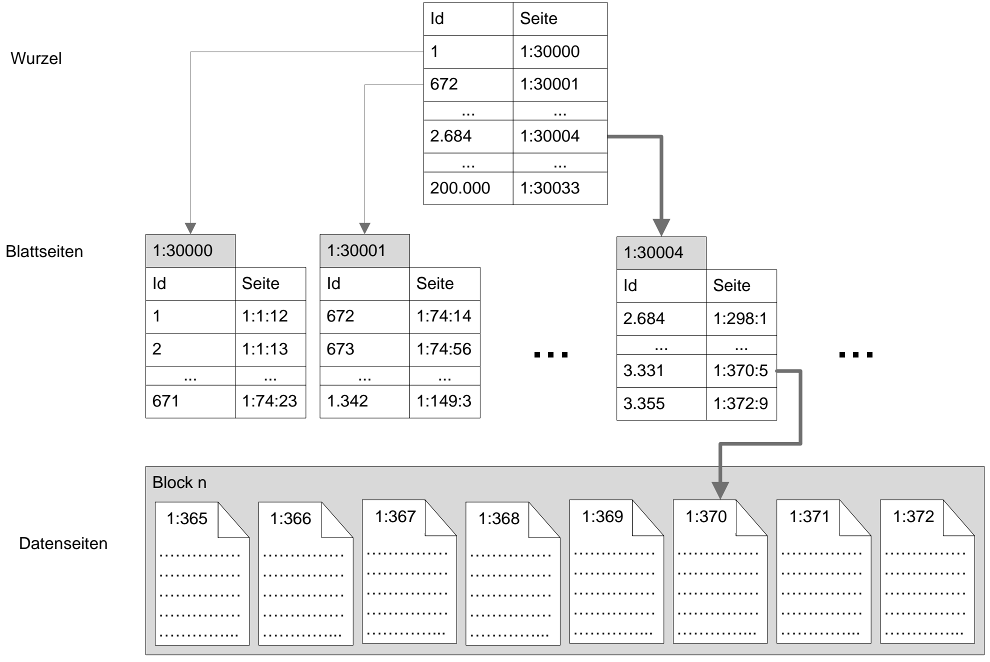


Abbildung 5.4: *Nichtgruppierter Index »T1.Id« auf einem Heap*

Der Unterschied zum gruppierten Index liegt in den Blattseiten. Hier sind nun nicht mehr die Tabellendaten selber gespeichert. Stattdessen existieren in den Blattseiten nur Verweise auf die Speicherorte der eigentlichen Tabellendaten. Jeder dieser Verweise ist in der Form `DateiID:DatenSeite:PositionImBlock` abgelegt. Die Abbildung verdeutlicht wiederum eine Suche nach der ID 3331. Die ersten beiden Suchschritte unterscheiden sich zunächst nicht von der Suche im gruppierten Index. Lediglich das eigentliche Holen der Tabellendaten, also der letzte Schritt, ist beim nichtgruppierten Index auf einem Heap anders. Die Tabellendaten werden nun über den eingetragenen Verweis auf die entsprechende Datenseite geladen, ein Vorgang, der im Englischen als *RowId-Lookup* bezeichnet wird.

Da der nichtgruppierte Index nur Verweise auf die eigentlichen Tabellendaten speichert, kommt er mit entsprechend weniger Datenseiten und meist auch weniger Stufen im Indexbaum aus. In unserem Beispiel hat der Index nur zwei Stufen, gegenüber drei beim gruppierten Index. In der Summe ist jedoch der benötigte Speicher für Index- und Tabellendaten in beiden Fällen etwa gleich.

5.4 Der nichtgruppierte Index auf einem gruppierten Index

Auch für eine Tabelle mit einem gruppierten Index können zusätzlich bis zu 249 nichtgruppierte Indizes erzeugt werden. Ein nichtgruppierter Index auf einem gruppierten Index unterscheidet sich allerdings in einem wesentlichen Punkt von einem nichtgruppierten Index auf einem Heap. Im nichtgruppierten Index auf einem gruppierten Index sind in den Blattseiten keine Verweise auf die eigentlichen Tabellendaten enthalten. Hier steht nun stattdessen der Schlüsselwert des gruppierten Index. Der Grund für diese Verfahrensweise ist leicht erklärt. Ändern sich Tabellendaten, dann betrifft dies auch die Blattseiten des gruppierten Index. In diesen Blattseiten können durch Tabellenänderungen Seiten entfernt, verschoben oder hinzugefügt werden. Würde der nichtgruppierte Index nun auf den physikalischen Speicherort der Tabellendaten (also die Blattseiten des gruppierten Index) verweisen, so müssten bei Datenänderungen am gruppierten Index stets auch alle nichtgruppierten Indizes angepasst werden – und das wäre ganz einfach zu kostspielig. Durch die Speicherung des Wertes für den gruppierten Index im nichtgruppierten Index wird dies vermieden. Diese Verfahrensweise bringt jedoch auch einen Nachteil mit sich. Da im nichtgruppierten Index lediglich der Schlüsselwert des gruppierten Index steht, können die eigentlichen Tabellendaten nicht über eine Suche im nichtgruppierten Index ermittelt werden. Die Suche über den nichtgruppierten Index liefert nun lediglich den Schlüsselwert des gruppierten Index. Mit diesem ermittelten Schlüsselwert wird dann der Indexbaum des gruppierten Index durchsucht.

Nehmen wir die folgende Abfrage:

```
select * from T1
 where Nr = 31
```

Angenommen, es existiert ein gruppierter Index für die Spalte `ID`. Um die in der Abfrage aufgelisteten Spalten (hier sind es alle) zurückzuliefern, müssen die Daten also über diesen Index ermittelt werden. Wir wollen hier zusätzlich davon ausgehen, dass auch ein nichtgruppierter Index auf der Spalte `Nr` existiert. Dieser Index kann für die Suche verwendet werden, da wir ja nur diejenigen Zeilen der Tabelle bekommen wollen, in denen der Wert der Spalte `Nr = 31` ist. Eine Suche über den nichtgruppierten Index läuft dann so ab wie in Abbildung 5.5. Der nichtgruppierte Index wird hier nur dazu verwendet, um die Schlüsselwerte für den gruppierten Index zu ermitteln. Diese Werte stehen in den Blattseiten des Index. Da in unserem Fall der gruppierte Index nicht eindeutig ist, also prinzipiell mehrere Zeilen mit demselben Schlüssel existieren können, wird noch eine künstliche *RowId* hinzugefügt, die eine eindeutige Identifizierung einer Zeile ermöglicht. Dies soll durch die Darstellung (ID, 1) für jeden Schlüsselwert in der Abbildung angedeutet werden. Über den Index auf der Spalte `Nr` wird hier also eine Liste von Schlüsselwerten ermittelt. Für jeden Wert aus dieser Liste muss dann anschließend eine Indexsuche auf dem gruppierten Index erfolgen. Diese Suche funktioniert dann genau so, wie in Abschnitt 5.2 bereits erklärt. Für den Wert `Nr=31` werden insgesamt 2.000 Schlüsselwerte für den gruppierten Index gefunden. Da für jeden dieser Schlüsselwerte drei Seiten im gruppierten Index gelesen werden müssen, um die Tabellendaten zu ermitteln (siehe nochmals Abschnitt 5.2), sind insgesamt 6.000 Leseoperationen erforderlich, um das Abfrageergebnis zurückzuliefern.

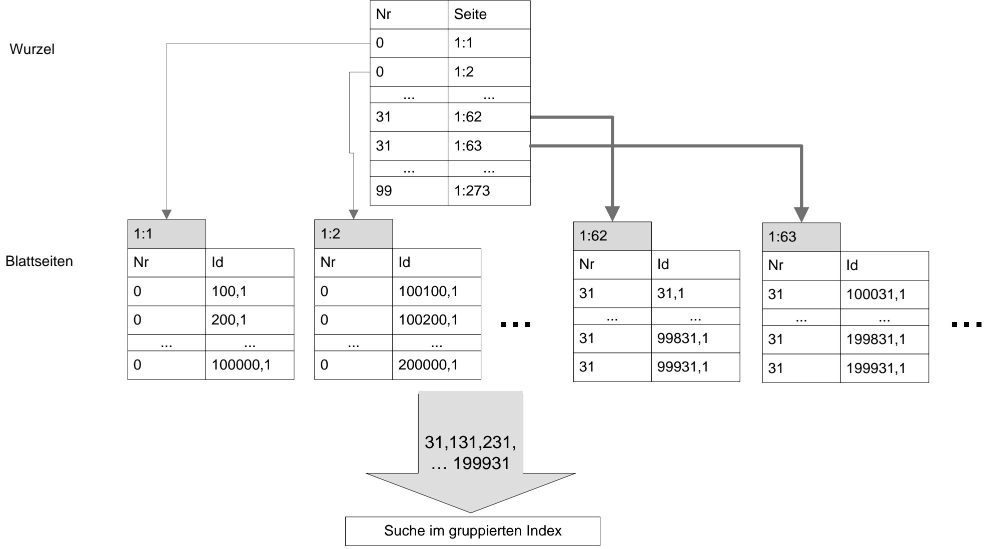


Abbildung 5.5: *Nichtgruppierter Index auf einem gruppierten Index*

Abschließend möchte ich Ihnen noch zwei Dinge mit auf den Weg geben:

Da alle nichtgruppierten Indizes einer Tabelle die Schlüsselwerte des gruppierten Index enthalten, müssen die nichtgruppierten Indizes immer angepasst werden, wenn Schlüsselspalten im gruppierten Index geändert werden. Für gruppierte Indizes sollten Sie daher immer Spalten wählen, die nicht oder nur selten geändert werden.

Wenn Sie eine Tabelle von einem Heap in einen gruppierten Index umwandeln, müssen alle nichtgruppierten Indizes neu erzeugt werden. Dasselbe gilt natürlich für den Fall, dass Sie einen gruppierten Index löschen. Auch in diesem Fall ist es erforderlich, alle nichtgruppierten Indizes neu zu erstellen. Beide Operationen können also viel Zeit beziehungsweise Ressourcen benötigen.

5.5 Eingeschlossene Spalten

Betrachten Sie bitte noch einmal die folgende Abfrage aus dem vorangegangenen Abschnitt:

```
select * from T1
 where Nr = 31
```

Wenn wir wiederum davon ausgehen, dass ein gruppierter Index auf der Spalte `ID` und ein nichtgruppierter Index auf der Spalte `Nr` existieren, dann ergibt sich ein Suchvorgang, wie in Abbildung 5.5 dargestellt. Bei genauerer Betrachtung fällt auf, dass die Suche im gruppierten Index nur durchgeführt wird, um den Wert der Spalte `Platzhalter` zu ermit-

teln. Die anderen beiden Spalten – also `Nr` und `ID` – sind ja bereits in den Blattseiten des nichtgruppierten Index vorhanden. Tatsächlich würde eine Anfrage der Art

```
select Id,Nr from T1
 where Nr = 31
```

keine Suche im gruppierten Index durchführen, da die Werte aller Spalten, die zurückgegeben werden sollen, bereits aus dem nichtgruppierten Index auf der Spalte `Nr` bestimmt werden können. Der Optimierer verzichtet in diesem Fall auf den sogenannten *Lookup* über den gruppierten Index. Dadurch sind dann statt über 6.000 Leseoperationen lediglich noch fünf Leseoperationen erforderlich – eine Verbesserung um den einen Faktor größer als 1000.

Möglicherweise kommen Sie nun auf die recht einfache Idee, die Spalte `Platzhalter` einfach als zusätzliche Spalte in den nichtgruppierten Index aufzunehmen. Damit wäre der Index ein sogenannter abdeckender Index (bezogen auf die obige Abfrage mit dem * als Spaltenliste). Dies würde für unser `SELECT` sicher helfen. Allerdings gilt es zu bedenken, dass jegliche Änderung einer indizierten Spalte immer auch ein Umsortieren des Index bewirkt – und das kann teuer werden.

SQL Server erlaubt seit der Version 2005 die Aufnahme zusätzlicher Spalten in einen nichtgruppierten Index, wobei diese Spalten nicht an der Indizierung selber beteiligt sind, sondern sozusagen aus Huckepack-Information einfach an den Index angefügt werden. Auf diese Weise kann dafür gesorgt werden, dass ein Index abdeckend ist, ohne mit dem Nachteil leben zu müssen, dass eine Änderung einer eingeschlossenen Spalte ein Umsortieren des Indexbaums bewirkt.

Mit eingeschlossenen Spalten beschäftigen wir uns noch einmal in den Kapiteln 9 und 11.

5.6 Gefilterte Indizes

Neu in SQL Server 2008 ist die Möglichkeit, nicht alle Zeilen einer Tabelle in den Index aufzunehmen. Diese Möglichkeit steht natürlich nur für nichtgruppierte Indizes zur Verfügung.

Abhängig von der Filterbedingung ist ein gefilterter Index kleiner als ein vergleichbarer Index über alle Zeilen einer Tabelle. Allerdings müssen Sie hier wirklich sehr genau wissen, was Sie tun. Sie benötigen eine exakte Kenntnis der zu erwartenden Abfragen, sofern Sie gefilterte Indizes verwenden. Allgemein lohnt sich der Einsatz gefilterter Indizes vor allem dann, wenn Sie dadurch die Tiefe des Indexbaums reduzieren können. Auf jeden Fall sind bei einer Suche über den Indexbaum so viele Lesevorgänge erforderlich, wie der Indexbaum Ebenen besitzt. Wenn ein gefilterter Index letztlich genau so viele Ebenen enthält wie ein entsprechender Index ohne Filter, dann lohnt sich der Einsatz sicher kaum. Das Einzige, was Sie damit erreichen, ist ein erhöhtes Risiko der Art, dass der Index nicht verwendet werden kann, falls Ihre Abfragen nicht mit den Filterbedingungen des Index korrespondieren.

Nehmen wir als Beispiel noch einmal unsere Abfrage:

```
select Id,Nr from T1
 where Nr = 31
```

Wenn ein gefilterter Index existiert, der nur die Zeilen enthält, in denen der Wert der Spalte `Nr = 31` ist, kann dann über diesen Index das Abfrageergebnis schneller ermittelt werden als bei einem gleichwertigen nicht gefilterten Index? Beide Indizes – also sowohl der gefilterte als auch der nicht gefilterte – besitzen eine Tiefe von zwei. Die Anzahl der Lesevorgänge konnte hier also durch den gefilterten Index nicht reduziert werden. Lediglich für Aktualisierungen kann die Indexfilterung in diesem Fall nützlich sein, da ja keine Indexaktualisierung für Werte `Nr <> 31` durchgeführt werden muss. Der gefilterte Index bringt das Risiko mit sich, dass Abfragen nicht mehr vom Index profitieren können. So erfordert die folgende Abfrage einen Table Scan:

```
select Id,Nr from T1
 where Nr =32
```

Es gibt sicherlich lohnende Anwendungsfälle für gefilterte Indizes. So ist es zum Beispiel vorstellbar, dass Sie in einer Tabelle mit Rechnungen immer nur die Daten des jeweils aktuellen Jahres abfragen. In diesem Fall kann ein gefilterter Index helfen, der auf der Datum-Spalte der Rechnung erzeugt wird und lediglich die Werte aus dem aktuellen Jahr berücksichtigt. Sie dürfen nur nicht vergessen, diesen Index jeweils am Jahresanfang neu zu generieren (denn Funktionsaufrufe sind in Indexfilterbedingungen nicht erlaubt).

Seien Sie sich bitte dessen bewusst, dass gefilterte Indizes immer auch einen gewissen zusätzlichen Administrationsaufwand erfordern. Wägen Sie deshalb ab, ob sich dieser Aufwand lohnt und, ob ein gefilterter Index also wirklich weniger Aktualisierungen erfordert oder eine geringere Tiefe besitzt als ein entsprechender nicht gefilterter Index.

5.7 Indizierte Sichten

SQL Server erlaubt die Erstellung eines gruppierten Index auch für Sichten. Auf diese Art können Sie eine Sicht materialisieren, also die Zeilen der Sicht permanent speichern. Dies kann für Abfragen vorteilhaft sein, zum Beispiel dann, wenn die Sicht sehr komplex ist, also viele verknüpfte Tabellen enthält. Nützlich ist eine indizierte Sicht meist auch dann, wenn die von der Sicht zurückgelieferten Zeilen Aggregationen oder Gruppierungen enthalten, wodurch die Zeilenanzahl in der Sicht meist erheblich geringer ist als die in den zugrunde liegenden, miteinander verknüpften Tabellen.

Normalerweise wird eine Abfrage auf einer Sicht immer auf die zugrunde liegenden Tabellen zurückgeführt. Bei einer indizierten Sicht kann sich der Optimierer allerdings für die Abfrage des auf einer Sicht existierenden gruppierten Index entscheiden. Darüber hinaus ist es auch möglich, weitere nichtgruppierte Indizes für eine Sicht zu erstellen, die bereits einen gruppierten Index enthält.

Indizierte Sichten weisen jedoch eine Reihe von Einschränkungen auf, die ihre Verwendung erschweren oder sogar unmöglich machen:

Zunächst einmal müssen Sichten, für die ein gruppierter Index erstellt wird, eine Reihe von Voraussetzungen erfüllen. Für eine vollständige Auflistung dieser Voraussetzungen konsultieren Sie bitte die Online-Dokumentation. Hier nur einige Punkte:

- Der gruppierte Index muss eindeutig sein, also mit der Option `UNIQUE` erstellt werden.
- Die Sicht darf keinen OUTER JOIN verwenden.
- Die Sicht muss mit der Option `SCHEMABINDING` erstellt werden.
- Falls Sie in der Sicht Aggregatfunktionen verwenden, muss die Sicht auch eine Spalte `COUNT_BIG(*)` zurückgeben.

Allein die ersten beiden Punkte sind oftmals bereits ein Ausschlusskriterium. Bitte bedenken Sie weiter, dass Modifikationen an Tabellen, die an einer indizierten Sicht beteiligt sind, natürlich immer auch eine Aktualisierung aller für die Sicht existierenden Indizes erfordern. Damit können sich indizierte Sichten (wie alle Indizes) negativ auf die Leistung von Aktualisierungsoperationen auswirken. Glücklicherweise verwendet SQL Server recht intelligente Mechanismen für die Aktualisierung gruppierter Indizes auf Sichten, sodass dieser Punkt meist nicht so schwer wiegt.

Die möglicherweise größte Hürde für den Einsatz von indizierten Sichten ist die Tatsache, dass der Abfrageoptimierer indizierte Sichten nur in der Enterprise Edition von SQL Server automatisch berücksichtigt. In allen anderen Editionen (ausgenommen die Developer Edition, die vom Funktionsumfang mit der Enterprise Edition identisch ist), müssen Sie den Abfragehinweis `NOEXPAND` verwenden, wenn eine Abfrage eine existierende indizierte Sicht verwenden soll. Ansonsten wird die Abfrage in diesen SQL Server-Editionen immer die Tabellen verwenden, auf denen die Sicht basiert. Wie so oft ist auch in diesem Fall die Verwendung des Abfragehinweises potenziell gefährlich. Je nach Struktur der Abfrage kann es nämlich durchaus teurer sein, die Sicht über den gruppierten Index abzufragen, anstatt die Basistabellen zu verwenden. Die Entscheidung, welcher Weg der bessere ist, sollten Sie auf jedem Fall dem Optimierer überlassen. – Diese Möglichkeit haben Sie aber eben nur in der relativ teuren Enterprise Edition von SQL Server.

Nachdem nun die theoretischen Grundlagen erläutert wurden, beschäftigt sich der Rest dieses Kapitels mit der Verwaltung von Indizes.

5.8 Erstellen von Indizes

SQL Server kennt prinzipiell zwei Möglichkeiten zur Erstellung eines Index. Zum einen können Indizes manuell erzeugt werden. Hierzu können Sie das Management Studio oder entsprechende DDL-Befehle verwenden. In einigen Fällen werden Indizes auch automatisch erstellt, zum Beispiel wenn Sie bestimmte Einschränkungen für Tabellen oder Spalten angeben. Etwas weiter unten werden wir hierzu einige Beispiele betrachten.

5.8.1 Manuelles Erstellen von Indizes: CREATE INDEX

Für die manuelle Erzeugung eines Index verwenden Sie das Kommando `CREATE INDEX`. Dieses Kommando hat die folgende (vereinfachte) Syntax:

```
create [unique] [clustered | nonclustered] index <index_name>
        on <tabellen-name> (<spalten-liste>)
  [include (<spalten-liste>)]
    [where <filter-kriterium>]
```

Für den Index muss also zunächst einmal ein Name angegeben werden. Außerdem ist natürlich der Name der Tabelle erforderlich. Nach dem Tabellennamen folgen dann die indizierten Spalten in einer kommagetrennten Liste. Hier ist die Reihenfolge der Spalten wesentlich. In der Regel wird der Abfrageoptimierer nur die erste Spalte für eine Suche im Indexbaum in Betracht ziehen.

Über die `INCLUDE`-Klausel können Sie zusätzliche Spalten in den Index einfügen, die im Index enthalten sind, aber nicht die Sortierung des Index beeinflussen. Dadurch haben Sie die Möglichkeit, abdeckende Indizes zu erstellen.

Schließlich können Sie für einen nichtgruppierten Index über die `WHERE`-Klausel noch eine Filterbedingung angeben, um zu erreichen, dass nicht alle Zeilen der Tabelle in den Index aufgenommen werden.

Wie aus der Syntax ersichtlich, können Sie zwischen den Schlüsselwörtern `CREATE` und `INDEX` über die folgenden Optionen die Art des erzeugten Index spezifizieren:

- **CLUSTERED**. Die Anweisung erstellt einen gruppierten Index
- **NONCLUSTERED**. Die Anweisung erstellt einen nichtgruppierten Index. Dies ist gleichzeitig auch die Standardoption. Wenn Sie also weder CLUSTERED noch NONCLUSTERED angeben, wird ein nichtgruppierter Index erstellt.
- **UNIQUE**. Der erstellte Index ist eindeutig. Dies bedeutet, dass in der Tabelle nicht mehrere Zeilen mit demselben Wert für die Indexspalte(n) existieren dürfen. Es ist von Vorteil, wenn Sie diese Option angeben, sofern Sie Indizes verwenden, die eindeutig sind, da dadurch die Verwaltung des Index erleichtert wird.

Wir wollen nun für unsere Beispieltabelle `T1` einen gruppierten Index auf der Spalte `ID` und einen nichtgruppierten Index auf der Spalte `Nr` erzeugen, so wie wir dies im ersten Teil dieses Kapitels angenommen haben. Betrachten Sie hierzu bitte noch einmal Abbildung 5.5.

Den gruppierten Index erzeugen wir also für die Spalte `ID`:

```
create unique clustered index Ix_T1_Id on T1(Id)
```

Da die `ID` eindeutig ist, legen wir den Index mit der Option `UNIQUE` an. Der zweite Index ist der nichtgruppierte Index auf der Spalte `Nr`:

```
create nonclustered index Ix_T1_Nr on T1(Nr)
```

Für beide Indizes können Sie die physikalischen Parameter über die dynamische Verwaltungssicht abfragen:

```
select index_id,index_type_desc,index_depth
      ,index_level,page_count,record_count
  from sys.dm_db_index_physical_stats(db_id(),object_id('T1')
                                     ,null,null,'detailed')
```

Die Abfrage liefert Ergebnisse für jede Stufe im Index zurück. Abbildung 5.6 zeigt das Ergebnis auf meinem PC.

index_id	index_type_desc	index_depth	index_level	page_count	record_count
1	CLUSTERED INDEX	3	0	22223	200000
1	CLUSTERED INDEX	3	1	36	22223
1	CLUSTERED INDEX	3	2	1	36
2	NONCLUSTERED INDEX	2	0	273	200000
2	NONCLUSTERED INDEX	2	1	1	273

Abbildung 5.6: *Physikalische Struktur der erzeugten Indizes*

Der gruppierte Index hat eine Tiefe von drei, der nichtgruppierte eine Tiefe von zwei. Außerdem zeigt das Ergebnis, wie viele Seiten bzw. Zeilen jede Stufe des Index enthält.

Sie können auch erkennen, dass die Anzahl der Blattseiten im gruppierten Index (dies sind die Seiten auf der Stufe 0) sehr groß ist. Das ist natürlich zu erwarten, da in diesen Seiten die eigentlichen Tabellendaten gespeichert werden (siehe Abbildung 5.2).

5.8.2 Automatische Erstellung von Indizes

Beim Verwenden folgender Einschränkungen erstellt SQL Server automatisch Indizes:

- **PRIMARY KEY**. Für den Primärschlüssel einer Tabelle wird ein eindeutiger gruppierter Index erstellt.
- **UNIQUE**. Für diese Einschränkung wird automatisch ein eindeutiger, nichtgruppierter Index erstellt.

Beide Optionen sind absolut sinnvoll, da sie die Prüfung der entsprechenden Einschränkung bei Datenänderungen radikal beschleunigen.

5.8.3 Indizes auf Sichten

Betrachten Sie bitte die folgende Abfrage, die für jeden vorhandenen Wert in der Spalte `Nr` die Anzahl der Zeilen zählt:

```
select Nr,count(*) as Anzahl
  from T1
  group by Nr
```

Die obige Abfrage muss alle 200.000 Zeilen der Tabelle `T1` durchlaufen, um das Ergebnis zu ermitteln. Dieses Ergebnis enthält letztlich nur 100 Zeilen.

Wir können eine indizierte Sicht erstellen, um die Abfrage zu beschleunigen. Wie bereits in Abschnitt 5.7 gesagt, muss eine solche Sicht mit der Option SCHEMABINDING erzeugt werden und – da wir Aggregatfunktionen verwenden – auch eine COUNT_BIG(*)-Spalte enthalten. Wir können diese Sicht also zum Beispiel so erzeugen:

```
create view V1 with schemabinding as
   select Nr,count_big(*) as anz
     from dbo.T1
    group by Nr
go
create unique clustered index Ix_V1_Nr on V1 (Nr)
```

Bitte beachten Sie, dass der Tabellenname hier die Angabe des Schemanamens (also dbo.T1) enthalten muss, da die Sicht mit der Option SCHEMABINDING erzeugt wird.

Wir können nun die folgende Abfrage ausführen, um die Anzahl der Zeilen je Nr zu erhalten:

```
select * from V1
```

Jetzt werden statt vorher 200.000 Zeilen nur noch 100 Zeilen gelesen, eine Verbesserung um den Faktor 2.000. Dies funktioniert jedoch nur in der Enterprise Edition von SQL Server so. In allen anderen Editionen (mit Ausnahme der Developer Edition) wird die indizierte Sicht nicht berücksichtigt. Die Abfrage verwendet in diesen Editionen die der Sicht zugrunde liegenden Tabellen. Hier müssen Sie explizit angeben, dass Sie auf die indizierte Sicht zugreifen wollen:

```
select * from V1 with (noexpand)
```

Wie bereits gesagt, ist dies gefährlich, da es in bestimmten Fällen durchaus effizienter sein kann, die Basistabellen abzufragen und eben nicht die indizierte Sicht direkt zu verwenden.

Interessant ist übrigens auch, dass der Optimierer die indizierte Sicht verwenden kann, auch wenn diese in der Abfrage überhaupt nicht auftaucht. Betrachten Sie hierzu bitte die folgende Abfrage:

```
select Nr,count(*)
  from T1
  group by Nr
```

Der Optimierer erkennt, dass die indizierte Sicht V1 der beste Weg ist, das Abfrageergebnis zu ermitteln und verwendet diese Sicht anstelle der Tabelle T1. Auch dieses Feature steht Ihnen nur in der Enterprise Edition zur Verfügung.

5.8.4 Index-Füllfaktor

Bei der Erstellung eines Index kann für diesen ein sogenannter Füllfaktor angegeben werden. Dieser Füllfaktor bestimmt, zu wie viel Prozent die Seiten in der Blattebene des Index mit Daten gefüllt werden. Es kann sinnvoll sein, die Blattseiten nicht zu 100 Prozent zu füllen, zum Beispiel dann, wenn Sie erwarten, dass die Indexspalten häufig geän-

dert werden. Bei Änderungen an Indexspalten ist ja jeweils ein Umorganisieren des Index erforderlich, was dazu führen kann, dass der Indexbaum neu strukturiert werden muss, beispielsweise weil Blattseiten hinzugekommen sind. In vielen Fällen kann es daher sinnvoll sein, etwas Speicher in den Blattseiten freizuhalten, um diese Umstrukturierungen zu minimieren. Hierzu dient der Füllfaktor, über den angegeben wird, wie viel Prozent in einer Blattseite bei der Erstellung des Index maximal belegt werden dürfen. Ein niedriger Füllfaktor beschleunigt also unter Umständen das Ändern von Indexspalten. Dafür benötigt ein solcher Index dann allerdings mehr Speicherplatz. Dies kann sogar so weit führen, dass ein Index mit einem niedrigen Füllfaktor letztlich mehr Ebenen benötigt, als ein gleicher Index mit einem höheren Füllfaktor. Dadurch wird die Suche über den Indexbaum, die ja immer die gesamte Tiefe des Baums durchforstet, natürlich dementsprechend langsamer.

Für die Angabe des Füllfaktors verwenden Sie die `WITH`-Klausel der `CREATE INDEX`-Anweisung. Die folgende Anweisung füllt die Blattseiten des Index zu 90 Prozent:

```
create index Ix_T1_Id2 on T1(Id)
   with (fillfactor=90)
```

Die `CREATE INDEX`-Anweisung kennt auch eine `PAD_INDEX`-Option, über die zusätzlich angegeben werden kann, dass der Füllfaktor nicht nur für die Blattseiten, sondern ebenso für Nicht-Blattseiten gelten soll:

```
create index Ix_T1_Id2 on T1(Id)
 with (fillfactor=90, pad_index=on)
```

`PAD_INDEX` darf nur zusammen mit `FILLFACTOR` verwendet werden.

Bitte bedenken Sie, dass der angegebene Füllfaktor nur beim Erzeugen des Index verwendet wird. Eine spätere dynamische Verwaltung erfolgt nicht.

Falls Sie auf die Angabe des Füllfaktors verzichten, so gilt für den Füllfaktor eine serverweite Standardeinstellung. Nach der Installation von SQL Server steht dieser Wert auf 0, was gleichbedeutend mit 100 Prozent ist. Die Einstellung des Standard-Füllfaktors können Sie über die gespeicherte Prozedur `sp_configure` abfragen bzw. festlegen. Da es sich hierbei um eine erweiterte Option handelt, ist es erforderlich, zunächst die Anzeige der erweiterten Optionen einzuschalten. Das folgende Skript fragt den Füllfaktor ab und ändert ihn anschließend auf 90 Prozent:

```
-- Einschalten der Anzeige der erweiterten Optionen
exec sp_configure 'show advanced options',1
reconfigure
go

-- Anzeige des aktuellen Füllfaktors
exec sp_configure 'fill factor %'

-- Ändere den Standard-Füllfaktor auf 90%
exec sp_configure 'fill factor %',90
reconfigure

-- Noch einmal: Anzeige des aktuellen Füllfaktors
exec sp_configure 'fill factor %'
```

5.8.5 Einen Index neu aufbauen

Das Kommando `CREATE INDEX` kennt auch eine Option `DROP_EXISTING=ON`. Über diese Option kann ein Index in einem Zuge gelöscht und neu erzeugt werden. Dies kann zum Beispiel für die Änderung des Füllfaktors eines Index nützlich sein. Da der Füllfaktor nur bei `CREATE INDEX` angegeben werden kann und eine spätere Änderung nicht möglich ist, muss der Index neu erzeugt werden. Um den gruppierten Index `Ix_T1_Id` mit einem Füllfaktor von 20 Prozent neu zu erstellen, können wir das folgende Kommando verwenden:

```
create clustered index Ix_T1_Id on T1(Id)
    with (fillfactor=20, pad_index=on,drop_existing=on)
```

Wir können nun nochmals prüfen, wie viele Stufen der gruppierte Index enthält. Verwenden Sie hierzu die dynamische Systemsicht `sys.dm_db_index_physical_stats`, wie weiter vorne in diesem Kapitel angegeben. Das Ergebnis zeigt Abbildung 5.7.

index_id	index_type_desc	index_depth	index_level	page_count	record_count
1	CLUSTERED INDEX	4	0	100000	200000
1	CLUSTERED INDEX	4	1	800	100000
1	CLUSTERED INDEX	4	2	8	800
1	CLUSTERED INDEX	4	3	1	8

Abbildung 5.7: *Der Index mit einem Füllfaktor von 20 Prozent hat nun vier Stufen.*

Tatsächlich besitzt der gruppierte Index also nun vier Stufen. Jede Suche über den Indexbaum benötigt jetzt einen Lesevorgang mehr. Dies mag Ihnen zunächst nicht allzu dramatisch erscheinen. Wenn Sie aber nochmals Abbildung 5.5 betrachten, dann wird Folgendes klar:

Die Suche über den nichtgruppierten Index hat 2.000 Schlüsselwerte gefunden. Für jeden dieser Schlüssel muss nun der gruppierte Index durchsucht werden. Bei einer Tiefe des Indexbaums von drei sind dies 6.000 Leseoperationen. Besitzt der Indexbaum vier Ebenen, so wie nach der Änderung des Füllfaktors auf 20 Prozent, so sind insgesamt 8.000 Leseoperationen erforderlich, also immerhin ein Drittel mehr.

5.8.6 Löschen von Indizes

Existierende Indizes werden durch die Anweisung `DROP INDEX` gelöscht. Hierbei muss neben dem Namen des zu löschenden Index auch der Name der Tabelle angegeben werden. Die folgende Anweisung löscht den gruppierten Index auf der Tabelle `T1`:

```
drop index Ix_T1_Id on T1
```

Noch einmal zur Erinnerung: Das Löschen eines gruppierten Index für eine Tabelle bewirkt die Umwandlung dieser Tabelle in einen Heap. Dadurch müssen alle nichtgruppierten Indizes neu erzeugt werden.

5.9 Zusammenfassung

In diesem einführenden Kapitel zu Indizes haben Sie zunächst erfahren, wie Indizes funktionieren und wie sie verwaltet werden. Im weiteren Verlauf des Buches werden diese Grundlagen noch häufig benötigt und darauf aufbauend weitere Konzepte entwickelt. Daher sollten Sie sicher sein, dass Sie die Erklärungen aus diesem Kapitel verstanden haben, bevor Sie fortfahren zu lesen. Insbesondere sollten Sie sich die folgenden Punkte einprägen:

Ein Index wird als B*-Baum-Struktur erstellt und stets über bestimmte Spalten einer Tabelle gebildet. Wenn sich der Optimierer für die Verwendung des Index entscheidet, wird eine Suche oder Sortierung nach den Indexspalten um Größenordnungen beschleunigt, da die Suche nun nicht mehr sequenziell, sondern binär erfolgt.

Ein Index ist letztlich nur ein physikalisches Hilfskonstrukt. Die im Index enthaltene Information ist redundant und für Anwendungen nicht relevant. Ein Index behindert oftmals Aktualisierungsoperationen, da neben den eigentlichen Tabellendaten auch die Daten im Indexbaum aktualisiert werden müssen. Insbesondere Einfüge- und Löschoperationen (`INSERT` bzw. `DELETE`) führen dazu, dass immer alle bestehenden Indizes einer Tabelle aktualisiert werden müssen.

Eine Antwort auf einige wesentliche Fragen wurde in diesem Kapitel zunächst nicht gegeben. Hierzu gehören zum Beispiel die folgenden Fragen:

- Was ist ein geeigneter Kandidat für den gruppierten Index?
- Wie findet man überhaupt geeignete Indizes?
- Welche Kriterien verwendet der Optimierer für die Auswahl von Indizes?
- Wann bzw. warum wird ein Index vom Optimierer schlichtweg ignoriert?
- Existieren geeignete Indizes in meiner Datenbank? Gibt es vielleicht überflüssige oder fehlende Indizes?

In den Kapiteln 6, 9 und 11 erhalten Sie Antworten auf diese Fragen.

6 Verwalten von Indizes

Im vorangegangenen Kapitel haben Sie gesehen, wie die Abfrageleistung durch geeignete Indizes ziemlich drastisch verbessert werden kann. Diese Verbesserung bekommen Sie leider nicht völlig umsonst. Indizes sind letztlich Datenbankobjekte, die auch einen Administrationsaufwand erfordern. Dieser administrative Aufwand kann durchaus ins Gewicht fallen, wenn Ihre Daten und Anwendungen sich häufig verändern.

Nun werden Sie sich vielleicht fragen, welche Art der »Index-Administration« durch Datenänderungen hervorgerufen wird. Schließlich sind Datenbanken ja dafür konzipiert, dass Daten geändert werden. Warum also muss man bei solchen Änderungen die Indexverwaltung überwachen bzw. anpassen?

Die Antwort auf diese Frage ist recht einfach. Ganz allgemein ist es so, dass durch Datenänderungen immer auch ein administrativer Aufwand entsteht. So kann es ja zum Beispiel sein, dass das Datenvolumen derart angewachsen ist, dass Sie die Hardware aufrüsten oder etwa Ihr Datensicherungskonzept anpassen müssen.

Auch Ihre Indizes müssen natürlich hinsichtlich der Art und Weise des Datenzugriffs optimiert und überwacht werden. Die diesbezügliche Administration beschränkt sich im Wesentlichen auf zwei Punkte:

- Zunächst einmal verändern sich bei Datenänderungen natürlich auch die Indizes von Tabellen bzw. indizierten Sichten. Dadurch kann ein Index fragmentiert werden, was sich negativ auf die Abfrageleistung auswirkt. Hier hilft eine Reorganisation oder auch die erneute Erstellung eines solchen Index.
- Angenommen, Sie haben einen optimal passenden Satz von Indizes für Ihre Datenbank erzeugt. (Wie Sie vorgehen können, um die passenden Indizes zu finden, erfahren Sie weiter unten in diesem Kapitel und in Kapitel 11). Änderungen an Tabellendaten, aber auch Modifikationen an den Anwendungen, die Ihre Datenbank verwenden, bewirken unter Umständen, dass dieser Satz von Indizes jetzt auf einmal nicht mehr optimal ist. Hierbei kann es zum einen vorkommen, dass ehemals erforderliche Indizes nun nicht mehr benötigt werden. Auch der umgekehrte, meist sehr viel unerfreulichere Fall kann auftreten, dass nämlich auf einmal Indizes fehlen und Ihre Abfrageleistung damit ganz empfindlich nach unten geht. Um solche Situationen zu erkennen, müssen Sie das laufende System – also Ihren SQL Server – überwachen.

6.1 Fragmentierung und Reorganisation

Wie bereits in der Einleitung zu diesem Kapitel erwähnt, bewirken Datenänderungen an Tabellen stets auch Änderungen an den entsprechenden Indizes. Ein Index enthält letztlich redundante Information, die zu jedem Zeitpunkt mit den Daten in der zugehörigen Tabelle synchron sein muss. Mit der Zeit kann dies dazu führen, dass ein Index fragmentiert wird, das heißt die Blöcke, welche die Seiten des Index enthalten, sind nicht zusammenhängend auf der Festplatte gespeichert. Ein fragmentierter Index kann das Navigieren im Indexbaum ganz erheblich beeinträchtigen. Dies einfach deshalb, weil durch diese Fragmentierung zusätzliche Bewegungen des Schreib-/Lesekopfes der Festplatte erforderlich sind. Solche mechanischen Operationen sind nach wie vor der eigentliche Flaschenhals bei Ein- und Ausgabeoperationen.

Glücklicherweise bietet SQL Server komfortable Möglichkeiten sowohl für die Erkennung von Fragmentierungen als auch für die Beseitigung derselben an. Welche Möglichkeiten dies sind, soll wieder ein kleines Beispiel zeigen.

Wir erzeugen zunächst eine Tabelle mit einem gruppierten und einem nichtgruppierten Index:

```
use QueryTest;
if (object_id('T1') is not null)
  drop table t1;
create table T1
 (
   Id int not null
  ,Nr int not null
  ,Platzhalter nchar(400) null default '#'
 )
go
-- Lege zwei Indizes an
create clustered index Ix_T1_Id on T1(Id)
create nonclustered index Ix_T1_Nr on T1(Nr)
go
```

Diese Tabelle ist identisch mit der aus dem vorangegangenen Kapitel.

Nachdem die Tabelle und die Indizes erzeugt wurden, fügen wir nun zunächst Zeilen ein, löschen anschließend einige Zeilen und fügen dann weitere Zeilen ein. Diese Aufgabe erledigt das folgende Skript:

```
-- Füge 100.000 Zeilen ein
insert T1 (Id,Nr)
  select checksum(newid()), checksum(newid()) % 1000
    from Numbers
   where n <= 100000
go
-- Lösche einige Zeilen
delete T1
 where Nr between 0 and 500
```

```
go
-- Füge weitere 100.000 Zeilen ein
insert T1 (Id,Nr)
  select checksum(newid()), checksum(newid()) % 1000
    from Numbers
   where n <= 100000
```

Über den Objekt-Explorer im Management Studio können Sie sich nun die Indexeigenschaften für den existierenden Index `Ix_T1_Nr` anzeigen lassen (aus dem Kontextmenü für DATENBANKEN • QUERYTEST • TABELLEN • DBO.T1 • INDIZES • IX_T1_NR). Wählen Sie im Fenster INDEXEIGENSCHAFTEN die Seite FRAGMENTIERUNG. Auf meinem PC ergibt sich ein Bild, wie in Abbildung 6.1 gezeigt.

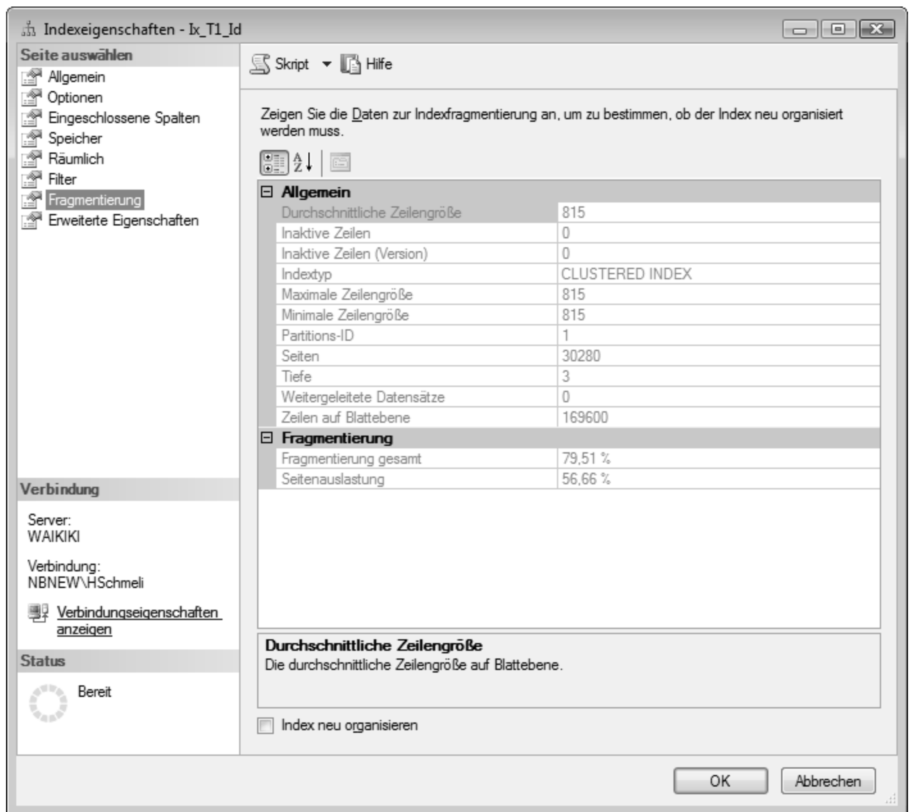


Abbildung 6.1: *Fragmentierung des Index »Ix_T1_Nr« nach Datenänderungen*

Der Index ist also zu beinahe 80 Prozent fragmentiert und sollte daher unbedingt reorganisiert werden. Sie können das Reorganisieren sofort erledigen, indem Sie die Option INDEX NEU ORGANISIEREN auswählen und das Fenster über OK schließen.

Wir werden den gruppierten Index für die Tabelle etwas weiter unten neu aufbauen. Zuvor sollen aber noch einige Untersuchungen mit dem fragmentierten Index durchgeführt werden.

Die in Abbildung 6.1 gezeigte Fragmentierung betrifft die Blattseiten des Index. Eine Information über die Fragmentierung der Nicht-Blattseiten wird nicht angezeigt. Falls Sie diese Information benötigen, können Sie hierfür die dynamische Systemsicht `sys.dm_db_index_physical_stats` verwenden:

```
select index_id,index_type_desc,index_depth
      ,index_level,page_count,record_count
      ,cast(avg_fragmentation_in_percent as decimal(6,2))
             as avg_fragmentation_in_percent
  from sys.dm_db_index_physical_stats(db_id(),object_id('T1')
                                     ,null,null,'detailed')
 where index_id=1  -- Gruppierter Index
```

Die Angabe des Wertes `detailed` für den letzten Parameter bewirkt, dass Informationen über alle Stufen des Indexbaums zurückgeliefert werden. Außerdem fragen wir nur die Werte für den gruppierten Index ab, dessen `index_id` stets den Wert 1 hat. Das Ergebnis der Abfrage zeigt nun auch die Werte für die Nicht-Blattseiten an (Abbildung 6.2).

	index_id	index_type_desc	index_depth	index_level	page_count	record_count	avg_fragmentation_in_percent
1	1	CLUSTERED INDEX	3	0	30280	174800	79.51
2	1	CLUSTERED INDEX	3	1	83	30280	98.80
3	1	CLUSTERED INDEX	3	2	1	83	0.00

Abbildung 6.2: *Fragmentierung des gruppierten Index der Tabelle »T1«*

Seien Sie bitte vorsichtig mit einer Abfrage der Sicht `sys.dm_db_index_physical_stats`, insbesondere bei Verwendung von `detailed` als Wert für den letzten Parameter. Bei einer solchen Abfrage wird das Festplattensystem sehr stark belastet.

Wir beschäftigen uns für die weiteren Betrachtungen mit dem gruppierten Index und werden sehen, wie die Fragmentierung des gruppierten Index die Ausführung von Abfragen in negativer Weise beeinflussen kann. Zunächst ist zu erkennen, dass in den Blattseiten 174.800 Zeilen gespeichert werden. Jede dieser Zeilen benötigt 808 Byte. Somit werden insgesamt für die Speicherung aller Zeilen der Tabelle 808 × 174.800 / 8060 = 17.524 Blattseiten benötigt, wenn die Seiten zu 100 Prozent gefüllt sind. Tatsächlich werden aber 30.280 Seiten verwendet, wie die Spalte `page_count` zeigt. Dies bedeutet, dass in den einzelnen Datenseiten sehr viel freier Platz existiert, die Seiten also nicht vollständig gefüllt sind. Da SQL Server stets komplette Datenseiten liest, werden demnach unnötige E/A-Operationen erzeugt, wenn Daten aus der Tabelle abgerufen werden. Wir wollen dies an einem Extrembeispiel zeigen.

Die folgende Abfrage benötigt alle Zeilen der Tabelle `T1`:

```
select checksum_agg(checksum(*)) from T1
```

Wenn wir für die Ausführung die Option `STATISTICS IO` einschalten, so erhalten wir die erwartete Anzahl logischer Lesevorgänge, nämlich 30.365 (30.280 Blattseiten + 84 Nicht-Blattseiten + 1 Lesevorgang zum Auffinden der Wurzel des Indexbaums):

```
'T1'-Tabelle. Scananzahl 1, logische Lesevorgänge 30365, physische Lesevorgänge 0,
Read Ahead-Lesevorgänge 0, logische LOB-Lesevorgänge 0, physische LOB-Lesevorgänge 0,
Read Ahead-LOB-Lesevorgänge 0.
```

Das sind immerhin über 40 Prozent mehr Lesevorgänge als im Idealfall – nämlich mit 100 Prozent gefüllten Blattseiten – benötigt würden.

Noch gravierender wirkt sich der Unterschied dann aus, wenn die benötigten Daten noch nicht im Datencache sind und von der Festplatte geholt werden müssen. Die dann erforderlichen physikalischen E/A-Operationen sind normalerweise noch einmal um ein Vielfaches langsamer als das Lesen von Daten aus dem Datencache.

Unser gruppierter Index sollte also defragmentiert werden. In den folgenden drei Abschnitten wird erklärt, wie Sie dabei vorgehen.

6.1.1 Einen Index reorganisieren

Mit dem Reorganisieren eines Index ist eine Neuanordnung der Blattseiten gemeint. Auf allen anderen Ebenen findet keine Neuanordnung der Indexseiten statt, mit der einen Ausnahme, dass in der ersten Nicht-Blattebene die Zeiger auf die Blattseiten aktualisiert werden.

Die vereinfachte Syntax für die Reorganisation eines Index sieht so aus:

```
alter index <indexname> on <tabelle> reorganize
```

Es ist also erforderlich, sowohl den Namen des Index als auch den Namen der Tabelle, für die der Index existiert, anzugeben. Die Index-Reorganisation ist eine Online-Operation. Dies bedeutet, dass der Index in Abfragen weiterhin verwendet werden kann – auch während die Reorganisation läuft.

Es wird empfohlen, einen Index zu reorganisieren, wenn der Grad der Fragmentierung 10 Prozent übersteigt. Das Reorganisieren des gruppierten Index bewirkt übrigens nicht, dass in der Folge auch alle nichtgruppierten Indizes neu organisiert werden müssen. Die Schlüsselwerte des gruppierten Index verändern sich durch die Reorganisation ja nicht. Die in den nichtgruppierten Indizes gespeicherten Schlüsselwerte des gruppierten Index sind daher auch nach der Reorganisation des gruppierten Index nach wie vor korrekt.

Um unseren gruppierten Index auf der Tabelle `T1` zu reorganisieren, können wir das folgende Kommando ausführen:

```
alter index Ix_T1_Id on T1 reorganize
```

Wenn wir nun noch einmal die physikalischen Parameter des gruppierten Index abfragen, dann sieht das Resultat bereits erheblich besser aus (Abbildung 6.3).

	index_id	index_type_desc	index_depth	index_level	page_count	record_count	avg_fragmentation_in_percent
1	1	CLUSTERED INDEX	3	0	21774	174800	0.39
2	1	CLUSTERED INDEX	3	1	83	21774	98.80
3	1	CLUSTERED INDEX	3	2	1	83	0.00

Abbildung 6.3: *Gruppierter Index nach einer Reorganisation*

Eine Abfrage über alle Zeilen der Tabelle muss jetzt nur noch ca. 22.000 Datenseiten lesen. Das sind immerhin rund 30 Prozent weniger als mit dem stark fragmentierten Index.

Hier kommt noch ein weiterer Aspekt hinzu. Insgesamt werden nun auch weniger Seiten in den Datencache übertragen, und dies verringert den vom SQL Server-Prozess benötigten Hauptspeicher – ein nicht zu unterschätzender Punkt.

6.1.2 Einen Index neu erstellen

Allgemein wird empfohlen, einen Index komplett neu zu erstellen, sobald der Fragmentierungsgrad des Index 40 Prozent übersteigt.

Die Neuerstellung eines Index kann auf zwei Arten erfolgen:

1. Auch für diesen Fall existiert ein `ALTER INDEX`-Kommando, dessen vereinfachte Syntax so aussieht:

   ```
   alter index <indexname> on <tabelle> rebuild
         [with (online=on | off)]
   ```

   Die `WITH`-Klausel ist hier optional und ermöglicht unter anderem die Online-Erstellung des Index. Der Standard ist `OFFLINE`, was zur Folge hat, dass der Index während der Neuerstellung nicht in Abfragen verwendet werden kann. Über die Angabe von `WITH (ONLINE=ON)` können Sie dieses Verhalten ändern. Allerdings benötigt die Online-Erstellung eines Index auch erheblich mehr Systemressourcen, da zunächst eine Kopie des Index erstellt wird, die nach ihrer Fertigstellung den ursprünglichen Index ersetzt.

   Um unseren gruppierten Index online neu zu erstellen, können wir also das folgende Kommando verwenden:

   ```
   alter index Ix_T1_Id on T1 rebuild
         with (online=on)
   ```

   Auch hier gilt, dass ein Neuerstellen eines gruppierten Index nicht die Schlüsselwerte des Index ändert und daher keine automatische Neuerstellung aller existierenden nichtgruppierten Indizes zur Folge hat.

2. Natürlich können Sie einen Index auch löschen und komplett neu erzeugen. Wie Sie dies bewerkstelligen, haben Sie bereits in Kapitel 5 erfahren.

Zum Schluss möchte ich Ihnen unbedingt noch einige Hinweise bezüglich gruppierter Indizes mit auf den Weg geben.

Wenn Sie einen gruppierten Index löschen, so werden stets alle existierenden nichtgruppierten Indizes neu erstellt. Dies gilt auch, wenn Sie einen gruppierten Index für einen Heap erzeugen.

Einen gruppierten Index über `DROP INDEX` zu löschen und anschließend durch `CREATE INDEX` neu zu erstellen ist daher die denkbar schlechteste Möglichkeit für die Reorganisation eines gruppierten Index, denn dies hat zur Folge, dass alle existierenden nichtgruppierten Indizes zweimal neu erzeugt werden. Sie können die Option `DROP_EXISTING=ON` verwenden, falls Sie einen gruppierten Index löschen und neu erzeugen möchten. Solange Sie dabei die existierenden Spalten und ihre Reihenfolge nicht verändern, werden die nichtgruppierten Indizes nicht neu erstellt.

6.1.3 Strategie zur Indexprüfung und Indexdefragmentierung

Sie sollten Ihre Indexstruktur regelmäßig überprüfen und zu stark fragmentierte Indizes neu organisieren oder komplett neu erstellen. Prinzipiell haben Sie in den vorangegangenen Abschnitten bereits gesehen, wie Sie den diesbezüglichen Status Ihrer Indizes durch die dynamische Systemsicht `sys.dm_db_index_physical_stats` abfragen können. Diese Sicht enthält Informationen über die Fragmentierung, die entsprechende Tabelle sowie die ID des zugehörigen Index. Was wir dort nicht finden, ist der Name des Index. Diesen Namen können wir über eine Verknüpfung zur Systemsicht `sys.indexes` aber leicht ermitteln. Eine entsprechende Abfrage kann dann so aussehen:

```
select object_name(ps.object_id) as TabellenName
      ,i.Name                    as IndexName
      ,cast(ps.avg_fragmentation_in_percent as decimal(6,2))
                                 as Fragmentierung
      ,case
         when ps.avg_fragmentation_in_percent < 10 then 'nichts'
         when ps.avg_fragmentation_in_percent >= 10
          and ps.avg_fragmentation_in_percent < 40 then 'reorganize'
         else 'rebuild'
       end                       as Erforderlich
  from sys.dm_db_index_physical_stats(db_id(),null,null,null,'limited') as ps
       inner join sys.indexes as i
               on i.index_id=ps.index_id
              and i.object_id=ps.object_id
 where objectproperty(i.object_id, 'IsUserTable') = 1
```

Abbildung 6.4 zeigt ein Beispiel für das Resultat der obigen Abfrage.

	TabellenName	IndexName	Fragmentierung	Erforderlich
1	T1	Ix_T1_Id	0.01	nichts
2	T1	Ix_T1_Nr	83.63	rebuild

Abbildung 6.4: *Existierende Indizes, die eine Neuerstellung benötigen*

Mit dieser Abfrage ist es nur noch ein kleiner Schritt zur Erzeugung von SQL-Kommandos, die eine Reorganisation oder eine Neuerstellung der vorhandenen Indizes durchführen. Hierzu erweitern wir die obige Abfrage etwas:

```
with IndexReorg(TabellenName, IndexName, Erforderlich) as
 (
  select object_name(ps.object_id)
        ,i.Name
        ,case
           when ps.avg_fragmentation_in_percent < 10 then 'nichts'
           when ps.avg_fragmentation_in_percent >= 10
            and ps.avg_fragmentation_in_percent < 40 then 'reorganize'
           else 'rebuild'
         end
    from sys.dm_db_index_physical_stats(db_id(),null
                                   ,null,null,'limited') as ps
         inner join sys.indexes as i
                 on i.index_id=ps.index_id
                and i.object_id=ps.object_id
   where objectproperty(i.object_id, 'IsUserTable') = 1
  )
  select 'alter index ' + quotename(IndexName)
         + ' on ' + quotename(TabellenName)
         + ' ' + Erforderlich as Kommando
    from IndexReorg
   where Erforderlich != 'nichts'
   order by TabellenName, IndexName
```

Die von dieser Abfrage zurückgelieferten SQL-Anweisungen können Sie dann einfach für die Reorganisation Ihrer Indizes verwenden. Natürlich wäre es auch möglich, das obige Skript durch einen Cursor so zu erweitern, dass dieser die Reorganisation gleich mit erledigt. Das funktioniert übrigens auch ohne Cursor. Nehmen wir an, dass eine Sicht `vIndexReorg` existiert, die die obige Abfrage enthält. Diese Sicht könnten Sie dann auch so verwenden:

```
declare @cmd nvarchar(max)
set @cmd = ';'
-- Sammle alle Index-Reorg-Anweisungen in der deklarierten Variablen
select @cmd = @cmd + Kommando
  from vIndexReorg
 order by TabellenName, IndexName
-- Zur Information die Anweisungen ausgeben
print @cmd
-- Jetzt die Reorganisation starten
exec (@cmd)
```

Ein derartiges Skript sollten Sie in periodischen Abständen – und natürlich zu Zeiten geringer sonstiger Serveraktivität – ausführen, damit Ihre Indizes nicht zu sehr fragmentiert werden. Leider kann ich Ihnen an dieser Stelle keine allgemeine Empfehlung dafür geben, wie oft eine solche Reorganisation ausgeführt werden sollte. Die Häufigkeit ist natürlich abhängig vom Volumen der Datenänderungen auf der entsprechenden Datenbank. Sie werden also ein wenig experimentieren müssen, um die für Ihre Zwecke optimale Periode herauszufinden. Generell ist es selbstverständlich immer eine gute Idee, nach umfangreichen Datenänderungen (zum Beispiel nach Importen) auch die betroffenen Indizes zu reorganisieren. Ansonsten dürfte der Bedarf für eine allgemeine Reorganisation etwa zwischen täglich und wöchentlich schwanken.

6.2 Fehlende Indizes

In Kapitel 5 haben Sie gesehen, wie drastisch ein geeigneter Index die Abfrageleistung verbessern kann. Umgekehrt gilt natürlich auch, dass ein fehlender Index die Abfrageleistung ganz erheblich negativ beeinflussen kann. Das Erstellen der passenden Indizes ist daher eine wirklich wichtige Aufgabe, die in der Regel vom Datenbankentwickler und vom Datenbankadministrator gemeinsam erledigt werden sollte. Allerdings ist dieser Prozess dynamisch. Auch darauf wurde im vorangegangenen Kapitel bereits hingewiesen. Änderungen an Ihren Daten können dazu führen, dass bis zu einem gewissen Zeitpunkt alles perfekt läuft – und mit einem Mal dann Ihre bis dato erzeugten Indizes nicht mehr ausreichen. Mit anderen Worten: Es fehlen Indizes, und es gilt herauszufinden, welche Indizes dies sind, damit diese entsprechend hinzugefügt werden können.

Glücklicherweise protokolliert der Optimierer die Information über fehlende Indizes, und Sie haben die Möglichkeit, diese Information abzufragen. Hierzu gibt es verschiedene Möglichkeiten, auf die wir in den Kapiteln 9 und 11 näher eingehen werden. An dieser Stelle soll zunächst nur eine Einführung erfolgen.

6.2.1 Fehlende Indizes in gespeicherten Ausführungsplänen

Der Optimierer ist stets bestrebt, einen erstellten Ausführungsplan zur späteren Wiederverwendung im sogenannten Plan Cache zu speichern. Wie dies genau funktioniert, erfahren Sie in Kapitel 9. An dieser Stelle ist zunächst nur wichtig, dass in den gespeicherten Abfrageplänen, die im XML-Format abgelegt werden, auch Informationen über fehlende Indizes enthalten sind. Der Optimierer erstellt hier tatsächlich eine Berechnung, um welchen Prozentsatz sich die Ausführung einer Abfrage verbessern würde, sofern ein bestimmter Index vorhanden gewesen wäre. Im entsprechenden XML-Abfrageplan ist in einem solchen Fall eine Sektion `<MissingIndexes/>` enthalten. Da die gespeicherten Abfragepläne über die dynamische Systemsicht `sys.dm_exec_cached_plans` abgefragt werden können, eröffnet sich Ihnen die faszinierende Möglichkeit, auch bereits beendete Abfragen auf fehlende Indizes hin zu untersuchen. Es ist also nicht unbedingt erforderlich, Abfragen online, also während ihrer Ausführung, auf fehlende Indizes hin zu kontrollieren. Vielmehr können Sie tatsächlich die durch die Sicht `sys.dm_exec_cached_plans` zur Verfügung stehende Abfragehistorie verwenden, um einen Blick in die Vergangenheit zu werfen. Dazu müssen die entsprechenden Abfragen natürlich noch im Plan Cache existieren, was nicht in jedem Fall garantiert ist (hierauf kommen wir in Kapitel 9 noch einmal zurück).

Die Sektion `<MissingIndexes/>` wird also immer dann in den gespeicherten Abfrageplan aufgenommen, wenn der Optimierer wenigstens einen Index vermisst. Natürlich wollen wir hierzu wieder ein Beispiel untersuchen. Dafür verwenden wir die Tabelle `T1` und löschen zunächst den existierenden nichtgruppierten Index auf der Spalte `Nr`. Anschließend starten wir eine Abfrage, von der wir wissen, dass sie von dem gerade gelöschten Index profitiert hätte. Wir wollen die Abfrage jedoch nicht ausführen, sondern uns nur den XML-Abfrageplan anzeigen lassen. Das entsprechende Skript sieht so aus:

```
drop index Ix_T1_Nr on T1
go
set showplan_xml on
go
select * from T1 where Nr = 3000
```

Bei Ausführung des Skripts wird nur ein Link auf den Ausführungsplan im Ergebnisbereich angezeigt. Klicken Sie auf diesen Link, so öffnet sich der Ausführungsplan in grafischer Form (Abbildung 6.5).

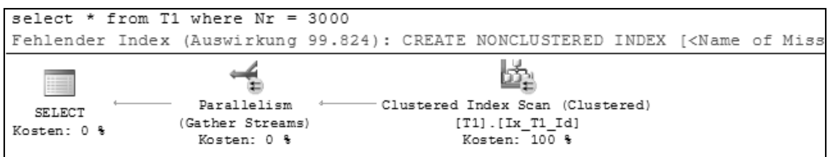


Abbildung 6.5: *Der grafische Ausführungsplan*

Bereits in dieser Ansicht ist zu erkennen, dass der Optimierer einen fehlenden Index beanstandet. (In der Abbildung ist dies aus Platzgründen nicht vollständig dargestellt.)

Aus dem Kontextmenü des grafischen Ausführungsplan heraus können Sie sich die Details für den fehlenden Index sofort anzeigen lassen (Abbildung 6.6). Diese Möglichkeit interessiert uns hier jedoch nicht. Wir wollen untersuchen, wie die entsprechende Sektion im XML-Abfrageplan aussieht.

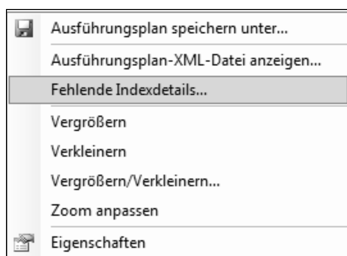


Abbildung 6.6:
Kontextmenü des Abfrageplans

Daher wählen wir den Punkt AUSFÜHRUNGSPLAN-XML-DATEI ANZEIGEN. Im angezeigten Plan finden Sie dann eine Sektion `<MissingIndexes/>`, die etwa so aussieht:

```
<MissingIndexes>
  <MissingIndexGroup Impact="99.824">
    <MissingIndex Database="[QueryTest]" Schema="[dbo]" Table="[T1]">
      <ColumnGroup Usage="EQUALITY">
        <Column Name="[Nr]" ColumnId="2" />
      </ColumnGroup>
      <ColumnGroup Usage="INCLUDE">
        <Column Name="[Id]" ColumnId="1" />
        <Column Name="[Platzhalter]" ColumnId="3" />
      </ColumnGroup>
```

```
    </MissingIndex>
  </MissingIndexGroup>
</MissingIndexes>
```

In dieser Form sind also die Informationen über fehlende Indizes in den gespeicherten Ausführungsplänen abgelegt.

Über die Systemfunktion `sys.dm_exec_query_plan` können diese Pläne abgefragt werden. Die Funktion kann allerdings immer nur die Information für einen einzigen Plan liefern, der durch den an diese Funktion übergebenen Parameter identifiziert wird. Dieser sogenannte `plan_handle` muss also bekannt sein. Die dynamische Systemsicht `sys.dm_exec_cached_plans` hilft hier weiter. Diese Sicht gibt unter anderem auch das jeweilige `plan_handle` eines gespeicherten Abfrageplans zurück. Unter Verwendung der beiden Systemsichten können wir die im Puffer befindlichen XML-Abfragepläne dann so abfragen:

```
select p.query_plan
   from sys.dm_exec_cached_plans
        cross apply sys.dm_exec_query_plan(plan_handle) as p
  where p.query_plan.exist(
         'declare namespace
          mi="http://schemas.microsoft.com/sqlserver/2004/07/showplan";
            //mi:MissingIndexes')=1
```

Über die `WHERE`-Klausel werden hier nur die Abfragepläne herausgefiltert, für die der Optimierer wenigstens einen fehlenden Index gefunden hat.

Die obige Abfrage bildet die Grundlage für das folgende, etwas komplexere Skript. Dieses Skript liefert Detailinformationen über alle fehlenden Indizes in den gespeicherten Abfrageplänen:

```
with XmlNameSpaces('http://schemas.microsoft.com/sqlserver/2004/07/showplan'
                      as qp)
  ,MissingIndexPlans(query_plan) as
   (
    select p.query_plan
      from sys.dm_exec_cached_plans
           cross apply sys.dm_exec_query_plan(plan_handle) as p
     where p.query_plan.exist(
            'declare namespace
             mi="http://schemas.microsoft.com/sqlserver/2004/07/showplan";
               //mi:MissingIndexes')=1
   )
  ,Statements(StatementId, StatementText, StatementType
             ,StatementCost, StatementRows, MissingIndexesXml) as
   (
     select stmt.value('(//qp:Statements/qp:StmtSimple)[1]/@StatementId'
                      ,'int')
       ,stmt.value('(//qp:Statements/qp:StmtSimple)[1]/@StatementText'
                      ,'nvarchar(max)')
       ,stmt.value('(//qp:Statements/qp:StmtSimple)[1]/@StatementType'
                      ,'nvarchar(80)')
```

```
    ,stmt.value('(//qp:Statements/qp:StmtSimple)[1]/@StatementSubTreeCost'
                ,'float')
    ,stmt.value('(//qp:Statements/qp:StmtSimple)[1]/@StatementEstRows'
               ,'float')
    ,stmt.query('//qp:MissingIndexes')
    from MissingIndexPlans
         cross apply query_plan.nodes('//qp:StmtSimple') as qp(stmt)
)
,MissingIndexGroup(StatementId, StatementText, StatementType
                  ,StatementCost, StatementRows
                  ,Impact, MissingIndexXml) as
(
  select StatementId, StatementText, StatementType
        ,StatementCost, StatementRows
        ,mi.value('@Impact', 'float')
        ,mi.query('.[position()]/qp:MissingIndex')
    from Statements
         cross apply MissingIndexesXml.nodes('//qp:MissingIndexGroup')
                     as mig(mi)
)
,MissingIndex(StatementId, StatementText, StatementType
         ,StatementCost, StatementRows
         ,Impact, DbName, TableName
         ,EqualityColumnsXml, InEqualityColumnsXml, IncludeColumnsXml) as
(
  select StatementId, StatementText, StatementType
        ,StatementCost, StatementRows
        ,Impact
        ,mi.value('@Database', 'sysname')
        ,mi.value('@Table', 'sysname')
        ,mi.query('//qp:ColumnGroup[@Usage="EQUALITY"]')
        ,mi.query('//qp:ColumnGroup[@Usage="INEQUALITY"]')
        ,mi.query('//qp:ColumnGroup[@Usage="INCLUDE"]')
    from MissingIndexGroup
         cross apply MissingIndexXml.nodes('//qp:MissingIndex') as mig(mi)
)
,ColumnGroup(StatementId, StatementText, StatementType
            ,StatementCost, StatementRows
            ,Impact, DbName, TableName
            ,IndexColumns, IncludeColumns) as
(
  select StatementId, StatementText, StatementType
        ,StatementCost, StatementRows
        ,Impact, DbName, TableName
        ,ltrim(replace(cast(
     EqualityColumnsXml.query('data(//qp:Column/@Name)') as nvarchar(max))
            + ' '
            + cast(InEqualityColumnsXml.query('data(//qp:Column/@Name)')
                   as nvarchar(max)), '] [','],['))
```

```
          ,replace(cast(IncludeColumnsXml.query('data(//qp:Column/@Name)')
                    as nvarchar(max)), '] [','],[')
       from MissingIndex
   )
select StatementId, StatementText, StatementType
      ,StatementCost, StatementRows
      ,Impact, DbName, TableName, IndexColumns, IncludeColumns
  from ColumnGroup
```

Abbildung 6.7 zeigt ein Beispiel für das Ergebnis der obigen Abfrage für den fehlenden Index in unserer Tabelle T1.

StatementId	StatementText	StatementType	StatementCost	StatementRows	Impact	DbName	TableName	IndexColumns	IncludeColumns
1	(@1 smallint)SEL...	SELECT	22,6322	1	99,824	[QueryTest]	[T1]	[Nr]	[Id],[Platzhalter]

Abbildung 6.7: *Fehlende Indizes in gespeicherten Abfrageplänen*

Die Abfrage der Informationen über fehlende Indizes direkt aus den gespeicherten Ausführungsplänen ist bereits recht komplex. Seien Sie bitte vorsichtig mit dem Einsatz dieser Abfrage in Produktivsystemen. Sie ist sehr »ressourcenhungrig«.

Der folgende Abschnitt zeigt eine Möglichkeit, wie Sie auch ohne XML-Abfragepläne fehlende Indizes aufspüren.

6.2.2 Die sys.dm_db_missing_index...-Systemsichten

Sie müssen nicht XQuery verstehen, um an Informationen über fehlende Indizes zu gelangen. SQL Server bietet hierfür vier Systemsichten an, die Sie direkt abfragen können:

- **sys.dm_db_missing_index_groups**. Fehlende Indizes werden in Gruppen zusammengefasst. Diese Sicht enthält nur zwei Schlüsselspalten, für die Verknüpfungen zu den anderen missing_index-Sichten.
- **sys.dm_db_missing_index_group_stats**. Über diese Sicht können statistische Informationen zu einer Gruppe fehlender Indizes abgefragt werden. Hier sind insbesondere die folgenden Spalten interessant:

  user_seeks. Die Anzahl von Suchen im Indexbaum, für die der fehlende Index verwendet worden wäre.

  user_scans. Die Anzahl sequenzieller Suchen (Scans), für die der fehlende Index verwendet worden wäre.

  last_user_seek. Der letzte Zeitpunkt, zu dem der fehlende Index für eine Suche verwendet worden wäre.

  last_user_scan. Der letzte Zeitpunkt, zu dem der Index sequenziell durchsucht worden wäre.

  avg_user_cost. Der Mittelwert der Abfragekosten für alle Abfragen, die von dem fehlenden Index profitiert hätten.

  avg_user_impact. Die geschätzte Verbesserung der Abfragekosten, sofern der Index existiert hätte.

Die oben aufgelisteten Spalten beziehen sich sämtlich auf Benutzerabfragen. Für alle diese Spalten gibt es entsprechende `system_` -Äquivalente (zum Beispiel `system_scans`), in welchen die Informationen bezüglich der Abfragen enthalten sind, die durch Systemprozesse ausgeführt wurden.

- **`sys.dm_db_missing_index_details`**. In dieser Sicht finden Sie die Informationen zur Erzeugung der fehlenden Indizes. Hierfür sind die folgenden Spalten interessant:

  `database_id`. Dies ist die ID der Datenbank, in welcher der Index fehlt.

  `statement`. Der Name der Spalte ist etwas irreführend. Hier wird nicht die SQL-Anweisung genannt, die von dem Index profitiert hätte, sondern lediglich der Name der Tabelle, für die der Index erzeugt werden sollte. Möglicherweise deutet der Name der Spalte darauf hin, dass in einer kommenden Version hier tatsächlich die SQL-Anweisung aufgeführt wird.

  `equality_colums`. Diese Spalte enthält eine Liste derjenigen Indexspalten, die für einen exakten Vergleich der Art *spalte* = *wert* nützlich sind.

  `inequality_columns`. In dieser Spalte sind die Indexspalten aufgeführt, von der eine Bereichssuche (also zum Beispiel ein Vergleich der Art *spalte* < *wert*) profitiert hätte.

  `included_columns`. Hier finden Sie eine Liste derjenigen Spalten, die über die `INCLUDE`-Klausel in den Index aufgenommen werden sollten.

- **`sys.dm_db_missing_index_columns`**. Dies ist keine Sicht, sondern eine Tabellenwert-Funktion. Die Funktion erwartet die Id des vorgeschlagenen Index (wie von `sys.dm_db_missing_index_groups` zurückgeliefert) und gibt eine Tabelle mit den Spalten für den fehlenden Index aus. Die enthaltene Information ist dieselbe wie die von `sys.dm_db_missing_index_details` zurückgegebene.

Die folgende Abfrage verwendet die `missing_index`-Systemsichten, um fehlende Indizes aufzuspüren:

```
select db_name(d.database_id) as db_name
      ,d.statement
      ,d.equality_columns, d.inequality_columns
      ,d.included_columns
      ,cast(gs.avg_total_user_cost as decimal(8, 2)) as avg_total_user_cost
      ,gs.avg_user_impact
      ,gs.user_seeks, gs.user_scans
      ,gs.last_user_seek, gs.last_user_scan
  from sys.dm_db_missing_index_groups as g
       inner join sys.dm_db_missing_index_group_stats as gs
               on gs.group_handle = g.index_group_handle
       inner join sys.dm_db_missing_index_details as d
               on g.index_handle = d.index_handle
 where d.database_id > 4  -- Nur Benutzerdatenbanken
```

Die Funktion `sys.dm_db_missing_index_columns` wird hierfür nicht benötigt.

Einen Ausschnitt des Ergebnisses der Abfrage zeigt Abbildung 6.8.

db_name	statement	equality_columns	inequality_columns	included_columns	avg_user_impact
QueryTest	[QueryTest].[dbo].[T1]	[Nr]	NULL	[Id], [Platzhalter]	99,82

Abbildung 6.8: *Fehlender Index aus »missing_index«-Systemsichten*

Wie Sie sicherlich erkennen, sind diese Informationen nicht ganz so umfangreich wie diejenigen, welche in den gespeicherten Abfrageplänen enthalten sind. Insbesondere fehlt der Bezug zur SQL-Anweisung. Demgegenüber steht aber der große Vorteil, dass Abfragen der `missing_index`-Systemsichten wesentlich »leichtgewichtiger« sind. XQuery-Abfragen der im XML-Format gespeicherten Abfragepläne können Ihren Server im laufenden Betrieb ganz erheblich belasten.

Die Online-Dokumentation nennt eine Reihe von Einschränkungen, die Sie beachten sollten, wenn Sie fehlende Indizes auf die hier beschriebene Art suchen. Ich möchte diese Einschränkungen hier nicht alle wiederholen, sondern Sie bitten, dies in der Dokumentation nachzulesen. Als Fazit bleibt hier nur zu sagen, dass Sie die Indexempfehlungen nicht unüberlegt anwenden sollten, sondern sie als das verstehen sollen, was sind: nämlich als Empfehlungen. So kann es beispielsweise vorkommen, dass der Optimierer die Reihenfolge der Indexspalten nicht korrekt angibt.

Wohlgemerkt: Die vom Optimierer generierten Informationen über fehlende Indizes sind äußerst hilfreich beim Aufspüren von Performance-Engpässen. Es lohnt sich aber in jedem Fall, diese Informationen einer gewissenhaften Prüfung zu unterziehen. Schauen Sie sich bitte noch einmal Abbildung 6.8 an. Dort wird auch die Spalte `ID` als Kandidat für die eingeschlossenen Spalten genannt. Dies ist ganz sicher nicht erforderlich, da wir für diese Spalte einen gruppierten, eindeutigen Index erzeugt haben. Somit ist die Spalte `ID` als Schlüsselspalte des gruppierten Index ja sowieso schon in jedem nichtgruppierten Index enthalten und muss daher nicht nochmals in den Index aufgenommen werden. Allerdings können Sie die Spalte getrost in den Index einschließen. SQL Server ist intelligent genug, die Information nicht doppelt zu speichern. Normalerweise dürfen Sie diese Spalten ignorieren. Lediglich eine Änderung des gruppierten Index, bei welcher wenigstens eine der Indexspalten wegfällt, hätte dann zur Folge, dass die entsprechenden Spalten tatsächlich im nichtgruppierten Index fehlen würden.

Abschließend möchte ich Sie noch einmal daran erinnern, dass sowohl die in den Abfrageplänen gespeicherten Informationen als auch die Informationen in den dynamischen `missing_index`-Systemsichten flüchtig sind. Allerspätestens beim Neustart des SQL Server-Dienstes sind diese »historischen« Informationen nicht mehr verfügbar. Auch im laufenden Betrieb kann es vorkommen, dass zum Beispiel Abfragepläne aus dem Plan Cache entfernt werden, etwa dann, wenn SQL Server Speicher an das Betriebssystem zurückgibt.

Wenn Sie die Informationen dauerhaft speichern möchten, können Sie das Ergebnis der Abfragen in entsprechenden Tabellen einfügen. Dies kann recht einfach über eine `INSERT SELECT`-Anweisung geschehen, die in zyklischen Abständen – zum Beispiel über einen SQL Server Agent-Auftrag – ausgeführt wird.

In Kapitel 11 werden wir noch einmal auf die permanente Speicherung der Information über fehlende Indizes zurückkommen; dann unter Verwendung einer selbst definierten Datenauflistung. Diese Möglichkeit ist neu in SQL Server 2008.

6.3 Überflüssige Indizes

Da die passenden Indizes zu einer enormen Verbesserung der Abfrageleistung beitragen, kommt es häufig vor, dass Datenbanken »überindiziert« sind. Hiermit meine ich, dass Indizes oftmals nach dem Motto »Lieber zu viel als zu wenig« angelegt werden. Prinzipiell ist gegen eine solche Vorgehensweise auch nichts einzuwenden. Meist ist es ja so, dass man sowieso schon unter Zeitdruck steht – und nun kommt auch noch jemand daher und beschwert sich über mangelnde Performance. In vielen Fällen wird man dann einfach mit Indizes herumprobieren und hierbei auch oft eine akzeptable Lösung finden. Allerdings entstehen dabei eventuell auch Indizes, die nicht unbedingt sinnvoll sind. Hinzu kommt, dass durch Änderungen an den Daten und den Anwendungen einige Indizes überflüssig werden. Überflüssig sind zunächst einmal alle Indizes, die nicht für eine Suche in irgendeiner Form verwendbar sind. Solche Indizes verbrauchen unnötig Speicherplatz auf der Festplatte und verlangsamen natürlich Datensicherungen und Datenwiederherstellungen.

Schlimmer wird es dann, sobald Datenänderungen ins Spiel kommen. Jede Änderung muss auch immer die entsprechenden Indizes aktualisieren. Und so behindern Indizes, die für Suchen nicht verwendet werden, Aktualisierungsoperationen. Insbesondere `INSERT`- und `DELETE`-Anweisungen führen dazu, dass immer sämtliche Indizes einer Tabelle aktualisiert werden müssen.

Generell gilt aber, dass überflüssige Indizes sich bei Weitem nicht so dramatisch auf die Abfrageleistung auswirken, wie fehlende Indizes.

Allerdings sind überflüssige Indizes recht schwer zu entdecken. Immerhin ist es vorstellbar, dass ein Index über einen relativ langen Zeitraum nicht für eine Suche – in welcher Form auch immer – verwendet wurde. Dies kann durchaus vorkommen, beispielsweise wenn ein Index nur für Abfragen, die jeweils zum Quartalsabschluss ausgeführt werden, sinnvoll ist. Es ist auch möglich, dass Ihre Datenbank im initialen Zustand eine Reihe sehr kleiner Tabellen enthält, für die anfänglich einfach auf Grund ihrer Größe stets ein Table Scan durchgeführt wird. Damit sind zunächst einmal alle Indizes für eine solche Tabelle überflüssig. Mit einer zunehmenden Anzahl Zeilen wird dann ein existierender Index eventuell mehr und mehr sinnvoll.

Es ist also erforderlich, das System zu überwachen, um nicht benötigte Indizes zu entdecken. Finden Sie solche Indizes, so gehört allerdings noch eine gehörige Portion Mut dazu, diese Indizes aus dem System zu entfernen.

Sicherlich haben Sie bereits vermutet, dass SQL Server auch für eine Suche nach nicht verwendeten Indizes dynamische Systemsichten zur Verfügung stellt.

Für die Abfrage nach der Indexverwendung können Sie die Sicht `sys.dm_db_index_usage_stats` verwenden. Diese Sicht enthält entsprechende Statistiken. Sie kann leicht mit der Sicht `sys.indexes` verknüpft werden, um die Namen der Indizes und Tabellen zu erhalten:

```
select object_name(i.object_id) as TableName
      ,i.type_desc,i.name
      ,us.user_seeks, us.user_scans
      ,us.user_lookups,us.user_updates
      ,us.last_user_scan, us.last_user_update
  from sys.indexes as i
       left outer join sys.dm_db_index_usage_stats as us
                    on i.index_id=us.index_id
                   and i.object_id=us.object_id
 where objectproperty(i.object_id, 'IsUserTable') = 1
```

Die folgenden Spalten der Sicht `sys.dm_db_index_usage_stats` sind hierbei von Bedeutung:

- **`user_seeks`, `last_user_seek`**. Gibt Auskunft über die Anzahl Suchen im Indexbaum und den Zeitpunkt der letzten Suche. Der Indexbaum wird also für Suchoperationen verwendet. Das ist gut so, denn genau dafür erzeugt man ja einen Index. Ein Index, der niemals für eine Suchoperation verwendet wird, sollte auf jeden Fall einer genaueren Prüfung unterzogen werden. Sehr wahrscheinlich ist ein solcher Index überflüssig.
- **`user_scan`, `last_user_scan`**. Enthält die Anzahl serieller Suchen (Scans) sowie den Zeitpunkt des letzten Scans. Der Optimierer entscheidet sich für einen Index-Scan, wenn die Menge der zu durchsuchenden Daten im Index kleiner ist als die bei einem Scan der Tabelle und der Index alle Daten enthält, die für die Ausführung der Abfrage erforderlich sind. Das folgende Beispiel verdeutlicht dies:

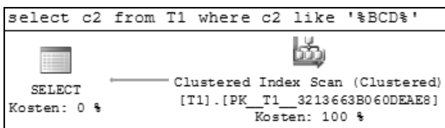
  Wir legen zunächst eine Tabelle an und fügen einige Zeilen hinzu:

  ```
  set nocount on
  use QueryTest;
  if (object_id('T1', 'U') is not null)
    drop table T1
  go
  create table T1
   (
    c1 int not null identity(1,1) primary key
   ,c2 nvarchar(80) not null default '*'
   ,c3 char(500) not null default '*'
   )
  go

  insert T1(c2) values(newid())
  go 3

  insert T1(c2) select newid() from T1
  go 10
  ```

  Bei der folgenden Abfrage entscheidet sich der Optimierer für einen Clustered Index Scan:

  ```
  set statistics io on
   select c2 from T1 where c2 like '%BCD%'
  set statistics io off
  ```

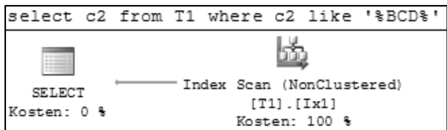
Durch das Einschalten der Statistikoptionen für die Abfrage bekommen wir die Information, dass 239 logische Lesevorgänge erforderlich sind. Hier noch der zugehörige Abfrageplan:



Wir erzeugen nun einen nichtgruppierten Index auf der Spalte `c2`:

```
create index Ix1 on T1(c2)
```

Anschließend liefert die Ausführung der obigen `SELECT`-Anweisung den folgenden Abfrageplan:



Der erzeugte Index kann nicht für eine Suche im Indexbaum verwendet werden, weil die `WHERE`-Bedingung eine Zeichenkette mitten in der indizierten Spalte herausfiltert. Eine Suche im Indexbaum ist nur dann sinnvoll, wenn die Sortierung des Index für die Suche verwendet werden kann – und das ist im Beispiel nicht der Fall. Der Index enthält jedoch alle Daten, die die Abfrage entweder zurückliefert oder herausfiltert. Da diese Datenmenge wesentlich kleiner ist als die Daten der gesamten Tabelle, können durch einen Index-Scan gegenüber dem Tabellen-Scan Leseoperationen eingespart werden. Insgesamt wurden für den Index-Scan nur 34 logische Lesevorgänge (gegenüber 239 für den Tabellen-Scan) benötigt.

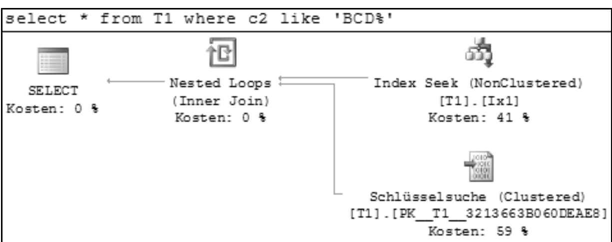
Interessant ist hier die Tatsache, dass der Optimierer den Index nicht als fehlend deklariert hat (er wird im obigen Ausführungsplan nicht erwähnt), obwohl der Index eine Verbesserung um ungefähr den Faktor 7 bewirkt. Ganz offensichtlich sucht der Optimierer derzeit nur solche Indizes, die für eine Suche im Indexbaum verwendet werden können.

Eine Entscheidung zu treffen, ob die Verwendung eines Index in einer Scan-Operation als positiv oder negativ einzustufen ist, ist sehr schwierig. Im obigen Beispiel ist der Index durchaus sinnvoll, auch wenn er niemals für eine Suchoperation verwendet wird. Allerdings legt man hierfür normalerweise keinen Index an. Wann immer ein Index erstellt wird, ist die Intention sicherlich die Verwendung des Indexbaums für Suchoperationen (Seeks).

- **`user_lookups`, `last_user_lookup`**. Hier stehen die Anzahl von Lookups und der Zeitpunkt des letzten Lookups. Index-Lookups werden nur für gruppierte Indizes ausgeführt. Ein Lookup ist immer dann erforderlich, wenn ein nichtgruppierter Index zwar für die Suche verwendet wird, aber nicht alle Daten enthält, die die Abfrage zurückgeben soll. Das bedeutet, sass der nichtgruppierte Index nicht abdeckend ist. In einem solchen Fall müssen die fehlenden Daten aus dem gruppierten Index sozusagen nachgeladen werden. Für die Tabelle und den Index aus dem obigen Beispiel können wir die folgende Abfrage ausführen:

```
select * from T1 where c2 like 'BCD%'
```

Der Abfrageplan sieht so aus:



Hier kann der vorhandene Index zwar für eine Suche herangezogen werden, da aber die Abfrage auch Spalten zurückgeben soll, die nicht im Index `Ix1` enthalten sind, müssen die Werte für diese Spalten aus dem gruppierten Index geholt werden. Dies geschieht über die Schlüsselsuche (also den Lookup) – übrigens eine relativ teure Operation.

- **`user_updates`, `_last_user_update`**. In diesen Spalten finden Sie die Anzahl der Aktualisierungen und den Zeitpunkt der letzten Aktualisierung des entsprechenden Index. Jedes `INSERT`, `UPDATE` oder `DELETE` auf einer Tabelle muss stets auch die Indizes aktualisieren. Während bei einem `UPDATE` lediglich die Indizes für die tatsächlich geänderten Spalten aktualisiert werden müssen, werden bei `INSERT` und `DELETE`-Operationen auf der Tabelle stets alle Indizes aktualisieret. Je nach Index kann eine solche Aktualisierung relativ »teuer« werden. Dies gilt insbesondere für den gruppierten Index einer Tabelle, der ja die Daten in sortierter Form enthält. Bei jeder Aktualisierung des gruppierten Index sind daher eventuell relativ kostspielige Umsortierungen erforderlich.

Für alle oben aufgeführten Spalten gibt es entsprechende System-Äquivalente (also zum Beispiel `system_seeks`). Dort stehen jeweils dieselben Informationen, nur eben bezogen auf Systemprozesse.

Als einfache Faustregel können Sie sich merken, dass Indizes, die niemals in einem Seek oder Lookup verwendet werden, einmal genauer unter die Lupe genommen werden sollten.

Insbesondere gilt die folgende Regel:

Ein Index, der nur für Updates verwendet wird und niemals in einer der anderen Kategorien (Seeks, Scans oder Lookups) auftaucht, ist mit Sicherheit überflüssig. Ein solcher Index ist für die Performance nicht nur nicht von Nutzen, er behindert die Performance durch die unnötigen Aktualisierungen auch noch.

Denken Sie bitte daran, dass die Information über die Indexverwendung ebenfalls nicht persistent gespeichert wird und spätestens beim Neustart von SQL Server verloren ist. Da eine fundierte Aussage darüber, ob ein Index wirklich überflüssig ist, nur getroffen werden kann, wenn Sie Ihr System über einen längeren Zeitraum (in der Regel mehrere Monate)

beobachten, ist es hier besonders wichtig, dass Sie diese Information permanent speichern. Natürlich geht dies auch in diesem Fall über einen entsprechenden Auftrag des SQL Server Agents, in welchem Sie das Ergebnis der Abfrage zur Indexverwendung in eine Tabelle einfügen. Hinzu kommt hier, dass die in der Sicht enthaltene Anzahl der Such- oder Scan-Vorgänge stets kumulativ ist. So ist es zum Beispiel nicht möglich, die Information zu erhalten, welcher Index wie oft für Seeks in der vergangenen Woche verwendet wurde. Falls Sie diese Information benötigen, müssen Sie in regelmäßigen Abständen Snapshots der Systemsicht speichern und später die Werte verschiedener Snapshots miteinander vergleichen.

Wir werden in Kapitel 11 hierzu ein Beispiel mit Datenauflistungen behandeln, die eine komfortable Möglichkeit zur Speicherung der Informationen über fehlende und überflüssige Indizes zur Verfügung stellen. Dort sehen Sie dann auch, wie mit Snapshots »gerechnet« wird, um Informationen über einen bestimmten Zeitraum zu erhalten.

6.4 Zusammenfassung

Dieses Kapitel hat Ihnen gezeigt, wie Sie mit Indizes im laufenden Betrieb umgehen. Sie wissen nun, dass ein Satz existierender Indizes nicht statisch ist. Vielmehr ist es erforderlich, dass Sie existierende Indizes in zweierlei Hinsicht überwachen:

1. Ist ein Index fragmentiert, muss er reorganisiert oder neu erstellt werden.
2. Falls Sie herausfinden, dass ein Index überflüssig ist, können Sie mutig sein und den Index löschen.

Hinzu kommt, dass Sie auch die Möglichkeit haben herauszufinden, ob Indizes vermisst werden. Solche Indizes sollten Sie natürlich hinzufügen.

Es ist sicherlich deutlich geworden, dass Sie eine entsprechende Strategie für die laufende Kontrolle Ihres Systems benötigen.

7 Partitionierung

Partitionierung bedeutet Aufteilung von logisch zusammengehörenden Tabellen- oder Indexdaten auf unterschiedliche Speicherorte. Das Ziel einer Partitionierung ist immer die Reduzierung bzw. Beschleunigung von Ein- und Ausgabevorgängen. Generell ist eine Partitionierung nur lohnend bei sehr großen Datenmengen, also für große Tabellen oder Indizes.

Wenn Sie vorhersehen können, wie auf Ihre Daten zugegriffen wird, dann können Sie die Daten in Partitionen aufteilen. Kriterien für diese Aufteilung sind zum Beispiel die Häufigkeit und die Art des Zugriffs. So können Sie etwa Bereiche, auf die normalerweise nur lesend zugegriffen wird, von den Bereichen trennen, die Veränderungen unterworfen sind. Selten verwendete Daten können Sie auf langsame Festplatten auslagern, damit Ihr Hochleistungs-RAID für die tatsächlichen Bewegungsdaten zur Verfügung steht.

Für die Versuche in diesem Kapitel verwenden wir eine einfache Tabelle mit Kundendaten. Diese Tabelle wird wie folgt erzeugt:

```
use QueryTest;
if (object_id('Kunde', 'U') is not null)
  drop table Kunde
go
create table Kunde
 (
   Nr int not null primary key nonclustered
  ,VorName nvarchar(80) not null
  ,NachName nvarchar(80) not null
  ,LetzteBestellung date not null
  ,MehrSpalten nchar(500) not null default '#'
 )
go
-- Füge 30.000 Zeilen hinzu
insert Kunde (Nr, VorName, NachName, LetzteBestellung)
 select n,newid(),newid()
       ,dateadd(d, -(abs(checksum(newid())) % 5000), getdate())
   from Numbers where n <= 30000
go
-- Erstelle gruppierten Index
create clustered index IxKdBestDat
   on Kunde(LetzteBestellung)
```

Weiter wollen wir für alle Abfragen zum Messen der Leistung die E/A-Statistiken anzeigen. Wie Sie aus Kapitel 4 bereits wissen, geschieht dies über das folgende Kommando:

```
set statistics io on
```

7.1 Horizontale Partitionierung

Seit der Version 2005 unterstützt SQL Server direkt die horizontale Partitionierung. Dieses Feature steht Ihnen allerdings ausschließlich in der Enterprise Edition zur Verfügung. Dabei werden Tabellen- oder Indexdaten letztlich in unterschiedlichen physikalischen Speicherorten abgelegt. Wir betrachten in der Folge nur Tabellen; für Indizes gilt das hier Gesagte aber analog.

Bei einer horizontalen Partitionierung wird die ursprüngliche Tabelle zeilenweise in verschiedene Bereiche aufgeteilt. Die abgeleiteten Bereiche entsprechen dabei in ihrer Struktur der Originaltabelle (Abbildung 7.1).

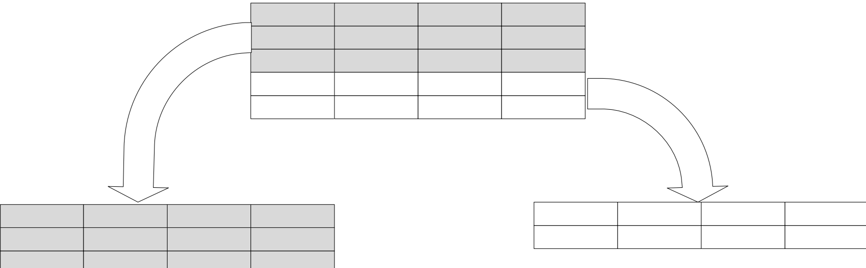


Abbildung 7.1: *Horizontale Partitionierung einer Tabelle*

Wie bereits in der Einleitung zu diesem Kapitel erwähnt, ist dies immer dann sinnvoll, wenn unterschiedliche Abfragen den Zugriff auf verschiedene Partitionen hervorrufen. Wenn zum Beispiel Aktualisierungen nur für die Daten des aktuellen Monats ausgeführt werden, Ihr Berichtssystem aber vorrangig auf Daten der vergangenen Monate zugreift, dann kann eine Aufteilung der Daten in zwei Partitionen sinnvoll sein. In unserer Kunden-Tabelle könnten wir die Spalte `LetzteBestellung` für eine Aufteilung in Partitionen verwenden und dadurch zum Beispiel Kunden, deren letzte Bestellung lange zurück liegt und auf die daher vermutlich nur sehr selten zugegriffen wird, in einer separaten Partition speichern.

Für das Einrichten einer horizontalen Partitionierung sind insgesamt vier Schritte erforderlich:

1. Planen der Partitionen und Einrichten der Speicherorte. Natürlich müssen Sie zunächst festlegen, welche Kriterien für die Partitionierung verwendet werden, und die entsprechenden Dateigruppen mit den zugehörigen Datenbankdateien einrichten.
2. Erstellen einer sogenannten Partitionsfunktion. Dies ist eine Entscheidungsfunktion, die anhand einer Regel festlegt, in welcher Partition welche Daten abgelegt werden.
3. Anlegen eines Partitionsschemas. Ein Partitionsschema ist quasi der »Klebstoff«, der die Partitionsfunktion mit den unterschiedlichen Speicherorten verbindet.
4. Erzeugen der partitionierten Tabelle. Diese Tabelle legen Sie im Partitionsschema an. Hierdurch ist dafür gesorgt, dass unterschiedliche Tabellendaten letztlich in unterschiedlichen Partitionen landen

Die Online-Dokumentation beschreibt sehr genau, wie Sie Partitionen entsprechend Ihren Anforderungen einrichten. Besser könnte ich es wirklich nicht erklären; und daher bitte ich Sie, in der Dokumentation nachzulesen, wenn Sie sich hier detaillierter einarbeiten möchten.

Was Sie auf jeden Fall bedenken sollten ist der Umstand, dass eine Änderung der Partitionierung nicht in jedem Falle einfach durchgeführt werden kann. Es ist relativ leicht, eine bestehende Partitionierung zu erweitern, also neue Partitionen am oberen Ende hinzuzufügen. Recht komplex sind hingegen eine Zusammenführung von Partitionen oder das Aufteilen einer existierenden Partition in mehrere neue Partitionen. Aber immerhin bietet das Management Studio nun die Möglichkeit, diese Aufgabe über die grafische Oberfläche zu erledigen. In der Vergangenheit war dies nur über T-SQL möglich.

Wir werden in Kapitel 11 sehen, wie man eine horizontale Partitionierung auch durch die Wahl geeigneter Indizes erreichen kann.

Ein wichtiger Aspekt bei der horizontalen Partitionierung ist die Möglichkeit, dass Sie die Dateigruppen, in welchen die Partitionen abgelegt sind, separat sichern und wiederherstellen können. In der Enterprise Edition von SQL Server ist eine Wiederherstellung unter Umständen sogar im Online-Betrieb möglich. Dadurch erhöht sich eventuell die Verfügbarkeit Ihres Gesamtsystems. Stellen Sie sich vor, dass Ihre historischen Kundendaten in einer 10 TByte großen Dateigruppe gespeichert sind. Die Daten der aktuellen Kunden sollen ebenfalls in einer eigenen Dateigruppe gespeichert werden, die aber nur 2 GByte groß ist. Wenn nun die Dateigruppe mit den aktuellen Daten ausfällt, so können Sie diese eine Dateigruppe mit 2 GByte wesentlich schneller wiederherstellen als die gesamte Datenbank mit 10 TByte. Dieser administrative Aspekt ist sicherlich ein besseres Argument für eine horizontale Partitionierung als ein möglicherweise zu erwartender Performance-Gewinn.

7.1.1 Partitionierte Sichten

SQL kennt den `UNION`-Operator, mit dem Resultate verschiedener Abfragen zusammengefasst werden können. Dadurch erhalten Sie die Möglichkeit, eine Tabelle in unterschiedliche Tabellen gleicher Struktur aufzuteilen und die einzelnen Tabellen anschließend in einer Sicht wieder zusammenzufassen. Diese Sicht besteht aus `SELECT`-Anweisungen, die die einzelnen Tabellen abfragen und die über den `UNION`-Operator aneinandergehängt sind. Bei Abfragen über diese Sicht entfernt der Optimierer unnötige `SELECT`-Anweisungen. Eine solche Technik ist besonders dann sinnvoll, wenn Sie große Tabellen so auf unterschiedliche SQL Server aufteilen können, dass die Partitionen standortbezogen sind.

Stellen Sie sich bitte einmal vor, dass in der Kunden-Tabelle Kunden aus allen Standorten zusammengefasst werden. Normalerweise wird jeder Standort aber nur mit seinen eigenen Kunden arbeiten. In einem solchen Fall können Sie die Kunden-Tabelle auf die verschiedenen Standorte aufteilen und über eine Sicht mit `UNION`-Befehlen zu einer Gesamt-Kunden-Tabelle zusammenfassen. Jeder Standort wird dann normalerweise nur die Daten aus seiner lokalen Kunden-Tabelle abfragen; diese lokalen Abfragen sind relativ schnell. Es existiert dadurch auch die Möglichkeit, an jedem Standort mit Kunden aus allen Standorten arbeiten zu können, was aber nur sehr selten vorkommen wird. Da dies

zu einem Zugriff auf entfernte Server führen würde, würde solch eine (seltene) Abfrage dann natürlich etwas länger dauern. Insgesamt hat eine solche Lösung aber Vorteile gegenüber einer Verfahrensweise, bei der die Kunden-Tabelle zentral auf einem Server liegt, der für alle Standorte ein entfernter Server ist.

Eine solche Sicht ist übrigens nicht aktualisierbar. Ist dies gewünscht, so müssen Sie entsprechende `INSTEAD OF`-Trigger implementieren.

7.2 Vertikale Partitionierung

Bei einer vertikalen Partitionierung wird eine Basistabelle in mehrere abgeleitete Tabellen aufgeteilt (Abbildung 7.2). Dies kann zum Beispiel dann sinnvoll sein, wenn eine Tabelle über viele Spalten verfügt, von denen die meisten nur sehr selten verwendet werden. Eine vertikale Partitionierung kann sich auch lohnen, wenn in einer Tabelle sehr breite Spalten enthalten sind, auf die nur in bestimmten Szenarien zugegriffen wird.

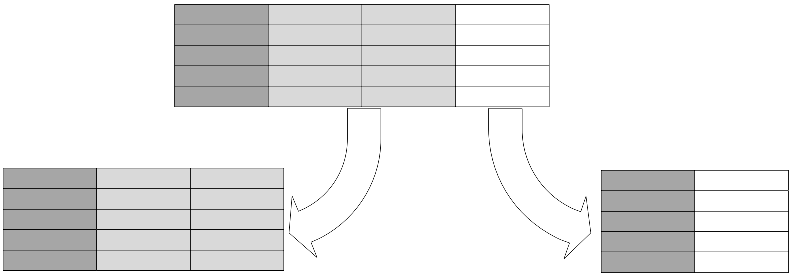


Abbildung 7.2: *Vertikale Partitionierung*

In Abbildung 7.2 ist die Aufteilung der Originaltabelle in zwei Tabellen dargestellt. Eine solche Aufteilung kann selbstverständlich auch physikalisch erfolgen, indem Sie die abgeleiteten Tabellen in verschiedenen Dateigruppen unterbringen. Natürlich können Sie durchaus auch mehr als zwei Tabellen aus der ursprünglichen Tabelle ableiten. Die abgeleiteten Tabellen stehen hierbei in einer 1:1-Beziehung zueinander. Wichtig ist es daher, dass alle Zeilen in allen Tabellen dieselbe Zeilenidentität besitzen. Dies erfordert, dass der Primärschlüssel in allen abgeleiteten Tabellen dupliziert wird. In der Abbildung wird dies durch die erste (dunkelgraue) Spalte verdeutlicht. Dieser Umstand verkompliziert die Sache natürlich, wenn Sie für Primärschlüsselspalten automatisch vergebene Werte (also zum Beispiel `IDENTITY`-Spalten) verwenden.

Sie können die partitionierten Tabellen in einer Sicht wieder zusammenführen und diese Sicht für Abfragen verwenden. Der Optimierer analysiert die Spalten in `SELECT`-Anweisungen und entfernt nicht benötigte Tabellen aus den JOINs.

Lassen Sie uns hierfür noch einmal unsere einfache Kunden-Tabelle hernehmen. Betrachten Sie bitte die folgende Abfrage:

```
select Nr, NachName, LetzteBestellung
  from Kunde
 where LetzteBestellung >= '20080101'
```

Die Abfrage führt zu einer Suche im gruppierten Index und benötigt 269 logische Leseoperationen. Wir wollen davon ausgehen, dass die Anwendung in den meisten Fällen nur die Spalten `Nr`, `LetzteBestellung` und `Nachname` verwendet. Daher teilen wir die Tabelle `Kunde` wie folgt auf:

```
create table KundeBasis
 (
  Nr int not null primary key nonclustered
 ,NachName nvarchar(80) not null
 ,LetzteBestellung date not null
 )
go
-- Erstelle gruppierten Index
create clustered index IxKdBasisBestDat
   on KundeBasis(LetzteBestellung)
go
create table KundeErweitert
 (
  Nr int not null primary key clustered
 ,VorName nvarchar(80) not null
 ,MehrSpalten nchar(500) not null default '#'
 )
go
alter table KundeErweitert
  add constraint FK_KundeBasis foreign key (Nr)
    references KundeBasis(Nr)
```

Wir übertragen nun alle Kunden aus der ursprünglichen Tabelle in die beiden abgeleiteten Tabellen und entfernen die Tabelle `Kunde`:

```
insert KundeBasis(Nr, NachName, LetzteBestellung)
  select Nr, NachName, LetzteBestellung
    from Kunde
go
insert KundeErweitert(Nr, VorName, MehrSpalten)
  select Nr, VorName,MehrSpalten from Kunde
go
drop table Kunde
```

Nun erstellen wir die nachfolgende Sicht, in welcher die beiden Tabellen wieder zusammengeführt werden:

```
create view Kunde as
 select KundeBasis.Nr, KundeBasis.NachName, KundeBasis.LetzteBestellung
       ,KundeErweitert.VorName, KundeErweitert.MehrSpalten
  from KundeBasis
       left outer join KundeErweitert
                    on KundeErweitert.Nr = KundeBasis.Nr
```

Der LEFT JOIN ist hier erforderlich, weil der Optimierer die Tabelle `KundeErweitert` andernfalls nicht eliminieren kann, wenn keine Spalten aus dieser Tabelle abgefragt werden.

Wenn Sie nun die obige Abfrage noch einmal ausführen, so führt dies zu einer Abfrage der Sicht `Kunde` (wir haben ja die Tabelle durch die Sicht ersetzt). Da die Abfrage nur auf Spalten aus der Tabelle `KundeBasis` zugreift, entfernt der Optimierer einfach den JOIN auf die Tabelle `KundeErweitert`. Im Ausführungsplan ist dies klar erkennbar (Abbildung 7.3).

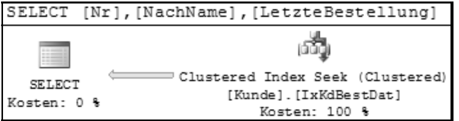


Abbildung 7.3: *Suche nur im gruppierten Index der Tabelle »KundeBasis«*

Die ausgegebene E/A-Statistik nennt 22 logische Lesevorgänge. Das sind immerhin nur etwa 80 Prozent des ursprünglichen Wertes ohne Partitionierung.

Leider ist die Sicht nicht aktualisierbar. Änderungsoperationen müssen also direkt auf den zugrunde liegenden Tabellen vorgenommen werden. Alternativ können Sie `INSTEAD OF`-Trigger erstellen, damit die Aufteilung der Tabelle für die Anwendung vollkommen transparent ist.

Die Transparenz hat allerdings auch einen Nachteil: Falls sich die Anwendung derart ändert, dass die vertikale Partitionierung nutzlos ist, so werden Sie dies nicht unmittelbar mitbekommen. In unserem Beispiel ist es ja vorstellbar, dass durch eine Änderung in der Anwendung nun in den meisten Fällen auch der Vorname des Kunden benötigt wird. Die Standardabfrage für die Daten eines Kunden würde jetzt so aussehen:

```
select Nr, NachName, LetzteBestellung, Vorname
  from Kunde
 where LetzteBestellung >= '20080101'
```

Eine solche Änderung in der Anwendung führt zu einem geänderten Ausführungsplan (Abbildung 7.4).

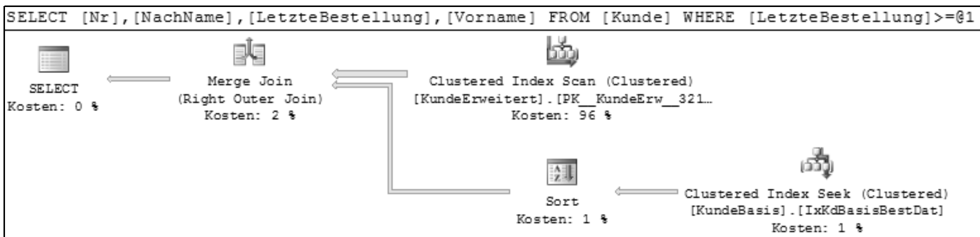


Abbildung 7.4: *Spalten aus beiden Tabellen werden abgefragt.*

Nun werden beide Tabellen und insgesamt 4.323 logische Lesevorgänge benötigt. Das sind immerhin etwa 16-mal so viele wie ohne vertikale Partitionierung. Das eigentliche Problem ist hier also, dass die Abfrage jetzt sogar teurer ist, als wenn die ursprüngliche Tabelle verwendet worden wäre. Damit erzielen wir durch die Partitionierung an dieser

Stelle keinen Performance-Gewinn mehr. Die Partitionierung bewirkt jetzt sogar das Gegenteil.

Bitte bedenken Sie, dass der Optimierer die Tabelle `KundeBasis` nie aus dem JOIN entfernen kann. Die folgende Abfrage benötigt daher beide Tabellen, auch wenn letztlich nur Spalten aus der Tabelle `KundeErweitert` zurückgegeben werden:

```
select Nr, VorName
   from Kunde
  where Nr = 2222
```

Das Fazit ist also recht einfach:

Sie müssen ständig überwachen, ob die konfigurierten vertikalen Partitionen noch den Anforderungen genügen, und die Partitionierung daher laufend den neuen Gegebenheiten anpassen. Da es sich hierbei stets um Änderungen des Schemas an Tabellen, Sichten und Triggern mit anschließender Datenmigration handelt, ist der dabei entstehende Aufwand nicht zu unterschätzen. Überlegen Sie also bitte genau, ob sich ein solcher Aufwand wirklich lohnt.

In Kapitel 11 werden Sie sehen, wie sich eine vertikale Partitionierung durch Indizes erzielen lässt.

7.3 Zusammenfassung

In diesem Kapitel haben Sie gesehen, wie Tabellendaten auf unterschiedliche Speicherorte verteilt werden können – eine Verfahrensweise, die als Partitionierung bezeichnet wird.

Generell sollten Sie sich merken, dass eine Partitionierung – egal ob horizontal oder vertikal – nur für sehr große Tabellen sinnvoll ist.

Mit »groß« ist hier der Speicherplatz gemeint, den die Tabellen- und Indexdaten belegen. Der benötigte Speicherplatz kann hierbei sowohl durch eine große Anzahl Spalten als auch durch eine große Anzahl Zeilen (und natürlich erst recht durch beides) derartig anwachsen, dass sich eine Partitionierung lohnt.

Falls eine Tabelle viele Spalten besitzt, die nicht alle gleich häufig verwendet werden, ist zu überlegen, ob eine vertikale Partitionierung in Frage kommt. Besitzt die Tabelle viele Zeilen, und ist es möglich, diese Zeilen in Gruppen einzuteilen, welche die Verwendungshäufigkeit widerspiegeln, sollten Sie über eine horizontale Partitionierung nachdenken.

In Kapitel 11 kommen wir noch einmal auf Partitionierung zurück. Dort werden Sie sehen, wie mit passenden Indizes ein ähnlicher Effekt wie bei einer Partitionierung erzielt werden kann.

8 Komprimierung von Daten

Die Minimierung von Ein-/Ausgabeoperationen ist bei der Optimierung einer der wichtigsten Punkte. Diese Aufgabe wird nicht gerade dadurch erleichtert, dass die Datenmengen, die in unseren Datenbanken gespeichert sind, beständig anwachsen, und die Entwicklung der Hardware hier nur schwer Schritt halten kann. Dies ist eine Tatsache, an der wir natürlich nichts ändern können. Was wir aber sehr wohl unternehmen können, ist die Ausnutzung aller existierenden Möglichkeiten zur Minimierung bzw. Beschleunigung von Ein-/Ausgabevorgängen.

Die vorangegangenen Kapitel haben gezeigt, wie eine solche Minimierung sehr wirkungsvoll mit Indizes erreicht werden kann. In diesem Kapitel wird eine weitere Möglichkeit vorgestellt: Die Komprimierung von Daten und Indizes.

8.1 Allgemeines zur Komprimierung

Eines gleich vorweg: Die Möglichkeit der Datenkomprimierung ist ein Feature, welches Ihnen nur in der Enterprise Edition zur Verfügung steht. Das Ziel der Komprimierung ist eine verbesserte Ausnutzung von Datenseiten, sodass letztlich in einer Seite mehr Zeilen untergebracht werden können. Hierbei können die folgenden Datenbankobjekte komprimiert werden:

- Benutzertabellen (Heaps oder gruppierte Indizes)
- nichtgruppierte Indizes
- indizierte Sichten

Sofern Sie Tabellen oder Indizes horizontal partitioniert haben, können Sie außerdem für jede Partition die Komprimierung extra konfigurieren.

Das Objekt *Datenbank* fehlt in der obigen Auflistung. Es ist nicht möglich, die Komprimierung für eine gesamte Datenbank, also alle enthaltenen Tabellen und Indizes, einzuschalten. Wenn Sie so etwas wünschen, können Sie es nur über entsprechende T-SQL-Skripte erledigen, indem Sie die Komprimierung für jedes Objekt einzeln konfigurieren. Das Management Studio unterstützt nur die Komprimierung einzelner Objekte.

Sobald die Komprimierung für ein Objekt eingeschaltet wurde, kümmert sich die Storage Engine von SQL Server automatisch um die Komprimierung und Dekomprimierung Ihrer Daten. Änderungen an Anwendungen, Abfragen oder gar am Datenbankdesign sind nicht erforderlich.

8.2 Vorteile einer Komprimierung

Komprimierte Daten benötigen weniger Speicherplatz auf der Festplatte. Der Nutzen liegt hierbei nicht so sehr in der Einsparung von Speicherplatz, denn dieser ist mittlerweile günstig genug. Der weitaus größere Vorteil ergibt sich aus der Einsparung von physikalischen E/A-Operationen. Aber es geht noch einen Schritt weiter. Wenn komprimierte Daten gelesen werden, so werden sie zunächst in den Datencache übertragen. Bei diesem Vorgang werden die Daten noch nicht dekomprimiert. Dadurch werden bei komprimierten Daten auch logische Lesevorgänge eingespart. Letztlich bewirkt dies auch, dass der Datencache insgesamt mehr Seiten aufnehmen kann, wodurch die Cache-Trefferquote ansteigt und auch eine bessere Ausnutzung des Hauptspeichers erreicht wird. Auch beim Schreiben erfolgt die Komprimierung vor dem Ablegen der Daten im Datencache.

Die SQL Server Storage Engine ist letztlich der Kanal für eine Komprimierung und Dekomprimierung. Wann immer die Storage Engine Daten empfängt, werden diese komprimiert. Sobald die Storage Engine Daten an andere SQL Server-Komponenten weitergibt, werden diese Daten zuerst dekomprimiert.

8.3 Komprimierungsarten

SQL Server kennt zwei Arten der Komprimierung:

- **Zeilenkomprimierung**. Bei der Zeilenkomprimierung einer Tabelle oder eines Index wird lediglich die Art und Weise der Speicherung der Datentypen verändert. Hierbei werden im Wesentlichen feste Formate, wie zum Beispiel `INTEGER` oder `CHAR` in ein variables Format überführt. Dadurch werden letztlich keine Leerbytes mehr gespeichert.
- **Seitenkomprimierung**. Eine Seitenkomprimierung tut noch etwas mehr. Zunächst einmal wird die Zeilenkomprimierung durchgeführt. Anschließend führen weitere Komprimierungsschritte für jede Datenseite zu einer noch stärkeren Komprimierung. Hierbei werden die für jede Komprimierung gängigen Verfahrensweisen angewendet, wie zum Beispiel die spezielle Behandlung von Wiederholungsgruppen oder die Speicherung von relativen, anstelle von absoluten Werten. In der Online-Dokumentation finden Sie hierzu eine sehr detaillierte Beschreibung, die allerdings auch mit dem Hinweis versehen wurde, dass die Verfahrensweise jederzeit geändert werden kann. Vergessen wir also einfach die technischen Details und merken uns nur, dass eine Seitenkomprimierung, ausgehend von der Zeilenkomprimierung, noch stärker komprimiert.

Der zu erreichende Grad der Komprimierung ist dabei sehr stark von den zu komprimierenden Daten abhängig. Wenn Sie für eine Tabelle, in der größtenteils Spalten vom Typ `SMALLINT` oder `VARDECIMAL` enthalten sind, die Komprimierung einschalten, wird der Effekt sicherlich nicht besonders zu beobachten sein. Im nächsten Abschnitt werden Sie das andere Extrem kennenlernen, bei dem eine Tabelle mit einer `NCHAR`-Spalte komprimiert wird, wobei die Komprimierung dort enorm lohnend ist.

Selbstverständlich gibt es eine Komprimierung nicht umsonst. Zunächst einmal müssen Sie hierfür die teure Enterprise Edition kaufen, aber das ist hier gar nicht gemeint. Datenkomprimierungen sind grundsätzlich CPU-lastige Vorgänge, benötigen also vor allem

Prozessorzeit. Der Vorteil, den Sie durch die Einsparung von E/A-Vorgängen gewinnen, kann also durchaus dadurch zunichte gemacht werden, dass Ihre Lese- und Schreiboperationen nun auf die Zuteilung von Prozessorzeit warten müssen. Falls Ihre Prozessoren also bereits beständig auf 80 Prozent der Last arbeiten, ist es sicherlich keine gute Idee, auch noch eine Datenkomprimierung zu konfigurieren.

Hierbei gilt grundsätzlich, dass eine Zeilenkomprimierung die Prozessoren weniger belastet als eine Seitenkomprimierung.

8.4 Beispiel: Auswirkung der Komprimierung auf die Abfrageleistung

Wir wollen nun an einem Beispiel untersuchen, wie sich eine Komprimierung auf die Abfrageleistung auswirken kann. Hierzu verwenden wir noch einmal die Tabelle `Kunde` aus dem vorherigen Kapitel. In diese Tabelle wurden mit dem Skript zu Beginn von Kapitel 7 insgesamt 30.000 Zeilen eingefügt.

Zum Konfigurieren der Komprimierung für die Tabelle `Kunden` wählen Sie SPEICHER • KOMPRIMIERUNG VERWALTEN... aus dem Kontextmenü der Tabelle. Es wird ein Assistent gestartet, der Sie in mehreren Schritten durch die Konfiguration führt. Auf der zweiten Seite haben Sie die Möglichkeit, die zeilen- oder seitenweise Komprimierung für die Tabelle einzustellen (Abbildung 8.1).

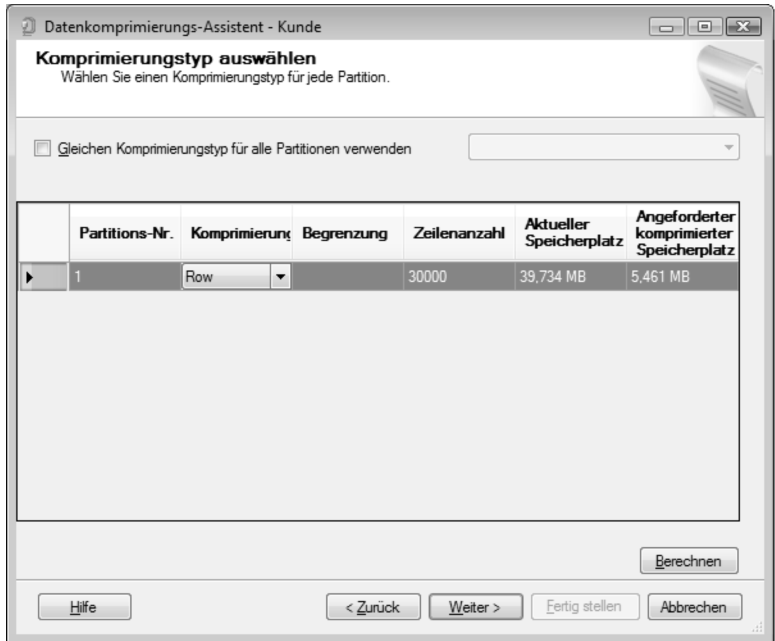


***Abbildung 8.1:** Konfigurieren einer Komprimierung*

Sehr nützlich ist in diesem Dialog der Button BERECHNEN, über den Sie eine Vorschau auf die Auswirkung der gewählten Komprimierung erhalten. In unserem Beispiel sind diese Auswirkungen recht drastisch. Immerhin werden für die zeilenweise komprimierte Tabelle (genau genommen ist es ein gruppierter Index) nur noch ca. 14 Prozent des ursprünglichen Speicherplatzes benötigt, also eine Einsparung von 86 Prozent! So ausgeprägt wird eine Komprimierung sich nicht in jedem Fall auswirken. Hier liegt es schlicht daran, dass unsere Spalte `Mehrspalten` sehr viele Leerzeichen enthält.

Da sich eine Komprimierung also lohnt, wollen wir sie entsprechend konfigurieren und führen daher die weiteren Schritte des Assistenten aus. Im nächsten Schritt haben Sie die Möglichkeit festzulegen, wann eine Komprimierung ausgeführt werden soll. Wählen Sie hier bitte die Option SOFORT AUSFÜHREN und schließen Sie den Assistenten.

Wir führen nun noch einmal die Abfrage aus dem vorherigen Kapitel aus, wobei vorher der Pufferspeicher geleert wird, damit auch die physikalischen Leseoperationen in den E/A-Statistiken zu sehen sind:

```
checkpoint
dbcc dropcleanbuffers
set statistics io on
set statistics time on
select Nr, LetzteBestellung, Nachname
  from Kunde
 where LetzteBestellung >= '20080101'
```

Tabelle 8.1 zeigt die Ergebnisse der Messungen der Abfrageleistung jeweils mit und ohne Komprimierung.

	Komprimierung		
	ohne	**Zeilen**	**Verbesserung um**
logische Lesevorgänge	272	38	86 %
physikalische Lesevorgänge	6	3	50 %
Read Aheads	269	35	87 %
Abfragekosten	0,20	0,03	85 %
CPU-Nutzung	0	0	0 %

Tabelle 8.1: *Auswirkungen einer Komprimierung auf die Abfrageleistung*

Es wird deutlich, dass der Unterschied in den Messergebnissen fast direkt aus der Einsparung von Speicherplatz durch die Komprimierung abgeleitet werden kann. Dies ist darauf zurückzuführen, dass unsere Abfragekosten hauptsächlich durch Datenleseoperationen bestimmt werden. Und genau diese Operationen profitieren von der Komprimierung. Interessant ist, dass in beiden Fällen keine messbare CPU-Last erzeugt wurde.

8.5 Komprimierten Speicherplatz berechnen

In der Online-Dokumentation finden Sie Beispiele für eine Abschätzung des bei einer Komprimierung eingesparten Speicherplatzes für die unterschiedlichen Datentypen. Es lohnt sich, dies einmal zu lesen, damit Sie ein Gefühl dafür bekommen, wann eine Komprimierung sich lohnt.

In Tabelle 8.1 haben Sie gesehen, dass die Auswirkungen der Komprimierung über den Button BERECHNEN ermittelt werden können. Falls Sie die Tabellen in Ihren Datenbanken dahingehend untersuchen möchten, ob eine Komprimierung sinnvoll ist, ist es natürlich sehr mühselig, wenn Sie für jede Tabelle den Assistenten starten müssen, um die Berechnung auszuführen.

Glücklicherweise gibt es hierfür die gespeicherte Prozedur `sp_estimate_data_compression_savings`, mit der sie diese Vorhersage auch über T-SQL erstellen können. Die Prozedur erwartet insgesamt fünf Parameter, wobei die ersten drei das zu komprimierende Objekt identifizieren. Der vierte Parameter wird nur benötigt, falls Sie die Komprimierung nur für eine bestimmte Partition planen. Der letzte Parameter gibt schließlich die Art der Komprimierung an. *ROW* steht hierbei für Zeilenkomprimierung und *PAGE* für Seitenkomprimierung.

Diese Prozedur wird im folgenden SQL-Skript verwendet, um für alle Tabellen der Beispieldatenbank `AdventureWorks2008` die erwarteten Einsparungen für eine Zeilenkomprimierung zu berechnen:

```
set nocount on
-- Erzeuge eine temporäre Tabelle, die das Ergebnis aufnimmt
create table #komp
 (
  object_name sysname
 ,schema_name sysname
 ,index_id int
 ,partition_number int
 ,[size_with_current_compression_setting (KB)] bigint
 ,[size_with_requested_compression_setting (KB)] bigint
 ,[sample_size_with_current_compression_setting (KB)] bigint
 ,[sample_size_with_requested_compression_setting (KB)] bigint
 )
go

-- Für alle Tabellen in allen Schemata die Auswirkungen
-- der Komprimierung berechnen
use AdventureWorks2008;
declare @cmd nvarchar(max)
set @cmd = ''
select @cmd = @cmd
    +';insert #komp exec sp_estimate_data_compression_savings '''
    + schema_name(schema_id)+''','''
    + name + ''',null, null, ''row'''
  from sys.tables
```

```
exec (@cmd)

-- Ergebnis ausgeben
select object_name as TabellenName
      ,schema_name as SchemaName
      ,cast(case
              when [size_with_current_compression_setting (KB)] = 0 then 0
              else 100.0*(1.0-1.0
                     *[size_with_requested_compression_setting (KB)]
                     /[size_with_current_compression_setting (KB)])
            end as decimal(6,2)) as [Einsparung (%)]
  from #komp
 where index_id < 2
  order by [Einsparung (%)] desc

-- Temporäre Tabelle wieder löschen
drop table #komp
```

Im Skript wird die Prozedur für jede Tabelle der Datenbank aufgerufen. Das Ergebnis des Prozeduraufrufs wird in eine temporäre Tabelle eingetragen. Abschließend wird der Inhalt dieser Tabelle ausgegeben, wobei eine Sortierung nach dem Grad der Komprimierung erfolgt.

Abbildung 8.2 zeigt die ersten Zeilen aus dem Ergebnis meiner Datenbank `AdventureWorks2008`.

	TabellenName	SchemaName	Einsparung (%)
1	EmployeeDepartmentHistory	HumanResources	50.00
2	SalesPersonQuotaHistory	Sales	50.00
3	TransactionHistoryArchive	Production	36.90
4	PurchaseOrderDetail	Purchasing	36.36
5	PurchaseOrderHeader	Purchasing	34.09
6	TransactionHistory	Production	33.75

Abbildung 8.2: *Auswirkungen der Komprimierung auf ausgewählte Tabellen*

Im Skript werden ausschließlich Tabellen untersucht. Sie können das Skript aber leicht so modifizieren, dass auch Indizes und Partitionen geprüft werden.

Seien Sie bitte vorsichtig, wenn Sie das Skript in Produktivumgebungen verwenden. Für die Vorhersage werden Stichproben der Daten aus den Tabellen geholt, was einiges an E/A-Last erzeugt.

Bei der Interpretation des Ergebnisses ist ein weiterer Punkt zu beachten: Die Vorhersage der Einsparung muss nämlich nicht in jedem Fall zutreffen. Letztlich sollen durch eine Komprimierung mehr Zeilendaten in einer Datenseite untergebracht werden. Bei Indizes spielt hier auch der Füllfaktor eine Rolle, über den festgelegt werden kann, dass Indexseiten nicht komplett gefüllt werden sollen. Ein niedriger Füllfaktor ist für die Komprimierung also eher hinderlich.

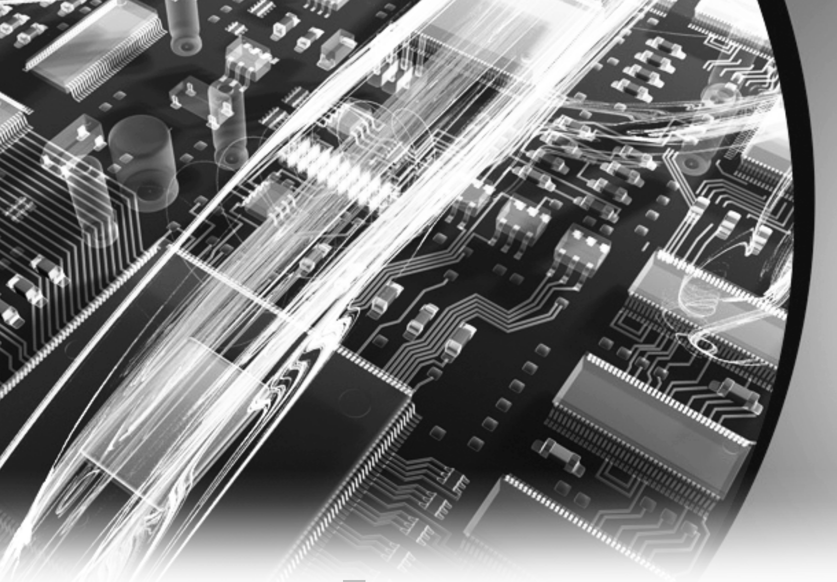
Auch für die Komprimierung von Tabellendaten sollten Sie die Vorhersage noch einer Prüfung unterziehen. Stellen Sie sich vor, die Prognose verspricht eine Einsparung von 20 Prozent. Diese Angabe bezieht sich auf die Einsparung, die für die Daten einer Zeile erzielt werden kann. Falls Ihre Zeile 6.000 Byte groß ist, so benötigt sie eine Datenseite. Nach einer Komprimierung um 20 Prozent ist die Zeile 4.800 Byte lang. Damit passt in eine Datenseite nach wie vor nur noch eine einzige Zeile; die Komprimierung hat sich also letztlich nicht gelohnt.

Auch die Fragmentierung spielt hier eine Rolle. Fragmentierte Daten- oder Indexseiten sind ja nicht komplett gefüllt. Die Defragmentierung eines gruppierten Index wird dafür sorgen, dass die Blattseiten mit den eigentlichen Daten wieder besser ausgenutzt werden, was in gewisser Weise ebenfalls eine Komprimierung dargestellt. Die Prozedur `sp_estimate_data_compression_savings` erstellt ihre Prognose für die Einsparung stets für komprimierte Daten, die nicht fragmentiert sind. Falls Ihr gruppierter Index also stark fragmentiert ist, kann es passieren, dass Sie den Grad der Einsparung überschätzen, da die Originaldaten fragmentiert sind, die Schätzwerte aber für nicht fragmentierte Daten berechnet werden. Es ist also empfehlenswert, vor der Abschätzung der Komprimierung für einen Index (gruppiert oder nichtgruppiert), eine Defragmentierung durchzuführen, so wie in Kapitel 6 beschrieben.

8.6 Zusammenfassung

Datenkomprimierung kann helfen, kostenintensive E/A-Operationen einzusparen. In diesem Kapitel haben Sie gesehen, wie Sie Ihre Tabellen- und Indexdaten komprimieren können und wie Sie feststellen, ob eine Komprimierung sich lohnt.

Um die Auswirkungen der Komprimierung direkt zu messen, ist es sicherlich am sinnvollsten, wenn Sie so vorgehen, wie in Abschnitt 8.4 gezeigt. Erstellen Sie entsprechende Testskripte zum Messen der E/A-Operationen und führen Sie die Messungen sowohl mit als auch ohne Komprimierung aus.



Teil 3

Optimierung

9 Analysieren und Optimieren von Abfragen 181
10 Auffinden problematischer Abfragen 265
11 Optimierung des physischen Datenbankentwurfs 289
12 Kontrollieren von Ressourcen 311
13 Testen und Optimieren des E/A-Systems 319

9 Analysieren und Optimieren von Abfragen

In den vorangegangenen Kapiteln haben Sie eine Reihe Fakten über die Arbeitsweise von SQL Server und speziell die Arbeitsweise des Optimierers und der SQL Server-Abfrage-Engine erfahren. Das vorliegende Kapitel befasst sich nun noch einmal tiefergehend mit der Optimierung von T-SQL-Abfragen.

Ich möchte Sie noch einmal daran erinnern, dass die Abfrageoptimierung den zentralen Schwerpunkt dieses Buches bildet. Aus diesem Grund sind die beiden nun folgenden Kapitel aus meiner Sicht die wichtigsten Abschnitte des Buches. Natürlich ist SQL Server insgesamt ein komplexes System, in dem viele Komponenten leistungsfähig und »wohlkonfiguriert« sein müssen. Es ist aber gerade die Optimierung von Abfragen, bei der man nach meiner Erfahrung sozusagen das meiste herausholen kann. Hierbei werden wir uns gar nicht so sehr auf die Sprache T-SQL selbst konzentrieren. Zu diesem Thema gibt es bereits eine Reihe exzellenter Bücher (z.B. [1] und [4]), mit denen das Ihnen hier vorliegende Werk nicht konkurrieren möchte.

Den Schwerpunkt bilden auch an dieser Stelle wieder die Abfrage-Engine von SQL Server sowie der Optimierer. Sie werden in diesem Kapitel gleichermaßen interessante und wichtige Details zur Abfrageausführung erfahren. Das Ziel ist ein möglichst tiefes Verständnis dieser Materie. Es ist sicherlich leicht einzusehen, dass Sie für eine Abfrageoptimierung möglichst genau wissen sollten, nach welchen Kriterien der Optimierer einen Plan erstellt, oder welche Möglichkeiten der Planerstellung überhaupt existieren, und in welcher Weise Sie diese Möglichkeiten beeinflussen können. Sie werden anhand von aussagekräftigen und dennoch leicht nachvollziehbaren Beispielen wesentliche Konzepte der Abfrageoptimierung kennenlernen.

9.1 Ausführungspläne und der Plancache

Bereits aus den Kapiteln 3 und 4 wissen Sie, dass der Optimierer für jede Ausführung einer Abfrage einen Plan benötigt. Der Optimierer erstellt diesen Plan nach Kriterien, über die Sie ebenfalls bereits in den Kapiteln 3 und 4 eine Übersicht erhalten haben. In diesem Abschnitt werden Sie nun etwas mehr darüber erfahren, welche Kriterien für die Erstellung von Ausführungsplänen verwendet werden.

Ein wesentliches Ziel der SQL Server-Abfrage-Engine ist die Wiederverwendung bereits erstellter Ausführungspläne. Zu diesem Zweck werden erstellte Pläne nach Möglichkeit im Plancache (der oft auch als Prozedurcache bezeichnet wird) gespeichert. Der Optimierer prüft stets, ob in diesem Plancache bereits ein geeigneter Plan vorhanden ist, bevor er mit der Erstellung eines neuen Abfrageplans beginnt. Dieses Konzept wurde entwickelt, um Systemressourcen zu sparen. Das Kompilieren einer Abfrage ist ein CPU-intensiver

Prozess. Deshalb ist es in den meisten Fällen günstiger, einen gespeicherten Plan zu verwenden, als einen neuen zu erstellen. Ganz problemlos ist diese Verfahrensweise allerdings nicht. Sie sollten daher die Faktoren kennen, die Einfluss auf die Erstellung, Speicherung und Wiederverwendung von Abfrageplänen haben. Hierfür werden Ihnen die folgenden drei Abschnitte die nötigen Informationen liefern, indem Antworten auf die folgenden Fragen gegeben werden:

- Welche Strategien kennt der Optimierer zur Speicherung von Abfrageplänen?
- Wann ist ein gespeicherter Plan für eine Wiederverwendung geeignet?
- Welche Vor- und Nachteile bringt das Puffern (Cachen) von Abfrageplänen?

Ein erstellter Abfrageplan wird nach Möglichkeit im Plancache gespeichert. Darüber hinaus kann der Plancache auch noch andere Typen von Objekten aufnehmen, er ist also nicht auf die Speicherung »ausführungsbereiter« Pläne beschränkt. Stellen Sie sich zum Beispiel die Definition einer Sicht vor. Für diese Sicht kann kein fertiger Ausführungsplan gespeichert werden, weil erst bei der Verwendung der Sicht in einer Abfrage bekannt ist, nach welchen Parametern gefiltert wird, oder auch welche weiteren Tabellen über Joins hinzugefügt werden. Allerdings können bereits bei der Definition der Sicht eine Syntaxprüfung sowie eine Namensauflösung erfolgen. Daher kann der durch den Algebrizer (siehe Kapitel 3) erzeugte Syntaxbaum sehr wohl gespeichert werden, und das wird auch getan. Bei der Verwendung der Sicht in einer Abfrage muss dann ein entsprechender Ausführungsplan nicht von Grund auf neu erstellt werden, sondern kann auf den gespeicherten Algebrizer-Baum zurückgreifen.

Ebenso werden im Plancache die sogenannten Ausführungskontexte gespeichert. Ein Ausführungskontext wird letztlich aus einem gespeicherten Plan abgeleitet und enthält zusätzliche Informationen, beispielsweise zu Abfrageparametern oder zu Einstellungen der Verbindung. Zu einem existierenden Plan können mehrere Ausführungskontexte existieren, die ebenfalls nach Möglichkeit wiederverwendet werden.

Wir werden uns in diesem Kapitel vorwiegend mit gespeicherten Ausführungsplänen auseinandersetzen. Der Plancache kann die folgenden Typen von Ausführungsplänen aufnehmen, die uns im weiteren Verlauf besonders interessieren:

- **Ad-Hoc-Abfragen**. Ad-Hoc-Abfragen werden mit ihrem Abfragetext im Plancache gespeichert. Dies gilt auch für die Ausführung von dynamischem SQL über die Funktion `EXEC()`. So wird zum Beispiel auch ein Plan für die folgende `SELECT`-Anweisung innerhalb der `EXEC`-Funktion gespeichert:

  ```
  declare @id varchar(8)
  set @id='0'
  exec('select * from msdb.dbo.backupfile
         where backup_set_id=' + @id)
  ```

- **Gespeicherte Prozeduren**. Für gespeicherte Prozeduren wird bei der ersten Ausführung ein Plan erstellt und gespeichert. Sofern die Prozedur Parameter verarbeitet, wird der erstellte Plan hinsichtlich der bei diesem ersten Aufruf übergebenen Parameter optimiert. Dies kann unter Umständen problematisch sein, wie Sie etwas weiter unten noch sehen werden.

- **Parametrisierte Abfragen**. Für die Wiederverwendung von Abfragen versucht SQL Server, nicht jeden Abfragetext exakt zu speichern. Dies würde die Kapazität des Plancache selbst auf leistungsfähigen Systemen sehr schnell überfordern. SQL Server wird stets versuchen, für eine gestellte Abfrage nur eine Vorlage, also gewissermaßen ein Muster für Abfragen ähnlicher Art, zu speichern und diese Vorlage wiederzuverwenden. Hierbei wird so vorgegangen, dass der Abfragetext nach Parametern (Literalen) durchsucht wird, für die stattdessen Platzhalter gespeichert werden. Diese Platzhalter können dann bei jeder Verwendung der gespeicherten Abfrage einfach gegen die konkreten Werte ausgetauscht werden. Diese Parametrisierung kann automatisch erfolgen oder auf unterschiedliche Weise erzwungen werden. Da dieses Konzept sehr wesentlich ist, beschäftigen wir uns damit in einem eigenen Abschnitt (9.3).

Nicht alle Abfragen werden übrigens im Plancache gespeichert. Damit eine Abfrage in den Cache aufgenommen wird, muss sie einige Kriterien erfüllen, was jedoch bei den meisten Abfragen der Fall ist. So werden zum Beispiel keine Abfragen gespeichert, die Literale mit einer Größe von mehr als 8 kByte enthalten. Das Speichern eines Plans im Plancache kann auch durch SQL-Code unterbunden werden, was in einigen Fällen eine sinnvolle Option ist. Hierauf werden wir etwas weiter unten noch detaillierter eingehen.

Aus den vorangegangenen Kapiteln wissen Sie bereits, dass Sie zur Abfrage der im Plancache befindlichen Objekte die dynamische Verwaltungssicht `sys.dm_exec_cached_plans` verwenden können. Allerdings gibt diese Sicht alleine noch nicht alle interessanten Informationen zurück. Sie können die Sicht mit `sys.dm_exec_query_stats` verknüpfen, um auch statistische Informationen zu bekommen. Diese statistischen Informationen geben zum Beispiel Auskunft über die Ausführungshäufigkeit einer Abfrage oder über die Gesamtdauer aller Ausführungen. Über die Sicht `sys.dm_exec_plan_attributes` fügen Sie zusätzliche Parameter für die Ausführungsumgebung der Abfrage hinzu. Den Text der Abfrage erhalten Sie über die Funktion `sys.dm_exec_sql_text`. Schließlich können Sie auch noch den Ausführungsplan selber abfragen, den Sie durch die Funktion `sys.dm_exec_query_plan` erhalten. Sie erhalten schließlich die folgende Abfrage:

```
select left(p.cacheobjtype + '(' + p.objtype + ')', 35) as cacheobjtype
      ,p.usecounts
      ,cast(p.size_in_bytes/1024.0 as decimal(12,2)) as size_in_kb
      ,cast(qs.total_worker_time*0.001 as decimal(12,2)) as tot_cpu_ms
      ,cast(qs.total_elapsed_time*0.001 as decimal(12,2)) as tot_duration_ms
      ,qs.total_physical_reads
      ,qs.total_logical_writes
      ,qs.total_logical_reads
      ,left(case
             when pa.value=32767 then 'ResourceDb'
             else db_name(cast(pa.value as int))
           end, 40) as db_name
      ,qt.objectid
      ,case
         when qt.objectid is null then null
         else replace(replace(qt.text, char(13), ' '), char(10), ' ')
       end as proc_name
      ,pln.query_plan as query_plan
```

```
      ,replace(replace(substring(qt.text, qs.statement_start_offset/2+1,
               case
                 when qs.statement_end_offset = -1 then
                      len(convert(nvarchar(max), qt.text))
                 else qs.statement_end_offset/2
                     - qs.statement_start_offset/2+1
               end), char(13), ' '), char(10), ' ') as stmt_text
  from sys.dm_exec_cached_plans as p
       outer apply sys.dm_exec_plan_attributes(p.plan_handle) as pa
       outer apply sys.dm_exec_query_plan(p.plan_handle) as pln
       inner join sys.dm_exec_query_stats as qs
               on qs.plan_handle = p.plan_handle
       outer apply sys.dm_exec_sql_text(qs.sql_handle) as qt
 where pa.attribute = 'dbid'
 order by tot_cpu_ms desc
```

Die Abfrage gibt alle im Plancache enthaltenen Pläne, absteigend nach der Ausführungsdauer, zurück. Vielleicht fragen Sie sich, warum in der obigen Abfrage der Text der SQL-Anweisung auf so eine recht komplizierte Art und Weise ermittelt wird: Die Sicht `sys.dm_exec_query_stats` enthält eine Statistikzeile je SQL-Anweisung in einem T-SQL-Stapel, die Funktion `sys.dm_exec_sql_text` gibt aber stets den gesamten Text des T-SQL-Stapels zurück. In `sys.dm_exec_query_stats` finden Sie die Positionsangaben der entsprechenden Zeile, bezogen auf den gesamten Text in `sys.dm_exec_sql_text`. Da diese Angaben Unicode-bezogen sind, ist die Rechnung dann etwas umständlich.

9.1.1 Kompilierung und Re-Kompilierung von Ausführungsplänen

Bei der Abfrageausführung wird zunächst versucht, einen passenden Plan im Plancache zu finden und zu verwenden. Wird ein solcher Plan gefunden, ist die Planerstellung abgeschlossen und die Abfrage wird unter Verwendung dieses Plans gestartet. Dies ist letztlich ja das Ziel der Speicherung von Plänen im Plancache.

Falls kein passender Plan gefunden wird, muss zunächst ein neuer Plan erstellt werden. Diesen Vorgang bezeichnet man als *Kompilierung*. Es ist auch möglich, dass ein passender Plan existiert, der Optimierer sich aber entscheidet, diesen Plan nicht zu verwenden, weil der Plan mittlerweile ungültig ist. In diesem Fall wir der vorhandene Plan durch einen neuen Plan ersetzt. Dieses Verfahren bezeichnet man als *Re-Kompilierung*.

Unter welchen Voraussetzungen ein Plan ungültig wird, erfahren Sie in Abschnitt 9.1.4.

9.1.2 Entfernen von Plänen aus dem Plancache

Der Plancache benötigt letztlich auch Hauptspeicher, den er sich mit anderen Systemkomponenten teilen muss. Zu diesen Komponenten gehören einerseits die SQL Server eigenen Komponenten, hauptsächlich der Datencache. Auf der anderen Seite steht natürlich das Betriebssystem als externe Komponente, welches ebenfalls Speicher benötigt. Aus diesem Grund kann der Plancache in der Regel nicht beliebig groß werden. Das SQL Server-

Betriebssystem (SQLOS) sorgt dafür, dass selten verwendete Pläne automatisch aus dem Plancache entfernt werden, sobald sich ein externer oder interner Mangel an Hauptspeicher bemerkbar macht. Dazu läuft im Hintergrund ein Prozess, der für alle im Plancache vorhandenen Pläne deren Verwendung zählt. Die am wenigsten verwendeten Pläne werden durch diesen Prozess aus dem Cache entfernt, sodass das SQLOS Hauptspeicher zurückgibt. Denken Sie bitte daran, wenn Sie die im Plancache vorhandenen Pläne abfragen. In vielen Fällen werden Sie nicht auf alle seit dem letzten Start von SQL Server erstellten Pläne zurückgreifen können.

Es gibt noch eine zweite Möglichkeit, gespeicherte Abfragepläne aus dem Plancache zu entfernen, die Sie in den vorangegangenen Kapiteln bereits kennengelernt haben. Die Anweisung

```
dbcc freeproccache
```

löscht den Plancache des gesamten Servers.

Eine dritte Möglichkeit ist undokumentiert. Die Anweisung

```
dbcc flushprocindb(<db_id>)
```

löscht den Plancache für eine bestimmte Datenbank, deren db_id Sie als Parameter übergeben müssen. Dies betrifft die Pläne für alle Abfragen, die mit der entsprechenden Datenbank im Kontext (use <db>) in den Plancache aufgenommen wurden.

Darüber hinaus gibt es eine Reihe von SQL-Anweisungen, die ebenfalls den Plancache für eine bestimmte Datenbank komplett leeren. Hierzu zählen zum Beispiel diverse ALTER DATABASE-Anweisungen zum Ändern des Datenbankstatus, wie etwa das Onlinesetzen oder Offlinesetzen einer Datenbank oder auch die Anweisung DROP DATABASE.

9.1.3 Parametrisierte Abfragen

Der Begriff »parametrisierte Abfragen« ist nun bereits einige Male gefallen. Bevor wir dieses Konzept näher untersuchen, sind zunächst noch einige einleitende Bemerkungen nötig.

Betrachten Sie bitte einmal die folgende Abfrage:

```
select FirstName, LastName
  from AdventureWorks2008.Person.Person
 where BusinessEntityID = 10312
```

Schauen Sie sich den grafischen Ausführungsplan für diese Abfrage an, (siehe Abbildung 9.1) dann fällt auf, dass der im Plan dargestellte Text der Abfrage nicht den Wert des in der WHERE-Klausel verwendeten Parameters anzeigt, sondern nur einen Platzhalter der Form @1.

SELECT [FirstName],[LastName] FROM [AdventureWorks2008].[Person].[Person] WHERE [BusinessEntityID]=@1

SELECT
Kosten: 0 %

Clustered Index Seek (Clustered)
[Person].[PK_Person_BusinessEntityI…
Kosten: 100 %

Abbildung 9.1: *Ein parametrisierter Abfrageplan*

Die Abfrage ist automatisch parametrisiert worden. Diese Parametrisierung hat zur Konsequenz, dass eine identische Abfrage, bei der lediglich die Parameter variieren, den gespeicherten parametrisierten Plan verwenden kann. Für die folgende Abfrage

```
select FirstName, LastName
  from AdventureWorks2008.Person.Person
 where BusinessEntityID = 1770
```

wird also kein neuer Plan erstellt, sondern der bereits existierende (parametrisierte) Plan verwendet.

Wenn Sie sich den Inhalt des Plancache über die oben vorgestellte Abfrage ausgeben lassen, erkennen Sie die Parametrisierung im Abfragetext, der in diesem Fall so aussieht:

```
SELECT [FirstName],[LastName]
  FROM [AdventureWorks2008].[Person].[Person]
 WHERE [BusinessEntityID]=@1
```

Als Typ des gespeicherten Plans erhalten Sie den Wert *Compiled Plan(Prepared)*. Auch aus diesem Typ können Sie darauf schließen, dass die Abfrage parametrisiert – oder eben vorbereitet – ist.

Bitte beachten Sie, dass die Parametrisierung automatisch erfolgt ist. Wir mussten den Optimierer nicht erst durch irgendwelche Hinweise oder Tricks dazu veranlassen. Es gibt auch eine erzwungene Parametrisierung, auf die wir etwas weiter unten zu sprechen kommen.

Es sind übrigens sehr viele Restriktionen, die eine automatische Parametrisierung verhindern. Zu diesen Kriterien (die ich hier nicht alle angebe) zählen zum Beispiel die folgenden:

- Eine Verwendung der `DISTINCT`-Klausel
- Eine Verwendung der `TOP`-Klausel
- Die Verwendung von mehr als einer Tabelle in der Abfrage (!)
- Die Verwendung von `UNION`
- Die Verwendung der `IN`-Klausel

Wundern Sie sich also bitte nicht, wenn Sie eine parametrisierte Abfrage erwarten, aber nicht erhalten. Sehr wahrscheinlich erfüllt Ihre Abfrage eines der Ausschlusskriterien, die eine automatische Parametrisierung verhindern.

9.1.4 Wiederverwendung von Abfrageplänen

Eine hohe Trefferquote im Plancache – und damit eine hohe Wiederverwendungsquote gespeicherter Pläne – ist letztlich das Ziel des Pufferns von Abfrageplänen. Damit ein gespeicherter Plan wiederverwendet werden kann, müssen einige Voraussetzungen erfüllt sein. Zunächst einmal ist eine Wiederverwendung nur dann möglich, wenn der Plan überhaupt jemals in den Cache aufgenommen wurde. Das Speichern eines Plans im

Cache ist daher schon einmal eine erste Voraussetzung für eine Wiederverwendung. Generell muss ein Plan im Cache existieren, falls er wiederverwendet werden soll. Da es auch möglich ist, dass Pläne aus dem Cache entfernt werden, kann natürlich der Fall eintreten, dass ein ursprünglich gespeicherter Plan nicht wiederverwendet werden kann, weil er in der Zwischenzeit aus dem Plancache entfernt wurde.

Diese Möglichkeiten wollen wir in unseren folgenden Betrachtungen aber nicht weiter betrachten. In diesem Abschnitt soll geklärt werden, welche Voraussetzungen erfüllt sein müssen, damit ein gespeicherter Plan wiederverwendet werden kann, beziehungsweise wann ein gespeicherter Plan ungültig ist, sodass er nicht wiederverwendet werden kann.

Ein gespeicherter Plan kann aus zwei Gründen ungültig werden: Zum einen muss der Plan jederzeit ein richtiges Abfrageergebnis sicherstellen, das heißt, er muss korrekt sein. Ein weiterer Punkt ist die Effizienz des gespeicherten Planes. Ein gespeicherter Plan kann durch Datenänderungen ineffizient und damit ebenfalls ungültig werden.

Die Gültigkeit eines existierenden Plans hängt im Wesentlichen von drei Faktoren ab:

- **Metadaten**. Sobald sich beispielsweise die Struktur der beteiligten Tabellen ändert oder Indizes hinzugefügt bzw. gelöscht werden, ist ein Ausführungsplan, der auf den alten Metadaten basiert, ungültig.
- **Ausführungsumgebung**. Für jede Verbindung existiert eine Reihe von Optionen, die durch das SET-Kommando eingestellt werden. Viele dieser Optionen sind integraler Bestandteil von Abfrageplänen, welche die Optionen sozusagen als Randbedingungen enthalten. Ein existierender Plan ist ungültig, wenn dieselbe Abfrage mit anderen Einstellungen ausgeführt wird. Die folgenden SET-Optionen, bewirken die Invalidierung eines Plans:

  ```
  ANSI_NULL_DFLT_OFF
  ANSI_NULL_DFLT_ON
  ANSI_NULLS
  ANSI_PADDING
  ANSI_WARNINGS
  ARITHABORT
  CONCAT_NULL_YIELDS_NULL
  DATEFIRST
  DATEFORMAT
  FORCEPLAN
  LANGUAGE
  NO_BROWSETABLE
  NUMERIC_ROUNDABORT
  QUOTED_IDENTIFIER
  ```

- **Tabellen- bzw. Indexdaten**. Diese Abhängigkeit betrifft nicht die Korrektheit des Plans, sondern seine Effizienz. Wenn Tabellendaten geändert werden, ist es durchaus möglich, dass ein existierender Plan nicht mehr optimal ist, weil er zum Beispiel Indizes verwendet, deren Einsatz durch die Datenänderungen sich nicht mehr lohnt. Der Plan führt jedoch nach wie vor zum korrekten Ergebnis, nur eben nicht auf dem optimalen Weg. Solche Fälle sind schwer zu entdecken. SQL Server bietet jedoch Mechanismen an, die helfen, eine derartige Situation zu vermeiden. Hierauf gehen wir in Abschnitt 9.2 ausführlicher ein.

Die Funktion sys.dm_exec_plan_attributes liefert Informationen zu den Parametern eines Plans zurück. Die Spalte is_cache_key gibt hierbei an, ob der entsprechende Parameter bei einer Änderung eine Invalidierung des Abfrageplans bewirkt. Für alle Parameter mit dem Wert 1 in dieser Spalte besteht diese Abhängigkeit.

Wir wollen hierzu ein Beispiel untersuchen. Betrachten Sie bitte das folgende Skript:

```
set nocount on
declare @i int,@x int
set @i = 1
while @i<=1000
 begin
  if (@i%2=0)
    set ansi_nulls on
  else
    set ansi_nulls off

  set @x=(select top 1 checksum(*)
            from AdventureWorks2008.Sales.SalesOrderDetail
           where SalesOrderDetailId = 0)
  set @i=@i+1
end
```

In einer Schleife wird 1.000 Mal dieselbe SELECT-Anweisung ausgeführt. Der Wert der Option ANSI_NULLS wird dabei bei jeder Ausführung geändert. Dadurch wird der gespeicherte Plan ungültig, und es ist in jedem Schleifendurchlauf eine Re-Kompilierung erforderlich.

Wenn Sie im Windows-Systemmonitor die Prozessorlast und die SQL-Re-Kompilierungen pro Sekunde beobachten, können Sie diese Aussage nachvollziehen (Abbildung 9.2).

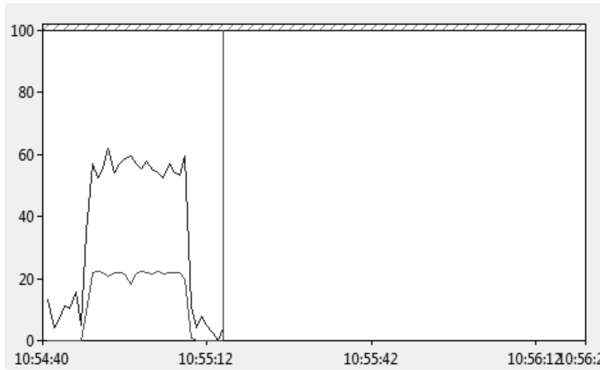


Abbildung 9.2: *Prozessorzeit (obere Kurve) und Re-Kompilierungen (untere Kurve)*

Im Plancache ist für die in der Schleife ausgeführte SELECT-Anweisung tatsächlich nur ein einziger Plan vorhanden. Für diesen Plan enthält die Spalte execution_count der Sicht sys.dm_exec_query_stats den Wert 1. Der Plan wurde also nach seiner Erstellung bzw. Neuerstellung nur ein Mal ausgeführt. Entfernen Sie die SET ANSI_NULLS-Anweisungen aus dem obigen Skript, so erhalten Sie für execution_count den Wert 1.000.

Damit ein nicht parametrisierter Plan wiederverwendet werden kann, muss außerdem der Abfragetext exakt übereinstimmen. Hierbei sind also beispielsweise auch die Groß-/Kleinschreibung sowie Leerzeichen relevant.

Betrachten Sie hierzu bitte das folgende Beispiel:

```
dbcc freeproccache with no_infomsgs
go
select checksum_agg(checksum(*)) from sys.all_columns
go
select checksum_agg(checksum(*))   from sys.all_columns
go
Select checksum_agg(checksum(*)) from sys.all_columns
go
SELECT checksum_agg(checksum(*)) from sys.all_columns
go
```

Nach der Ausführung der vier Abfragen werden Sie für jede Abfrage einen separaten Ausführungsplan im Plancache finden, da die obigen vier Abfragen sich allesamt im Abfragetext unterscheiden.

Parametrisierte Pläne verhalten sich in diesem Punkt anders. Wenn eine Abfrage parametrisiert wird, dann wird der Abfragetext in diesem Zusammenhang auch formatiert oder – wenn Sie so wollen – standardisiert. Da in der Folge der formatierte Abfragetext für einen Vergleich verwendet wird, findet kein exakter Vergleich des ursprünglichen Abfragetextes statt.

Mittlerweile ist Ihnen sicherlich klar geworden, dass eine hohe Trefferquote im Plancache in den meisten Fällen ein erstrebenswertes Ziel ist. Sie können die Plancache-Trefferquote beobachten, indem Sie zum Beispiel den in Kapitel 4 vorgestellten Indikator *SQL Server:Plan Cache → Cachetrefferqoute* des Systemmonitors verwenden. Möglich ist auch eine Abfrage über die dynamische Verwaltungssicht `sys.dm_os_performance_counters`:

```
select * from sys.dm_os_performance_counters
 where counter_name like 'Cache Hit%'
   and object_name like '%plan cache%'
```

Die Abfrage liefert allerdings nicht den Verlauf über die Zeit, sondern nur den kumulierten Wert seit dem letzten Start von SQL Server.

9.2 Die Rolle von Statistiken

Bei der Erstellung eines Abfrageplans wählt der Optimierer die physischen Operatoren unter Verwendung der Tabellen- und Indexdaten aus. Hierbei werden Kardinalitätsschätzungen vorgenommen, die zum Beispiel ausschlaggebend dafür sind, ob eine Suchoperation über einen vorhandenen Index ausgeführt wird oder etwa durch einen Table Scan.

Griffe der Optimierer hierbei jeweils auf die tatsächlich vorhandenen Daten der beteiligten Tabellen und Indizes zurück, so würde dies die Erstellung eines Plans zu einer kostspieligen und langwierigen Angelegenheit machen.

Aus diesem Grund wird eine andere Verfahrensweise angewendet: Der Optimierer verwendet für die Erstellung eines Plans Stichproben über die Datenverteilung in Tabellen und Indizes: die Statistiken. Diese Stichproben sind naturgemäß erheblich kleiner als beispielsweise die Daten einer gesamten Tabelle, und daher ermöglicht ihre Verwendung eine schnelle Erstellung eines Abfrageplans.

Allerdings wirft die Verwendung von Statistiken auch einige Probleme auf. Diese Probleme resultieren zum einen aus der bekannten Tatsache, dass die Verwaltung redundanter Daten gewisse Anomalien hervorrufen kann. Dies ist immer dann der Fall, wenn die ursprünglichen Daten nicht mehr synchron mit den redundanten Daten sind. Statistiken enthalten ja letztlich redundante Daten, da sie direkt aus Tabellen- und Indexdaten abgeleitet werden. Es ergibt sich also die Frage, wie aktuell diese Statistikdaten gehalten werden müssen, damit der Optimierer nicht von falschen Voraussetzungen ausgeht, sofern er Kardinalitätsschätzungen vornimmt. Wie häufig müssen Statistiken aktualisiert werden, damit sie nach Datenänderungen nach wie vor repräsentativ sind?

Das zweite Problem mit Statistiken ist ebenfalls allgemein bekannt und gilt auch an dieser Stelle: Die in einer Statistik enthaltene Stichprobe muss charakteristisch sein. Letztlich reduziert eine Statistik die Datenmenge, und hierdurch gehen Informationen verloren. Diese verlorenen Informationen dürfen letztlich nicht dazu führen, dass die Verwendung einer Statistik grob falsche Schlüsse auf die Datenverteilung liefert.

Wir werden diese beiden Probleme im weiteren Verlauf des Kapitels näher untersuchen. Zuvor sind aber noch einige generelle Anmerkungen über den Aufbau von Statistiken erforderlich. Wie immer, soll auch hierfür ein Beispiel verwendet werden. Hierzu legen wir durch das folgende Skript eine Testtabelle an:

```
use QueryTest;
if (object_id('T1', 'U') is not null)
  drop table T1
go
create table T1
 (
   x int not null
  ,a varchar(20) not null
  ,b int not null
  ,y char(20) null
 )
go
insert T1(x,a,b)
   select n % 1000,n%3000,n%5000
     from Numbers
    where n <= 100000
go
create nonclustered index IxT1_x on T1(x)
```

Die erzeugte Tabelle hat vier Spalten. Wir fügen 100.000 Zeilen in die Tabelle ein. Die Spalten x, a und b der Tabelle erhalten dabei unterschiedliche Werte aus dem Bereich 0 bis 5.000. Schließlich erzeugt das Skript noch einen nichtgruppierten Index für die Spalte x.

Wenn wir nun die folgende Abfrage

```
select y from T1 where x=234
```

ausführen und uns den tatsächlichen Ausführungsplan anzeigen lassen, erhalten wir das in Abbildung 9.3 gezeigte Ergebnis.

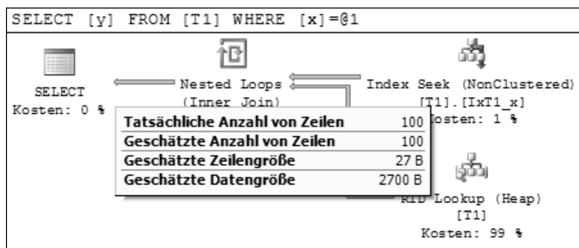


Abbildung 9.3: *Grafischer Ausführungsplan mit tatsächlichen sowie geschätzten Zeilen*

Wie zu erwarten, wird der existierende Index auf der Spalte x durchsucht. Für die gefundenen Zeilen wird dann noch der Wert von y über einen *RowId-Lookup* ermittelt. Sie können auch sehen, dass die Abfrage automatisch parametrisiert wurde. Soweit also nichts Besonderes. Und doch gibt es einen interessanten Aspekt, der bei der Betrachtung des Plans hervorsticht: Die Übereinstimmung zwischen der geschätzten und der tatsächlichen Anzahl Zeilen. Woher nimmt der Optimierer die Schätzung der Zeilenanzahl? Und wie kommt es, dass diese Schätzung sogar exakt mit der tatsächlichen Zeilenanzahl übereinstimmt?

Sie werden es sicherlich bereits vermuten: Bei der Erstellung des Plans verwendet der Optimierer die vorhandenen Statistiken, um die Kardinalität abzuschätzen. Dieser Schritt ist sehr wesentlich für den erzeugten Abfrageplan, da durch ihn ganz entscheidend mitbestimmt wird, ob zum Beispiel ein Index durchsucht wird oder doch lieber ein Scan der gesamten Tabelle erfolgt. Schauen wir uns die Statistiken für unsere Tabelle T1 also einmal an. Öffnen Sie hierzu im Objekt-Explorer den Ordner *QueryTest/Tabellen/dbo.T1/Statistik* (Abbildung 9.4).

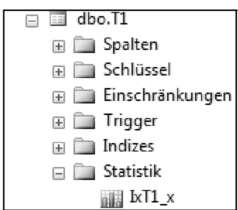


Abbildung 9.4: *Statistiken für die Tabelle T1*

Sie sehen in diesem Ordner eine Statistik, die denselben Namen aufweist wie unser nichtgruppierter Index. Diese Statistik wurde automatisch bei der Erstellung des Index erzeugt. Jeder Index hat immer auch eine entsprechende Statistik gleichen Namens, die automa-

tisch bei der Erstellung oder Neuerstellung (aber nicht beim Reorganisieren!) des Index neu erzeugt wird. Diese Indexstatistiken weisen eine Besonderheit auf: Die entnommene Stichprobe wird stets auf der Basis aller Tabellenzeilen bestimmt. Da bei der Erstellung eines Index sowieso alle Zeilen der Tabelle verarbeitet werden müssen, fließen in diesem Zusammenhang auch alle Tabellendaten in die Stichprobe ein. Dies ist bei anderen Statistiken in der Regel nicht der Fall. Dort werden jeweils nur zufällig ausgewählte Tabellendaten für die Stichprobe verwendet.

Sie können detaillierte Informationen über die Statistik erhalten, indem Sie aus dem Kontextmenü den Eintrag EIGENSCHAFTEN auswählen. Auf der Seite ALLGEMEIN sehen Sie zunächst, welche Spalten die Statistik beinhaltet und vor allem, wann diese Statistik zuletzt aktualisiert wurde. Sie haben dort auch die Möglichkeit, eine Aktualisierung sofort zu veranlassen.

Interessanter sind die auf der Seite DETAILS dargestellten Informationen, die Auskunft über die Datenverteilung in dieser Statistik liefern. Für unsere Statistik ergibt sich ein Bild, wie Abbildung 9.5 in gezeigt.

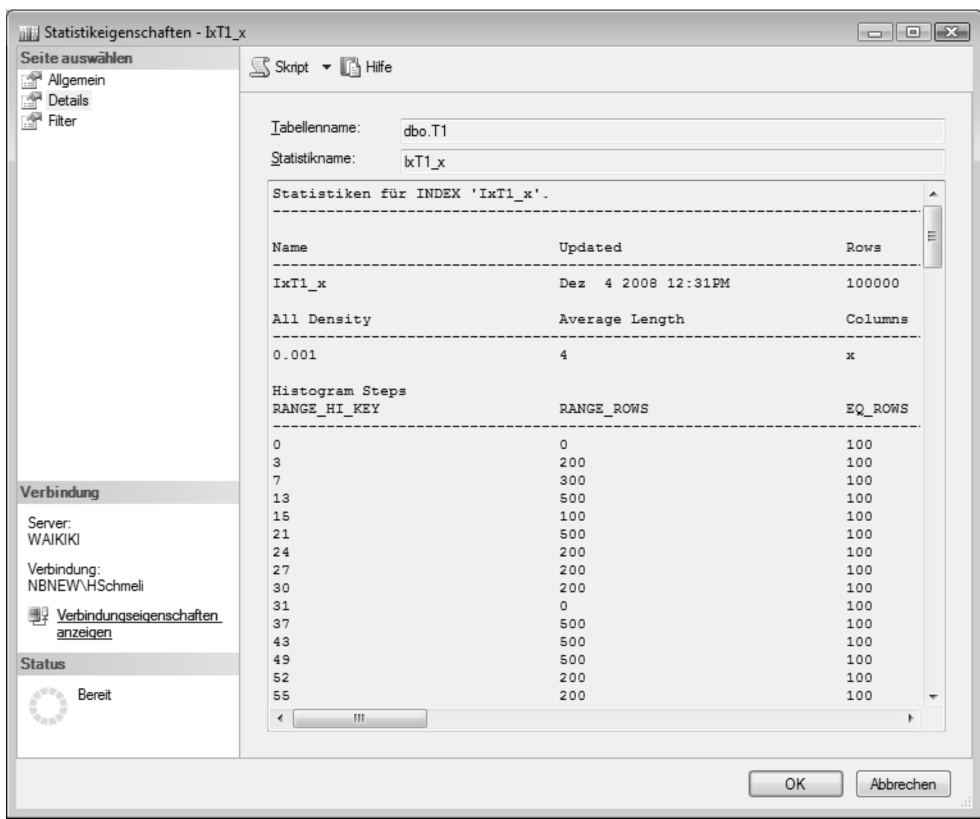


Abbildung 9.5: *Statistikdetails für »IxT1_x«*

Aufschlussreich sind die Informationen in dem Fenster unterhalb des Namens der Statistik (die leider nicht komplett in den Bildschirmausdruck passen). Diese Informationen teilen sich in drei Bereiche:

1. **Allgemeine Information**. Dieser Bereich enthält allgemeine Informationen zur Erstellung der Statistik, sowie die Anzahl der Zeilen im Index und die Anzahl der für die Erzeugung der Stichprobe ausgewerteten Zeilen.
2. **Verteilungsdichte insgesamt**. An dieser Stelle finden Sie die Dichte der Statistik für jede aufgenommene Spalte. Diese Dichte gibt letztlich die erwartete allgemeine Verteilung der Daten in der entsprechenden Tabelle bzw. dem Index an. Der Wert berechnet sich durch den Ausdruck *1/unterschiedliche Werte* für die angegebene Spalte. In unserem Fall gibt es insgesamt 1.000 unterschiedliche Werte für die Spalte `x`. Daher ist die Dichte 1/1000 = 0,001. Der Optimierer verwendet diesen Wert im Zusammenhang mit der Gesamtanzahl der Zeilen zur Abschätzung der Kardinalität, falls eine detaillierte Abschätzung über das Histogramm nicht durchgeführt werden kann.
3. **Histogramm**. Die hier gezeigte Tabelle gibt die genaue Verteilung der erhobenen Stichprobe wieder. In der Tabelle ist für Intervalle der ersten Spalte im Index verzeichnet, wie viele Zeilen jeweils in das Intervall fallen. Durch dieses Histogramm kann der Optimierer einen Plan erstellen, der genau auf die in einer Abfrage verwendeten Bedingungen zugeschnitten ist. So ist es zum Beispiel denkbar, dass für die Abfrage

   ```
   select * from T where c1 = 100
   ```

   ein Table Scan durchgeführt wird, weil der Optimierer aus dem Histogramm entnimmt, dass es insgesamt 10.000 Zeilen gibt, die diese Bedingung erfüllen. Die folgende Abfrage

   ```
   select * from T where c1 = 1
   ```

   kann dagegen möglicherweise einen Index Seek verwenden, weil es nur eine einzige Zeile gibt, welche die Bedingung erfüllt. Tabelle 9.1 erklärt die im Histogramm enthaltenen Spalten.

Spalte	Bedeutung
RANGE_HI_KEY	Dieser Wert zeigt die obere Intervallgrenze an.
RANGE_ROWS	Dies ist die Anzahl der Zeilen, die in das Intervall fallen, das durch RANGE_HI_KEY der vorherigen Zeile und RANGE_HI_KEY der aktuellen Zeile gebildet wird. Die obere Intervallgrenze, also der RANGE_HI_KEY der aktuellen Zeile, wird hierbei ausgeschlossen.
EQ_ROWS	Der Wert gibt an, wie viele Zeilen insgesamt auf der oberen Intervallgrenze (also auf RANGE_HI_KEY) liegen. Falls Sie also die Anzahl der Zeilen inklusive der unteren und oberen Grenze wissen möchten, so müssen Sie den hier dargestellten Wert und den bei RANGE_ROWS angegebenen Wert addieren.

***Tabelle 9.1:** Erklärung der im Histogramm dargestellten Spalten*

Spalte	Bedeutung
DISTINCT_RANGE_ROWS	An dieser Stelle finden Sie die Anzahl der unterschiedlichen Intervallwerte, wiederum unter Ausschluss der oberen Grenze.
AVG_RANGE_ROWS	Dies ist die mittlere Zeilenanzahl für jeden unterschiedlichen Intervallwert (wobei auch hier die obere Grenze ausgeschlossen ist) – also das Ergebnis des Ausdrucks RANGE_ROWS/DISTINCT_RANGE_ROWS.

Tabelle 9.1: *Erklärung der im Histogramm dargestellten Spalten (Forts.)*

Kommen wir nun darauf zurück, woher der Optimierer die geschätzte Zeilenanzahl von 100 nimmt. Für die Abschätzung der Zeilenanzahl untersucht der Optimierer die für den Index `IxT1_x` existierende Statistik gleichen Namens. Abbildung 9.6 zeigt einen Auszug aus dem Histogramm dieser Statistik.

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
226	500	100	5	100
232	500	100	5	100
238	500	100	5	100
244	500	100	5	100
247	200	100	2	100

Abbildung 9.6: *Auszug aus dem Histogramm der Statistik »IxT1_x«*

Interessant sind in unserem Fall die Spalten `RANGE_HI_KEY` und `AVG_RANGE_ROWS`. Der gesuchte Wert 234 fällt in das Stichprobenintervall zwischen 232 und 238. In der Spalte `AVG_RANGE_ROWS` für die Zeile mit `RANGE_HI_KEY = 238` können Sie – genauso wie der Optimierer – ablesen, dass für jeden separaten Wert in diesem Intervall 100 Zeilen existieren. Da die `SELECT`-Anweisung die Zeilen für genau einen Wert abfragt (`WHERE x = 234`), ist der Schätzwert somit gefunden: 100.

Betrachten Sie nun einmal die folgende Abfrage:

```
select y from T1 where a='234'
```

Es wird wieder die Spalte `y` abgefragt, wobei aber diesmal nach dem Wert »234« in der Spalte `a` gefiltert wird. Wie kann der Optimierer hier einen Abfrageplan erstellen, wo es doch gar keine Statistik für die Datenverteilung der Spaltenwerte in der Spalte `a` gibt? Wo nimmt der Optimierer die Schätzwerte her, die noch dazu ziemlich exakt sind (siehe Abbildung 9.7)?

Table Scan
Scannt die Zeilen einer Tabelle.

Physischer Vorgang	Table Scan
Logischer Vorgang	Table Scan
Tatsächliche Anzahl von Zeilen	34
Geschätzte E/A-Kosten	0,412755
Geschätzte CPU-Kosten	0,110157
Anzahl von Ausführungen	1
Geschätzte Anzahl von Ausführungen	1
Geschätzte Operatorkosten	0,522912 (100 %)
Geschätzte Unterstrukturkosten	0,522912
Geschätzte Anzahl von Zeilen	33,0625

Abbildung 9.7: *Geschätzte und tatsächliche Zeilenanzahl*

Die Antwort ist verblüffend einfach: Der Optimierer erkennt, dass keine entsprechende Statistik existiert und veranlasst die Erstellung einer passenden Statistik vor der Erstellung des Ausführungsplans. (Es gibt auch andere Möglichkeiten, auf die wir etwas weiter unten zu sprechen kommen.) Der Plan selber wird dann auf der Basis einer frisch erstellten, also aktuellen Statistik erstellt.

Wenn Sie sich die für die Tabelle `T1` existierenden Statistiken im Objekt-Explorer ansehen, erkennen Sie die automatisch erstellte Statistik (Abbildung 9.8).

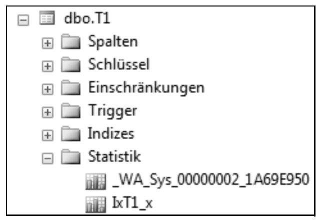


Abbildung 9.8:
Die automatisch erstellte Statistik für die Klausel »WHERE a='234'«

Es ist die erste Statistik mit dem zufällig erzeugten Namen `_WA_Sys_00000002_1A69E950`.

Das Management Studio hat ein Problem mit der Darstellung der Detaildaten für diese Statistik. Lassen Sie sich die Eigenschaften anzeigen, so werden Sie feststellen, dass Sie keine Detaildaten sehen. Natürlich existiert aber auch für diese Statistik ein entsprechendes Histogramm – es wird nur nicht dargestellt. Glücklicherweise können die kompletten Informationen über einen entsprechenden `DBCC`-Befehl angezeigt werden:

```
dbcc show_statistics(T1,_WA_Sys_00000002_1A69E950)
```

Das Kommando gibt die drei Informationsbereiche für die Statistik im Ergebnisbereich aus. Leider können Sie mittels dieses Kommandos immer nur die Daten einer einzigen Statistik abfragen. Eine Übersicht über alle Statistiken in allen Tabellen einer Datenbank erhalten Sie über die Sichten `sys.stats` und `sys.stats_columns`. Allerdings bekommen Sie dort keine Informationen über das Histogramm; dafür müssen Sie `DBCC` verwenden.

Wir setzen unsere Experimente nun mit der folgenden Abfrage fort:

```
select y from T1 where a='234' and b=1234
```

Die Verwendung der Spalten `a` und `b` in der Filterbedingung wirft die gleiche Frage auf wie im vorangegangenen Beispiel: Wie kommt der Optimierer an eine geeignete Statistik, welche die Datenverteilung für die Filterbedingung repräsentiert? Für die Spalte `a` gibt es eine solche Statistik bereits; aber was ist mit der Spalte `b`? Nun, auch hier wieder wird diese Statistik automatisch vor der Erstellung des eigentlichen Abfrageplans erzeugt. Sie können sich davon überzeugen, wenn Sie den entsprechenden Zweig im Objekt-Explorer anzeigen bzw. aktualisieren.

Wir vergleichen im Ausführungsplan wieder die geschätzte Anzahl von Zeilen mit der tatsächlichen Zeilenanzahl. In Abbildung 9.9 sehen Sie das Resultat.

Table Scan	
Scannt die Zeilen einer Tabelle.	
Physischer Vorgang	Table Scan
Logischer Vorgang	Table Scan
Tatsächliche Anzahl von Zeilen	7
Geschätzte E/A-Kosten	0,412755
Geschätzte CPU-Kosten	0,110157
Anzahl von Ausführungen	1
Geschätzte Anzahl von Ausführungen	1
Geschätzte Operatorkosten	0,522912 (100 %)
Geschätzte Unterstrukturkosten	0,522912
Geschätzte Anzahl von Zeilen	1

Abbildung 9.9:
Geschätzte und tatsächliche Zeilenanzahl einer kombinierten Filterbedingung

Hier verschätzt sich der Optimierer. Der absolute Unterschied ist in unserem Beispiel nicht dramatisch und hat keinen Einfluss auf die Erstellung des Plans. Es ist aber denkbar, dass dieser Unterschied durchaus die Erzeugung eines optimalen Plans behindern kann.

Woher kommt dieser Unterschied? Die Ursache liegt darin, dass automatisch erstellte Statistiken nicht über mehrere Spalten erstellt werden. Für die Filterbedingung unserer Abfrage (`WHERE a='234' AND b=1234`) wäre eine Statistik, welche die Verteilung beider Spalten gemeinsam berücksichtigt, sicherlich besser geeignet als eine separate Statistik je Spalte. Die separaten Statistiken für die Spalten `a` und `b` müssen bei der Erstellung des Plans für Kardinalitätsschätzungen kombiniert werden; der dabei ermittelte Schätzwert ist offensichtlich nicht ganz korrekt. Wie gesagt: Für unser einleitendes Beispiel ist diese Fehleinschätzung nicht relevant. Etwas weiter unten werden Sie ein Beispiel mit drastischeren Auswirkungen sehen.

Die Lösung wäre eine Statistik, die eine Stichprobe über beide Spalten ermittelt. Eine solche Statistik wird nicht automatisch erstellt, wir können sie jedoch manuell erzeugen. Hierzu verwenden wir das Kommando `CREATE STATISTICS`:

```
create statistics s1 on T1(b,a)
```

Das Kommando erwartet zunächst den Namen der zu erstellenden Statistik. Hinter der `ON`-Klausel müssen dann noch der Name der Tabelle und nachfolgend (in Klammern) die Spalten für die Statistik angegeben werden.

Wir führen die Abfrage mit der neu erstellten Statistik nun nochmals aus und vergleichen wieder die geschätzte Zeilenanzahl mit der tatsächlichen (Abbildung 9.10).

Table Scan	
Scannt die Zeilen einer Tabelle.	
Physischer Vorgang	Table Scan
Logischer Vorgang	Table Scan
Tatsächliche Anzahl von Zeilen	7
Geschätzte E/A-Kosten	0,412755
Geschätzte CPU-Kosten	0,110157
Anzahl von Ausführungen	1
Geschätzte Anzahl von Ausführungen	1
Geschätzte Operatorkosten	0,522912 (100 %)
Geschätzte Unterstrukturkosten	0,522912
Geschätzte Anzahl von Zeilen	6,66667

Abbildung 9.10:
Verbesserung der geschätzten Zeilenanzahl durch eine kombinierte Statistik

Der Schätzwert ist nun erheblich besser und kommt der tatsächlichen Zeilenanzahl sehr nahe.

Fassen wir also noch einmal zusammen:

1. Der Optimierer benötigt möglichst aktuelle Statistiken für die Erstellung effizienter Abfragepläne. Generell können diese Statistiken automatisch oder manuell erstellt werden. Die automatische Erstellung wird einerseits bei der Indexerstellung vorgenommen, indem für den Index eine Statistik erstellt wird, welche die Verteilung der Indexspalten repräsentiert. Diese Statistik kann auch über mehrere Spalten allgemeine Verteilungen enthalten, falls der Index mehrere Spalten enthält.
2. Auf der anderen Seite werden Statistiken automatisch erstellt, wenn der Optimierer diese Statistiken vermisst. Auf diese Weise erzeugte Statistiken werden stets nur für eine Spalte erstellt.
3. In einigen Fällen können manuell erstellte Statistiken, die sich über mehrere Spalten erstrecken, den Optimierer bei Kardinalitätsschätzungen unter Umständen besser unterstützen als die automatisch erstellten Statistiken allein.
4. Ein Histogramm wird auch bei Statistiken über mehrere Spalten nur für die erste Spalte einer Statistik erstellt.

9.2.1 Erstellen und Aktualisieren von Statistiken

Im vorangegangenen Abschnitt haben Sie bereits die automatische und manuelle Erzeugung von Statistiken kennengelernt. In diesem Abschnitt soll nun noch einmal näher darauf eingegangen werden, welche Faktoren und Optionen sowohl bei der Erstellung als auch bei der Aktualisierung von Statistiken eine Rolle spielen. Dabei werden wir zunächst die Erstellung und anschließend die Aktualisierung von Statistiken untersuchen.

Erstellen von Statistiken

Das Erstellen von Statistiken kann entweder automatisch oder manuell erfolgen. Ich empfehle Ihnen aber sehr, den automatischen Mechanismus zu verwenden. Die Einstellung, dass Statistiken automatisch erstellt werden sollen, ist eine Option der Datenbank. Wenn Sie an der Systemdatenbank *model* nichts verändert haben und diese Datenbankoption nicht explizit gesetzt haben, werden Ihre Statistiken automatisch erstellt. Dies ist also der Standard nach der Installation von SQL Server. Falls Sie diese Option ändern möchten, können Sie dies über den Eigenschaftendialog der Datenbank erledigen. Auf der Seite OPTIONEN finden Sie die Option STATISTIKEN AUTOMATISCH ERSTELLEN (siehe Abbildung 9.11).

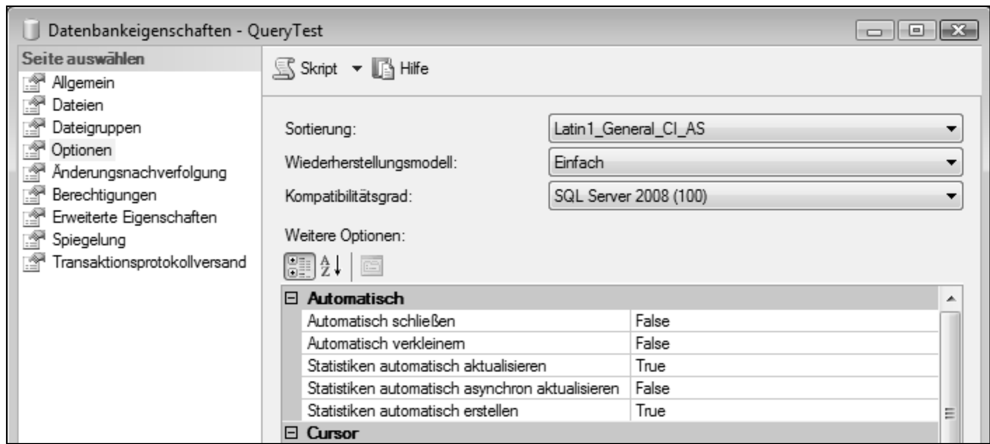


Abbildung 9.11: *Datenbankoptionen für Statistikerstellung und Statistikaktualisierung*

Selbstverständlich existiert hierfür auch ein entsprechendes `ALTER DATABASE`-Kommando:

```
alter database <dbname> set auto_create_statistics [off | on]
```

Falls Sie die automatische Erstellung von Statistiken ausschalten, werden fehlende Statistiken nicht mehr automatisch ergänzt. Dies betrifft nur fehlende Statistiken, die der Optimierer bei der Planerstellung vermisst. Statistiken, die automatisch bei der Indexerstellung erzeugt werden, werden auch bei ausgeschalteter Automatik angelegt. Wie gesagt: Sie sollten die automatische Erstellung einschalten, da Sie dann in jedem Fall auf der sicheren Seite sind. Falls Sie auf die Automatik verzichten, müssen Sie die Statistiken manuell erstellen. Hierfür verwenden Sie das Kommando `CREATE STATISTICS`, so wie im vorigen Abschnitt gezeigt. Mit diesem Kommando können Sie allerdings nur jeweils eine einzelne Statistik erzeugen. Eine Verwendung der gespeicherten Prozedur `sp_createstats` ist da möglicherweise die einfachere Lösung für die manuelle Erstellung von Statistiken. Diese Prozedur erstellt einspaltige Statistiken für alle Spalten in allen Tabellen einer Datenbank. Natürlich können Sie diese Prozedur auch verwenden, wenn Sie die automatische Erstellung von Statistiken eingeschaltet haben und die Prozedur beispielsweise einmal pro Nacht laufen lassen. Allerdings werden Sie dann sehr wahrscheinlich eine Vielzahl von Statistiken erstellen, die Sie niemals benötigen. Auf der anderen Seite müssen fehlende Statistiken nicht erst bei der Planerstellung ergänzt werden, was die Abfrageausführung beschleunigen kann. Das Online-Ergänzen fehlender Statistiken kann die erste Ausführung einer Abfrage ganz erheblich negativ beeinflussen. Der Profiler bietet ein Ereignis an, mit dem Sie die automatische Erstellung und Aktualisierung von Statistiken beobachten können: *Performance/Auto Stats*. Schauen Sie sich dazu bitte Abbildung 9.12 an.

EventClass	TextData	Duration
Trace Start		
Auto Stats	Created: a	1004
SQL:BatchCompleted	dbcc freeproccache dbcc dr...	1208
SQL:BatchCompleted	dbcc freeproccache dbcc dr...	231
Trace Stop		

```
dbcc freeproccache
dbcc dropcleanbuffers
select y from T1 where a='234'
```

Abbildung 9.12: *Das »Auto Stats«-Ereignis im Profiler*

Die dargestellte Ablaufverfolgung enthält die Ereignisse *TSQL/SQL:BatchCompleted* und *Performance/Auto Stats*. Protokolliert wird die Ausführung der `SELECT`-Anweisung aus dem vorangegangenen Abschnitt, die eine automatische Erstellung einer Statistik für die Spalte `a` der Tabelle `T1` hervorgerufen hat. Die `SELECT`-Anweisung wird zwei Mal ausgeführt, wobei vor jeder Ausführung sowohl der Datencache als auch der Plancache geleert werden. Sie sehen deutlich, dass die zweite Ausführung mit der bereits existierenden Statistik um einen Faktor 5 bis 6 schneller ist als beim ersten Mal. Dies liegt daran, dass die Erstellung der Statistik beim ersten Aufruf den Hauptteil der Ausführungszeit benötigt hat. Denken Sie bitte an dieses Szenario, wenn Sie den Effekt beobachten, dass die erste Ausführung einer Abfrage lange dauert und dann in der Folge alle weiteren Ausführungen schnell gehen. Die erste Vermutung ist in so einem Fall immer, dass der Datencache beim ersten Mal noch »kalt« war, und daher die physischen Leseoperationen für den Unterschied in der Ausführungszeit verantwortlich sind. In den meisten Fällen ist diese Vermutung auch absolut richtig. Es kann allerdings auch sein, dass beim ersten Aufruf fehlende Statistiken erstellt oder existierende Statistiken aktualisiert werden mussten. In unserem Beispiel ist nur eine Statistik erstellt worden. Es kann natürlich auch vorkommen, dass bei komplexeren Abfragen zunächst mehrere Statistiken erstellt werden müssen, und dann kann die dafür benötigte Zeit um ein Vielfaches höher sein als die für die eigentliche Abfrageausführung erforderliche Zeit.

Wenn Sie also beobachten, dass nur die erste Ausführung einer Abfrage lange dauert, so untersuchen Sie bitte auch, ob automatisch ergänzte oder aktualisierte Statistiken die Ursache sein könnten. Falls die erste Ausführung einer Abfrage auch mit leeren Caches deutlich länger dauert als nachfolgende Ausführungen derselben Abfrage (ebenfalls mit leeren Caches), dann sind wahrscheinlich fehlende oder nicht aktuelle Statistiken die Ursache hierfür. Genau dies ist im obigen Beispiel der Fall.

Für eine Beobachtung können Sie den Profiler oder auch einen passenden Auflistsatz verwenden. Eine entsprechende Untersuchung kann übrigens durchaus knifflig sein. Die bei der ersten Abfrageausführung automatisch hinzugefügten Statistiken können nämlich nicht gelöscht werden. Dadurch ist es enorm schwierig, die Situation nachzustellen oder nachzuvollziehen.

Aktualisieren von Statistiken

Wie bereits in der Einleitung zu diesem Abschnitt erwähnt, enthalten Statistiken letztlich redundante Daten, die mit den realen Tabellen- oder Indexdaten synchronisiert werden müssen. Die beste Statistik nützt natürlich nichts, wenn sie fehlerhafte Rückschlüsse auf die tatsächlichen Daten zulässt oder verursacht. Es ist daher offensichtlich, dass Statistiken auf irgendeine Weise mit den existierenden Daten synchronisiert werden müssen. Für diese Synchronisation sind verschiedene Ansätze denkbar. So könnte man zum Beispiel alle Datenänderungen auch online in die Statistiken einfließen lassen – und dadurch sehr wahrscheinlich das System komplett überlasten. Auf der anderen Seite wäre sicherlich auch eine monatliche Aktualisierung denkbar, wodurch möglicherweise die in den Statistiken enthaltenen Stichproben nicht mehr viel mit der Realität zu tun hätten, was wiederum ineffiziente Abfragepläne zur Folge hätte.

Eine vernünftige Verfahrensweise zur Aktualisierung von Statistiken liegt sicherlich irgendwo zwischen den beiden Situationen. SQL Server bietet zunächst einmal die Möglichkeit, Statistiken automatisch zu aktualisieren. Auch dies ist eine Option der Datenbank. In Abbildung 9.11 können Sie die entsprechenden Einstellungen sehen. Selbstverständlich haben Sie auch hier die Möglichkeit, die Konfiguration über `ALTER DATABASE`-Kommandos durchzuführen:

```
alter database <dbname> set auto_update_statistics [on | off]
alter database <dbname> set auto_update_statistics_async [on | off]
```

Durch dieses Kommando veranlassen Sie, dass nicht aktuelle Statistiken automatisch aktualisiert werden. Wie Sie anhand der Kommandos bereits erkennen, erlaubt SQL Server zwei Aktualisierungsmodi:

1. Synchrone Aktualisierung von Statistiken. Wenn der Optimierer feststellt, dass Statistiken nicht mehr aktuell sind, werden diese Statistiken vor der Kompilierung bzw. Re-Kompilierung des Plans aktualisiert. Erst wenn die Aktualisierung abgeschlossen ist, wird der Abfrageplan erstellt und die Abfrage ausgeführt.
2. Asynchrone Aktualisierung von Statistiken. Der Optimierer erkennt auch hier, dass Statistiken aktualisiert werden müssen. Allerdings wird in diesem Fall ein Abfrageplan mit den veralteten Statistiken verwendet und die Abfrage mit diesem Plan sofort ausgeführt. Die Aktualisierung der Statistiken erfolgt im Hintergrund. Die asynchrone Aktualisierung ist hierbei eine Zusatzoption der Aktualisierung, kann also nur verwendet werden, sofern Sie die automatische Aktualisierung generell einschalten.

Beide Verfahren haben natürlich ihre Vor- und Nachteile, sodass Sie selbst abwägen müssen, für welche Option Sie sich entscheiden. Und natürlich gibt es auch noch eine dritte Möglichkeit: Genau wie die automatische Erstellung können Sie auch die automatische Aktualisierung gänzlich ausschalten. In diesem Fall müssen Sie die Aktualisierung manuell durchführen. Für diese manuelle Aktualisierung können Sie das Kommando `UPDATE STATISTICS` verwenden, mit dem Sie jeweils eine Statistik aktualisieren. Etwas komfortabler ist die Verwendung der gespeicherten Prozedur `sp_updatestats`, die eine Aktualisierung aller vorhandenen Statistiken mit nur einem Aufruf ermöglicht. Falls Ihnen die Einstellung auf Datenbankebene zu »grob« ist, können Sie über die Prozedur `sp_autostats` auch pro Tabelle oder Index die automatische Aktualisierung ein- bzw. ausschalten.

Auch wenn Sie die automatische Aktualisierung verwenden, was ausdrücklich empfehlenswert ist, ist es eine gute Idee, `sp_updatestats` periodisch (etwa im Rahmen eines Wartungsplans) aufzurufen. Dadurch minimieren Sie die Wahrscheinlichkeit, dass der Optimierer erst bei der Abfrageausführung eine Aktualisierung nicht aktueller Statistiken vornimmt, und beschleunigen dadurch die Ausführung Ihrer Abfragen.

Die Prozedur `sp_updatestats` aktualisiert die vorhandenen Statistiken auf der Basis einer Stichprobe aus Tabellen- und Indexdaten.

Denken Sie bitte daran, dass `sp_updatestats` nur eine Stichprobe der Daten für die Aktualisierung verwendet. Eine Praxis, die ich oft beobachte, ist der Aufruf von `sp_updatestats` in einem nächtlichen Wartungsplan nach einer Neu-Indizierung. Diese Verfahrensweise ist ausdrücklich nicht empfehlenswert! Bei der Indexerstellung wird ja für jeden Index ebenfalls eine Statistik erstellt. Diese Statistik basiert aber nicht nur auf einer Stichprobe, sondern auf allen Tabellenzeilen. Dies bedeutet, dass eine Statistik, die mit einem Index verbunden ist, nach der Indizierung eine sehr gute Qualität besitzt. Wenn Sie nun nach der Indexerstellung `sp_updatestats` aufrufen, werden die Indexstatistiken überschrieben, wobei diesmal jeweils nur eine Stichprobe der Daten entnommen wird. Dadurch ersetzen Sie möglicherweise qualitativ hochwertige Statistiken durch weniger gute. Außerdem ist das Aktualisieren einer Indexstatistik durch `sp_updatestats` natürlich überflüssig, wenn der Index gerade neu erstellt wurde. Rufen Sie in einem Wartungsplan also zumindest `sp_updatestats` vor der Reorganisation der Indizes auf.

Es bleibt die Frage zu klären, woran der Optimierer erkennt, dass eine Statistik nicht mehr aktuell ist.

SQL Server führt intern für jede Spalte einer Tabelle einen Zähler mit, der bei einer Veränderung der Spaltendaten inkrementiert wird. Dieser *colmodctr* bestimmt letztlich, wann eine mit einer Spalte verbundene Statistik nicht mehr aktuell ist. Für Tabellen mit einer Zeilenanzahl von über 500 Zeilen ist eine Statistik dann nicht mehr aktuell, sobald die Anzahl der Änderungen der ersten Spalte in einer Statistik größer oder gleich 20 Prozent der zum Zeitpunkt der letzten Aktualisierung der Statistik in der Tabelle enthaltenen Zeilen zuzüglich 500 ist.

Ein kleines Rechenbeispiel soll dies verdeutlichen. Unsere im vorangegangenen Abschnitt erstellte Tabelle `T1` hat insgesamt 100.000 Zeilen. Für die automatisch erstellte Statistik auf der Spalte `a` sind also zunächst diese 100.000 Zeilen für eine Kontrolle der Aktualität maßgebend. Wenn nun der Wert der Spalte `a` in 10.000 Zeilen geändert wird, dann wurden insgesamt 10 Prozent der für die Aktualität der Statistik relevanten Daten geändert. Damit ist die Statistik nach wie vor aktuell. Werden weitere 10.000 Zeilen verändert, dann sind insgesamt 20 Prozent der Werte für die in der Statistik verwendete Spalte `a` verändert worden; die Statistik ist also immer noch aktuell. Auch nach der Änderung weiterer 499 Zeilen ist die Statistik aktuell. Ändern wir nun noch eine Zeile, dann wird die Statistik das nächste

Mal, wenn sie benötigt wird, aktualisiert. Sie können das folgende Skript ausprobieren, falls Sie das gerade Gesagte nachvollziehen möchten.

```
-- Änderung von 20.000 Zeilen
update top(20000) T1 set a='###'
-- Keine Aktualisierung
select y from T1 where a='234'

-- Änderung weiterer 499 Zeilen
update top(499) T1 set a='###'
-- Keine Aktualisierung
select y from T1 where a='234'

-- Änderung einer Zeile
update top(1) T1 set a='###'
-- Hier erfolgt die Aktualisierung
select y from T1 where a='234'
```

Prägen Sie sich bitte ein, dass die Aktualisierung der Statistik nicht bei der Aktualisierung der Tabellendaten erfolgt, sondern erst dann, wenn der Optimierer die Statistik für die Planerstellung benötigt. Wenn die Statistik aktualisiert wird, erfolgt auch eine Re-Kompilierung der Abfrage.

Entgegen der üblichen »Gesprächigkeit« von SQL Server, wenn es darum geht, interne Informationen preiszugeben, können Sie die *colmodctr*-Werte übrigens nicht abfragen. Diese Information ist intern und bleibt für den Anwender bzw. Administrator unsichtbar, was ich persönlich schade finde. Dadurch fehlt die Möglichkeit herauszufinden, welche Statistiken kurz vor der Aktualisierung stehen.

Die 20 %-Schwelle kann für einige Fälle zu groß sein. Dies sollten Sie beachten, wenn Sie Probleme mit Abfragen feststellen. Sie müssen also möglicherweise überprüfen, ob Ihre Statistiken aktuell sind. Hierzu können Sie die Systemfunktion `stats_date()` verwenden, mit der Sie das Aktualisierungsdatum einer Statistik abfragen. Im Zusammenhang mit der Systemsicht `sys.stats`, die alle Statistiken einer Datenbank zurückgibt, können Sie diese Funktion wie folgt verwenden:

```
select object_name(object_id) as ObjName
      ,name as statistik
      ,stats_date(object_id, stats_id) as AktualisiertAm
  from sys.stats
 where objectproperty(object_id, 'IsUserTable') = 1
 order by AktualisiertAm
```

Die Abfrage gibt alle in der aktuellen Datenbank enthaltenen Statistiken, sortiert nach dem Datum ihrer letzten Aktualisierung, zurück.

Nicht aktuelle Statistiken

Zum Abschluss dieses Abschnitts möchte ich Ihnen noch an einem Beispiel zeigen, welche Auswirkungen nicht aktuelle Statistiken auf die Abfrageleistung haben können.

Wir erstellen für dieses Beispiel eine Tabelle namens Produkt:

```
use QueryTest;
if (object_id('Produkt', 'U') is not null)
  drop table Produkt
go
create table Produkt
 (
   Id int identity(1,1) not null
  ,Preis decimal(8,2) not null
  ,LetzteAktualisierung date not null default current_timestamp
  ,Spalten nchar(500) not null default '#'
 )
go
alter table Produkt add constraint PK_Produkt
 primary key clustered (Id)
```

Die Tabelle soll unter anderem das Datum der letzten Aktualisierung der Produktdaten enthalten. Nach diesem Datum wollen wir später suchen.

Das nachfolgende Skript fügt 500.000 Produkte in die Tabelle ein, wobei die Spalten LetzteAktualisierung und Preis mit zufälligen Werten gefüllt werden:

```
insert Produkt(LetzteAktualisierung, Preis)
  select dateadd(day, abs(checksum(newid())) % 3250,'20000101')
        ,0.01*(abs(checksum(newid())) % 20000)
    from Numbers
   where n <= 500000
```

Da nach der letzten Aktualisierung gesucht werden soll, erstellen wir für diese Spalte einen Index:

```
create nonclustered index ix_Prod on Produkt(LetzteAktualisierung)
```

Bei der Erstellung des Index wird automatisch auch eine entsprechende Statistik angelegt.

Stellen Sie sich nun vor, dass durch eine Firmenübernahme 100.000 weitere Produkte am 01.12.2008 hinzukommen. Dies wird durch das folgende Skript simuliert:

```
insert Produkt(LetzteAktualisierung,Preis)
  select '20081201', 100 from Numbers where n <= 100000
```

Da der gruppierte Index nun wahrscheinlich stark fragmentiert ist, wollen wir ihn neu aufbauen, bevor wir fortfahren:

```
alter index PK_Produkt on Produkt rebuild
```

Die Anzahl der durch die letzte Anweisung geänderten Zeilen ist zu klein für eine automatische Aktualisierung der Statistiken. Insgesamt wurden nämlich lediglich 100.000 von 500.000 Zeilen, also 20 Prozent der Zeilen, geändert. Die folgende Abfrage wird daher mit veralteten Statistiken ausgeführt:

```
select *
  from Produkt
 where LetzteAktualisierung = '20081201'
```

Der verwendete Abfrageplan basiert also auf nicht aktuellen Statistiken und ist deshalb nicht optimal. In Abbildung 9.13 sehen Sie den grafischen Ausführungsplan.

SELECT
Kosten: 0 %

Nested Loops
(Inner Join)
Kosten: 0 %

Index Seek (NonClustered
[Produkt].[ix_Prod]
Kosten: 50 %

Schlüsselsuche (Clusterec
[Produkt].[PK_Produkt]
Kosten: 50 %

Index Seek (NonClustered)
Scannt einen bestimmten Bereich von Zeilen aus einem nicht gruppierten Index.

Physischer Vorgang	Index Seek
Logischer Vorgang	Index Seek
Tatsächliche Anzahl von Zeilen	100000
Geschätzte E/A-Kosten	0,003125
Geschätzte CPU-Kosten	0,0001581
Geschätzte Anzahl von Ausführungen	1
Anzahl von Ausführungen	1
Geschätzte Operatorkosten	0,0032831 (50 %)
Geschätzte Unterstrukturkosten	0,0032831
Geschätzte Anzahl von Zeilen	1

Abbildung 9.13: *Ein Ausführungsplan mit nicht aktuellen Statistiken*

Im Abfrageplan wird ein Index Seek verwendet, weil der Optimierer sich in der Anzahl der Zeilen verschätzt – und zwar ganz gewaltig. Die geschätzte Zeilenanzahl ist 1, die tatsächliche Anzahl von Zeilen ist 100.000. Für die Abfrage einer einzigen Zeile (so wie geschätzt), ist ein Index Seek natürlich der perfekte Operator. Bei 100.000 Zeilen sieht das allerdings anders aus. Hier wäre ein Clustered Index Scan eindeutig die bessere Wahl. Wie kommt nun aber der Optimierer zu seiner fehlerhaften Einschätzung? – Die Ursache ist die nicht aktuelle Statistik.

Wenn Sie sich das Histogramm der Statistik ansehen, finden Sie tatsächlich keine Einträge für das Datum 01.12.2008. Abbildung 9.14 zeigt das untere Ende der Histogramm-Daten.

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
2008-04-13	1292	180	8	161,5
2008-05-04	2964	136	20	148,2
2008-05-19	2188	134	14	156,2857
2008-07-24	10015	171	65	154,0769
2008-10-11	11941	138	78	153,0897
2008-11-12	4774	172	31	154
2008-11-23	1544	139	10	154,4

Abbildung 9.14: *Das untere Ende der Histogramm-Daten für den Index »ix_Prod«*

Hinzu kommt, dass die hinzugefügten Datumswerte allesamt außerhalb des existierenden Histogramms liegen. Alle hinzugefügten Werte sind größer als der größte im Histogramm enthaltene Wert. Deshalb kann der Optimierer keine Schätzung abgeben, die auf einer vorhandenen Statistik basiert. Er geht in diesem Fall davon aus, dass der exakte Vergleich mit einem Datumswert (`WHERE LetzteAktualisierung = '20081201'`) nur eine Zeile liefert.

Die Abfrage hat insgesamt 300.128 logische Lesevorgänge benötigt (ermittelt mit `SET STATISTICS IO ON`). Beachten Sie bitte, dass der dargestellte Plan zwar der Plan ist, nach dem die Abfrage ausgeführt wird, die im Plan dargestellten Kosten sind jedoch nicht die tatsächlich entstandenen Kosten! Der Plan wird vor der Ausführung erstellt, wobei der Optimierer eine Minimierung der erwarteten Kosten vornimmt. Die im Plan gezeigten Kosten sind denn auch erwartete Kosten, welche in unserem Beispiel nichts mit der Realität zu tun haben.

Es gibt verschiedene Möglichkeiten, dieses Problem zu lösen. Sie können sich zum Beispiel mit einem Abfragehinweis behelfen:

```
select *
  from Produkt with (index=0)
 where LetzteAktualisierung = '20081201'
```

Über `INDEX=0` wird festgelegt, dass kein Index verwendet werden soll. Nun wird also ein Clustered Index Scan durchgeführt, und die Anzahl der benötigten Lesevorgänge ist tatsächlich kleiner als bei Verwendung des Index, nämlich auf meiner Maschine nun nur noch 86.035, also mehr als zwei Drittel weniger als zuvor!

Allerdings sollten Sie Abfragehinweise, die den Optimierer zu etwas zwingen, was er normalerweise nicht tun würde, vermeiden. In unserem Fall ist es durchaus denkbar, dass zu einem späteren Zeitpunkt für das Datum 01.12.2008 die Suche über den Index doch irgendwann sinnvoll wird, weil die Daten sich entsprechend geändert haben. Sofern dieser Fall eintritt, wird durch die obige Abfrage die Indexverwendung unterbunden – was natürlich nicht erwünscht ist. Die Entscheidung über die Indexverwendung sollten Sie tatsächlich dem Optimierer überlassen. Generell ist es wesentlich besser, wenn Sie den Optimierer verstehen und mit ihm arbeiten, als versuchen, ihn auf irgendeine Weise auszutricksen.

Ein Indexhinweis hilft hier zwar in erster Instanz, ist jedoch absolut nicht empfehlenswert. Besser ist es, wenn Sie dafür sorgen, dass der Optimierer mit aktuellen Statistiken arbeiten kann, die in diesem Fall eben aktueller sein müssen, als es der Standard vorschreibt. (Denken Sie an die oben präsentierte 20 %-Regel.) Dann tritt das Problem gar nicht erst nicht auf. Hier hilft zum Beispiel die oben erwähnte gespeicherte Systemprozedur `sp_updatestats` weiter, die Sie im einfachsten Fall so aufrufen können:

```
sp_updatestats
```

Dadurch werden alle bestehenden Statistiken einer Datenbank aktualisiert. Wenn Sie sich nun noch einmal das Histogramm der Statistik anzeigen lassen, zum Beispiel durch das Kommando:

```
dbcc show_statistics(Produkt,ix_Prod) with histogram
```

dann werden Sie auch einen Eintrag für den 01.12.2008 im Histogramm finden. Starten Sie jetzt die ursprüngliche Abfrage erneut, so erhalten Sie im Ausführungsplan den erwarteten Clustered Index Scan (Abbildung 9.15).

SELECT
Kosten: 0 %

Clustered Index Scan (Clustered)
[Produkt].[PK_Produkt]

Clustered Index Scan (Clustered)

Scannt einen gruppierten Index, entweder vollständig oder nur einen Bereich.

Physischer Vorgang	Clustered Index Scan
Logischer Vorgang	Clustered Index Scan
Tatsächliche Anzahl von Zeilen	100000
Geschätzte E/A-Kosten	63,495
Geschätzte CPU-Kosten	0,660157
Anzahl von Ausführungen	1
Geschätzte Anzahl von Ausführungen	1
Geschätzte Operatorkosten	64,1551 (100 %)
Geschätzte Unterstrukturkosten	64,1551
Geschätzte Anzahl von Zeilen	102187

Abbildung 9.15: *Geänderter Ausführungsplan mit aktuellen Statistiken*

Achten Sie bitte auf die geschätzte Anzahl Zeilen, die nun – mit der aktualisierten Statistik – wesentlich näher an der Realität ist als zuvor.

Falls Sie nun denken, dass das gerade präsentierte Beispiel etwas weit hergeholt ist, dann irren Sie sich. Das Beispiel zeigt einen Fall, der in der Praxis recht häufig anzutreffen ist. Für viele Spalten trifft die Eigenschaft zu, dass die gespeicherten Werte ständig anwachsen. Dadurch kann es immer wieder vorkommen, dass Sie in Abfragen Parameter verwenden, für die noch keine Statistiken im Histogramm existieren. Denken Sie nur an Primärschlüssel, deren Werte automatisch vergeben werden, zum Beispiel durch Verwendung von `IDENTITY`-Spalten. Sobald Sie Zeilen zu einer Tabelle hinzufügen, werden die Werte der hinzugefügten Spalten stets außerhalb der bestehenden Histogramm-Daten liegen, sodass für entsprechende Abfragen keine Einträge im Histogramm existieren. Falls solche Werte in Abfragen verwendet werden, muss der Optimierer irgendwelche Annahmen treffen, die möglicherweise nicht korrekt sind. Daher sollten Sie erwägen, die Statistiken für solche Spalten häufiger zu aktualisieren, als der automatische Algorithmus dies erledigt.

Auf jeden Fall ist es eine gute Idee, alle Statistiken einmal pro Tag durch den Aufruf von `sp_updatestats` zu aktualisieren. Zusätzlich sollten Sie die Prozedur `sp_updatestats` auch nach größeren Datenänderungen (zum Beispiel `BULK IMPORT`) aufrufen. Bitte fangen Sie aber nicht an, `sp_updatestats` etwa jede Stunde zu starten. Das Aktualisieren von Statistiken benötigt natürlich Systemressourcen und sollte daher möglichst zu Zeiten geringer sonstiger Aktivitäten durchgeführt werden.

Noch eine Anmerkung zum Schluss: Wenn Sie mit manuell erstellten Statistiken arbeiten, sollten Sie darauf achten, dass Sie keine Statistiken erstellen, die es bereits gibt. Falls mehrere Statistiken für dieselbe(n) Spalte(n) existieren, muss der Optimierer sich für die Verwendung einer Statistik entscheiden. Bei dieser Entscheidung wird er die nach seiner Meinung »bessere« Statistik auswählen. Dies ist zum Beispiel die Statistik mit dem jüngeren Aktualisierungsdatum oder mit mehr Einträgen in der Stichprobe. Dies kann dazu führen, dass Re-Kompilierungen stattfinden, weil die für einen Plan verwendeten Statistiken sich ändern.

9.2.2 Probleme mit Statistiken

Im vorangegangenen Abschnitt haben Sie gesehen, wie wichtig möglichst aktuelle Statistiken für die Abfrageleistung sind. Sofern Sie dafür sorgen, dass die Aktualisierungsrate Ihrer Statistiken angemessen ist (was »angemessen« in diesem Zusammenhang konkret bedeutet, müssen Sie leider für Ihre Umgebung selbst herausfinden), kann der Optimierer stets einen effizienten Plan erstellen.

Es gibt jedoch eine Situation, in der der Optimierer überfordert ist, weil die vorhandenen Statistiken für eine Kardinalitätsschätzung nicht geeignet sind – mögen sie auch noch so aktuell sein. Wie immer, möchte ich Ihnen diesen Fall anhand eines Beispiels verdeutlichen.

Wir legen für dieses Beispiel eine Tabelle mit Personendaten an (etwa für demoskopische Statistiken), wobei uns hier nur die Spalten `Altersklasse` und `Gehalt` interessieren:

```
use QueryTest;
-- Lege eine Tabelle mit Personendaten an
if (object_id('Person', 'U') is not null)
  drop table Person
go
create table Person
 (
  Id int identity(1,1) not null primary key nonclustered
 ,Altersklasse nchar(4) not null
 ,Gehalt       int not null default 0
 ,Spalten      nchar(200) not null default '#'
 )
```

Die vierte Spalte dient wiederum nur als Platzhalter, um zu simulieren, dass eine Zeile noch mehr Spalten enthält.

Nun fügen wir Daten in die Tabelle ein, wobei als eine Besonderheit berücksichtigt wird, dass das Gehalt normalerweise mit der Erfahrung, also der höheren Altersklasse wächst. Als Altersklassen tragen wir die Werte »AK10« bis »AK70« in Zehnerschritten ein. Das entsprechende Skript sieht so aus:

```
-- Füge 1.000.000 Zeilen in die Tabelle ein
with Altersklassen(AK) as
 (
  select str(10+10*(abs(checksum(newid()))%7), 2)
    from Numbers
   where n <= 1000000
 )
insert Person (Altersklasse, Gehalt)
 select 'AK' + AK
       ,case AK
          when 10 then 0
          when 20 then 20000 + abs(checksum(newid())) % 10000
          when 30 then 30000 + abs(checksum(newid())) % 10000
          when 40 then 40000 + abs(checksum(newid())) % 20000
          when 50 then 50000 + abs(checksum(newid())) % 20000
```

```
          when 60 then 60000 + abs(checksum(newid())) % 30000
          else 20000 + abs(checksum(newid())) % 100000
       end
   from Altersklassen
```

Wir wissen, dass diese Tabelle nach Altersklasse und Gehalt durchsucht werden soll, und legen deshalb einen Index auf diesen beiden Spalten an:

```
create nonclustered index Ix_AK on Person(Altersklasse,Gehalt)
```

Da für die folgenden Abfragen auch die physikalischen E/A-Vorgänge gemessen werden sollen, wird nun noch ein `CHECKPOINT` veranlasst, damit alle im Speicher geänderten Seiten auf die Festplatte geschrieben werden:

```
checkpoint
```

Wir möchten jetzt alle Personen abfragen, die in der Altersklasse »AK60« mit einem Gehalt zwischen 30.000 und 40.000 vorhanden sind:

```
dbcc dropcleanbuffers
select *
  from Person
 where Altersklasse = 'AK60'
   and Gehalt between 30000 and 40000
       option (maxdop 1)
```

Für die Messung der E/A-Operationen wird zunächst der Datencache geleert. Außerdem verwenden wir die Option `MAXDOP 1`, damit die Abfrage nicht parallel ausgeführt wird. Dadurch ist der Vergleich mit der geänderten Abfrage etwas weiter unten einfacher und auch realistischer.

Sehen Sie sich den Ausführungsplan an, so (siehe Abbildung 9.16) stellen Sie fest, dass ein Clustered Index Scan durchgeführt wird.

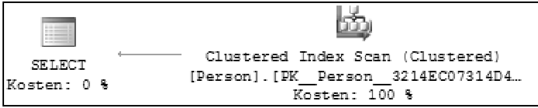


Abbildung 9.16: *Nicht optimaler Ausführungsplan durch Clustered Index Scan*

Der eigens für die Suche erstellte Index wurde ignoriert. Dies ist doch merkwürdig, oder? Immerhin gibt die Abfrage keine Zeile zurück; und das legt die Vermutung nahe, dass die Verwendung des Index sehr wohl sinnvoll gewesen wäre. Die Selektivität des Index für die Abfrageparameter ist sicherlich sehr gut, denn für »AK60« wurden keinerlei Zeilen eingefügt. Dies können Sie sofort sehen, wenn Sie nochmals das weiter oben verwendete Skript zur Erzeugung der Tabellendaten betrachten.

Also probieren wir den Index doch einmal aus, indem wir einen entsprechenden Abfragehinweis verwenden:

```
dbcc dropcleanbuffers
-- Mit Indexhinweis
select *
  from Person with (index=Ix_AK)
 where Altersklasse = 'AK60'
   and Gehalt between 30000 and 40000
       option (maxdop 1)
```

Statt der Klausel `WITH(INDEX=Ix_AK)` hätten wir auch den allgemeinen Indexhinweis `WITH (FORCESEEK)`verwenden können, um dem Optimierer mitzuteilen, dass er einen möglichst passenden Index auswählen soll. Dadurch wäre die Abfrage nicht mehr abhängig von der Existenz des Index `Ix_AK`.

Der erzeugte Ausführungsplan sieht nun natürlich anders aus, nämlich so, wie in Abbildung 9.17 gezeigt.

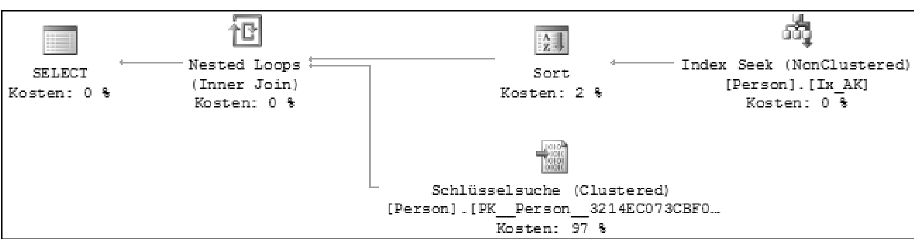


Abbildung 9.17: *Ausführungsplan mit Verwendung des Index auf »(AK, Gehalt)«*

Etwas eigenartig sieht der Plan allerdings immer noch aus. So ist zum Beispiel auf den ersten Blick nicht klar, wozu der Sort-Operator existiert. Das Problem ist, dass der Optimierer sich zwar zur Verwendung des Index überreden (oder besser: zwingen) lässt. Dies ändert jedoch nichts an der Kardinalitätsschätzung. Wenn Sie sich im Plan die geschätzten Werte für die Zeilenanzahl ansehen, werden Sie feststellen, dass der Optimierer ganz gehörig daneben liegt. In meinem Experiment beträgt die Schätzung für den Index Seek knapp 20.000 Zeilen, tatsächlich werden aber nur null Zeilen gefunden. Da der Plan auf der Grundlage der Schätzungen erstellt wird, geht der Optimierer eben auch davon aus, dass in den oberen Zweig des Nested Loop-Operators ca. 20.000 Zeilen einfließen, und fügt einen Sort-Operator ein, damit diese Zeilen zuvor sortiert werden. Das Ziel ist letztlich eine Beschleunigung des Nested Loop-Operators.

Interessanterweise ist also der durch einen Indexhinweis erzwungene Plan nicht optimal, weil sich der Optimierer eben nach wie vor in der Zeilenanzahl verschätzt. Trotzdem benötigt die Abfrage mit dem Indexhinweis erheblich weniger Ressourcen als zuvor. Daher erfüllt der (nicht optimale) Plan letztlich seinen Zweck.

In Tabelle 9.2 habe ich einmal für beide Abfragen die ermittelten Statistiken eingetragen.

	Lesevorgänge		**Zeit**	
	Physikalisch	**Logisch**	**CPU**	**Dauer**
ohne Index-Hinweis	1.192	52.829	0,61 s	15,36 s
mit Index-Hinweis	3	4	0,00 s	0,04 s

Tabelle 9.2: *Ergebnisse der Abfrage mit und ohne Index-Abfragehinweis*

Sie können deutlich erkennen, dass die Fehleinschätzung des Optimierers in diesem Fall zu einem sehr ineffizienten Plan führt.

Was ist aber die Ursache dafür? Veraltete Statistiken können es sicherlich nicht sein, denn wir haben nach der Erstellung des nichtgruppierten Index `Ix_AK` keinerlei Datenänderungen mehr vorgenommen. Und doch muss die Ursache für die Fehleinschätzung irgendwie in den Statistiken zu finden sein. Das bedeutet: Es ist möglich, dass auch aktuelle Statistiken zu falschen Kardinalitätsschätzungen und letztlich nicht optimalen Ausführungsplänen führen.

Sehen wir uns die Statistiken also einmal an und versuchen herauszufinden, warum der Optimierer für das Prädikat

```
where Altersklasse = 'AK60' and Gehalt between 30000 and 40000
```

20.000 Zeilen erwartet, obwohl tatsächlich keine Zeile existiert.

Zunächst sollte Ihnen auffallen, dass für die Spalte `Gehalt` automatisch eine Statistik erzeugt wurde. Diese Statistik wurde erstellt, weil in unserer `WHERE`-Bedingung nach `Gehalt` gefiltert wird. Folglich wird für diese Spalte auch ein Histogramm benötigt. Die für den Index über die beiden Spalten `Altersklasse` und `Gehalt` automatisch erstellte Statistik stellt dieses Histogramm nicht zur Verfügung, da ein Histogramm stets nur für die führende Spalte einer Statistik angefertigt wird.

In Abbildung 9.18 sehen Sie das Histogramm für die mit dem Index `Ix_AK` verbundene Statistik, wobei die interessante Zeile in der Abbildung hervorgehoben ist.

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
AK10	0	142742	0	1
AK20	0	143659	0	1
AK30	0	142861	0	1
AK40	0	142882	0	1
AK50	0	142553	0	1
AK60	0	142714	0	1
AK70	0	142589	0	1

Abbildung 9.18: *Histogramm für die Statistik »Ix_AK«*

Unser Prädikat

```
where Altersklasse = 'AK60' and Gehalt between 30000 and 40000
```

führt dazu, dass nur die erste Spalte des Index auf (`Altersklasse, Gehalt`) verwendet wird. Dies wird aus der Bedingung `Altersklasse = 'AK60'` abgeleitet. Anhand des Histogramms

für den vorhandenen Index auf der Spalte `Altersklasse` erkennt der Optimierer, dass insgesamt 142.714 Zeilen für den Wert »AK60« existieren (siehe Abbildung 9.18). Damit ist der Index nicht selektiv genug. Der Optimierer geht nun einen Schritt weiter und untersucht auch das Prädikat `Gehalt between 30000 and 40000`, allerdings nicht innerhalb des Index, sondern separat. Hierzu wird die Statistik für die Spalte `Gehalt` benötigt und demzufolge auch erzeugt.

Aus dem Histogramm dieser Statistik entnimmt der Optimierer die Tatsache, dass insgesamt (in meinem Experiment) 138.259 Zeilen existieren, für die der Wert der Spalte `Gehalt` zwischen 30.000 und 40.000 liegt. Dies können Sie anhand des Histogramms nachrechnen, indem Sie einfach die entsprechenden Werte für `RANGE_ROWS` und `EQ_ROWS` addieren. Falls Ihnen das zu mühsam ist, führen Sie einfach diese Abfrage aus:

```
select Altersklasse
  from Person
 where Gehalt between 30000 and 40000
```

Schauen Sie sich bitte im Ausführungsplan die geschätzte Zeilenanzahl an.

Bei insgesamt 1.000.000 Zeilen in der Tabelle hat das Prädikat `Gehalt between 30000 and 40000` also eine Selektivität von 138.259/1.000.000 14 % für die gesamte Tabelle. Der Optimierer wendet nun diese Selektivität auch auf die Datenverteilung im Index an. Hier existieren für `Altersklasse = 'AK60'` insgesamt 142.714 Zeilen. Und damit ergibt sich die Schätzung dass die Abfrage 14 Prozent von 142.714 Zeilen liefert, also knapp 20.000 Zeilen.

Die mathematisch Interessierten unter Ihnen werden das Problem bereits erkannt haben: Der Optimierer untersucht die Datenverteilung in den Statistiken für die Spalten `Altersklasse` und `Gehalt` unabhängig voneinander – und das ist in diesem Fall schlichtweg nicht korrekt. Aus der Gesamtdatenverteilung der Spalte `Gehalt` zu schließen, dass diese Verteilung auch für den Wert `Altersklasse = 'AK60'` gelten würde, ist deshalb falsch, da die Spaltenwerte korrelieren. Derzeit werden solche Abhängigkeiten bei der Erstellung und Auswertung von Statistiken allerdings nicht berücksichtigt.

Spalten, deren Werte voneinander abhängig sind, werden Sie normalerweise in der Praxis recht selten antreffen. Von daher ist das obige Beispiel vielleicht etwas zu akademisch. Es sollte Ihnen allerdings verdeutlichen, dass der Optimierer auch seine Grenzen hat.

In dem obigen Beispiel hätte es geholfen, den Index auf (`Altersklasse, Gehalt`) als gruppierten Index anzulegen. In diesem Fall wird der gruppierte Index für die Suche verwendet, weil dies die kostengünstigste Möglichkeit darstellt. Irgendwie spielt dabei aber auch der Zufall eine Rolle, denn die Kardinalitätsschätzung ist nach wie vor falsch (siehe Abbildung 9.19). Achten Sie in der Abbildung auf die geschätzte Anzahl Zeilen, die nach wie vor nichts mit der Realität zu tun hat!

SELECT
Kosten: 0 %

Clustered Index Seek (Cluste
[Person].[Ix_AK]
Kosten: 100 %

Clustered Index Seek (Clustered)
Scannt einen bestimmten Bereich von Zeilen aus einem gruppierten Index.

Physischer Vorgang	Clustered Index Seek
Logischer Vorgang	Clustered Index Seek
Tatsächliche Anzahl von Zeilen	0
Geschätzte E/A-Kosten	0,926829
Geschätzte CPU-Kosten	0,0248657
Geschätzte Anzahl von Ausführungen	1
Anzahl von Ausführungen	1
Geschätzte Operatorkosten	0,951694 (100 %)
Geschätzte Unterstrukturkosten	0,951694
Geschätzte Anzahl von Zeilen	22462,5

Abbildung 9.19: *Der zufälligerweise korrekte Ausführungsplan*

Die Suche auf dem gruppierten Index wird hier verwendet, weil dies die effizienteste Möglichkeit ist – und zwar unabhängig von der zu verarbeitenden Anzahl von Zeilen. Die Aussage, dass einzelne Statistiken stets als voneinander unabhängig bewertet werden, ist natürlich dennoch gültig. Der Plan ist einfach nur zufälligerweise effizient.

9.3 Parametrisierte Abfragen

In Abschnitt 9.1.3 haben Sie eine erste Einführung in die Parametrisierung von Abfragen erhalten. In diesem Abschnitt soll nun detaillierter auf diesen Punkt eingegangen werden. Sie werden dabei die Vor- und Nachteile der Parametrisierung kennenlernen und anschließend wissen, welche Besonderheiten Sie in Bezug auf Parametrisierung unbedingt beachten sollten.

Parametrisierung ist ein sehr wesentliches Konzept, wobei generell zwei Methoden unterschieden werden: Zum einen existiert die automatische Parametrisierung, bei welcher der Optimierer entscheidet, ob eine Abfrage parametrisiert wird. Zum anderen kann eine Parametrisierung durch die Verwendung der Prozedur `sp_executesql` oder eine entsprechende Datenbankeinstellung auch erzwungen werden. Beide Verfahrensweisen werden wir im vorliegenden Abschnitt näher untersuchen.

9.3.1 Positive Auswirkungen der Parametrisierung

Durch eine Parametrisierung werden letztlich zwei Ziele verfolgt:

1. **Erhöhung der Wiederverwendbarkeit von gespeicherten Ausführungsplänen**. Durch die Parametrisierung werden Ausführungspläne klassifiziert. Für jeden Vertreter einer Klasse wird nur noch ein Plan gespeichert. Diese Verfahrensweise erhöht letztlich die Wiederverwendbarkeit gespeicherter Ausführungspläne, da die Wahrscheinlichkeit steigt, dass für eine Abfrage bereits ein parametrisierter Ausführungsplan, sozusagen als Muster, im Cache vorgefunden wird. Dadurch werden CPU-intensive Kompiliervorgänge eingespart, was sich positiv auf die Leistung auswirkt.

2. **Verringerung des vom Plancache benötigten Speichers**. Dieser Punkt ergibt sich unmittelbar aus der Reduzierung der im Cache gespeicherten Pläne. Die Verringerung des Plancache reduziert auch den Hauptspeicherbedarf von SQL Server insgesamt. Dadurch steht alles in allem mehr Hauptspeicher für den Datencache zur Verfügung; und das hat natürlich positive Auswirkungen auf die Abfrageleistung.

Der Unterschied zwischen Parametrisierung und Nicht-Parametrisierung kann hierbei wirklich enorm sein, wie das folgende Beispiel beweist. In diesem Beispiel führen wir eine identische Abfrage in zwei Versuchen jewils 5.000 Mal aus. Im ersten Versuch verzichten wir dabei zunächst auf eine Parametrisierung:

```
-- 1. Versuch. Ohne Parametrisierung
set nocount on
dbcc freeproccache
dbcc dropcleanbuffers
go
declare @i int
       ,@cmd nvarchar(200)
set @i = 0
while (@i <= 5000)
 begin
   set @cmd = 'declare @x int;
               select @x=checksum_agg(checksum(*))
                 from sys.all_columns
                where object_id=' + cast(@i as nvarchar(30))
   exec (@cmd)
   set @i = @i + 1
 end
go
```

Durch das dynamische Erzeugen der SQL-Anweisung und die Ausführung über die `EXEC()`-Funktion wird eine Parametrisierung unterbunden. Dadurch wird in jedem Schleifendurchlauf erneut ein Ausführungsplan erstellt und im Plancache gespeichert.

Die Ausführung des obigen Skripts hat auf meiner schon leicht in die Jahre gekommenen Maschine insgesamt 48 Sekunden gedauert.

Wenn Sie sich nach der Ausführung der Abfrage den Plancache ansehen (verwenden Sie hierzu die am Beginn dieses Kapitels in Abschnitt 9.1 präsentierte Abfrage), werden Sie feststellen, dass der Cache mehr oder weniger voll ist. Bei mir sind nach der Ausführung ca. 4.300 Pläne im Cache – und jeder dieser Pläne sieht gleich aus. Allein die Tatsache, dass nicht alle 5.000 Pläne im Cache Platz gefunden haben, lässt schon vermuten, dass Ausführungspläne aus dem Cache entfernt wurden, weil SQL Server über nicht genügend Hauptspeicher verfügt (auf meinem PC ist für SQL Server 1 GByte Speicher zugewiesen).

Wenn wir uns die Speicherverwendung ansehen, wird die Situation sofort klar. Sie können die Speicherverwendung zum Beispiel mit dem Kommando `DBCC MEMORYSTATUS` abfragen, um eine umfassende Auskunft zu erhalten. Die Auslastung des Plancache erhalten Sie auch über eine Abfrage der dynamischen Verwaltungssicht `sys.dm_os_memory_cache_counters`. Es ist der Counter des Typs `CACHESTORE_SQLCP`, der Auskunft über die Speicher-

verwendung durch Ad-Hoc-Abfragepläne gibt. Die folgende Abfrage liefert diese Informationen:

```
select single_pages_kb+multi_pages_kb
      +single_pages_in_use_kb+multi_pages_in_use_kb as AdHoc_Kb
      ,entries_count
  from sys.dm_os_memory_cache_counters
 where type = 'CACHESTORE_SQLCP'
```

Auf meinem PC enthält der Plancache ca. 4.300 Einträge und ist insgesamt ungefähr 590 MByte groß.

Sie können auch den Bericht zur Arbeitsspeichernutzung der SQL Server-Instanz ausführen, den Sie in Kapitel 4 kennengelernt haben. Abbildung 9.20 zeigt den Bereich, in dem die Pufferseitenverteilung dargestellt wird.

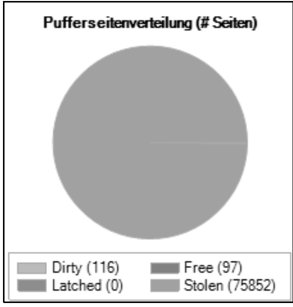


Abbildung 9.20: *Speicherverteilung mit vollem Plancache*

Es ist klar zu erkennen, dass die »gestohlenen« Seiten den Hauptanteil des Arbeitsspeichers belegen. In unserem Fall ist dies hauptsächlich der Plancache. Im Datencache sind insgesamt nur noch 97 Seiten, also etwa 776 kByte frei.

Wir wollen nun den Versuch wiederholen, wobei wir diesmal eine parametrisierte Abfrage verwenden. Dadurch sollte nur ein einziger Plan im Plancache existieren. Außerdem ist dann auch nur eine einzige Kompilierung erforderlich. Dadurch sollte die Schleife deutlich schneller sein.

Das folgende Skript verwendet eine parametrisierte Abfrage durch den Einsatz der gespeicherten Prozedur `sp_executesql`, die wir etwas später in Abschnitt 9.3.3 noch einmal näher betrachten werden.

```
-- 2. Versuch. Mit Parametrisierung.
set nocount on
dbcc freeproccache
dbcc dropcleanbuffers
go
declare @i int
       ,@x int
       ,@cmd nvarchar(200)
set @i = 0
```

```
set @cmd = 'select @x=checksum_agg(checksum(*))
              from sys.all_columns
             where object_id=@i'
while (@i <= 5000)
 begin
   exec sp_executesql @cmd, N'@x int out, @i int', @x=@x, @i=@i
   set @i = @i + 1
 end
go
```

Diesmal hat die Ausführung nicht einmal eine Sekunde gedauert! Der Plancache enthält zwei Einträge und ist insgesamt lediglich 464 kByte groß. Die Auslastung des Plancache ist also um mehr als den Faktor 1.000 kleiner als zuvor.

Für die Aufteilung des Hauptspeichers können wir eine ähnlich drastische Verbesserung beobachten, wie in Abbildung 9.21 zu sehen ist.

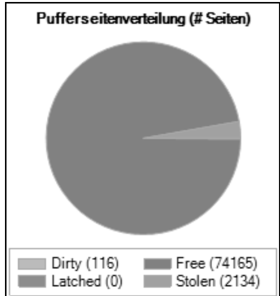


Abbildung 9.21: *Speicheraufteilung nach Ausführung der parametrisierten Abfrage*

Im Datencache sind nun 74.165 Seiten, also knapp 580 MByte, frei.

Insgesamt zeigt das durchgeführte Experiment also sehr deutlich, dass sich eine Parametrisierung lohnt. Sie sollten daher darauf achten, dass Ihre Anwendungen parametrisierte Abfragen verwenden. Oftmals ist es leider so, dass Anwendungen den Text der SQL-Abfragen inklusive der verwendeten Abfrageparameter zusammensetzen und diesen fertigen Text dann zur Ausführung an SQL Server übergeben. Betrachten Sie hierzu das folgende Beispiel, das einen Auszug aus einer Client-Anwendung, die ADO.NET verwendet, zeigt:

```
SqlCommand cmd = new SqlCommand();
cmd.CommandText = "select * from Person where PersonID=" + Id.ToString();
```

Diese Abfrage wird sehr wahrscheinlich nicht parametrisiert. Daher wird jede Ausführung der Abfrage eine Kompilierung und einen Eintrag im Prozedurcache bewirken. Viel besser ist es, die Abfrage wie folgt zu programmieren:

```
SqlCommand cmd = new SqlCommand();
cmd.CommandText = "select * from Person where PersonID=@Id";
cmd.Parameters.Add("@Id", SqlDbType.Int);
```

In diesem Fall findet eine Parametrisierung statt. Welche Vorteile sich daraus ergeben, können Sie nun beurteilen.

Oftmals werden Sie die Entwicklung der Client-Anwendungen nicht dahingehend beeinflussen können, auf welche Weise SQL-Abfragen erstellt werden. Falls Sie die Möglichkeit haben, hier mitreden zu dürfen, denken Sie bitte daran, dass Parametrisierung eingesetzt werden sollte. Sie haben in jedem Fall die Möglichkeit, den Plancache dahingehend zu beobachten, ob die im Cache gespeicherten Pläne wiederverwendet werden, oder ob die vorhandenen Pläne nur ein einziges Mal benötigt werden. Die dynamische Verwaltungssicht `sys.dm_exec_query_stats` enthält eine Spalte `execution_count`, aus der Sie ablesen können, wie häufig ein gespeicherter Plan verwendet wurde. Sie können diese Spalte zu der am Anfang dieses Kapitels präsentierten Abfrage, die den Plancache auswertet, hinzufügen und so beobachten, ob in Ihrer Umgebung eine Wiederverwendung der gespeicherten Pläne stattfindet. Falls die meisten Abfragepläne einen `execution_count` von 1 besitzen, haben Sie wahrscheinlich ein Problem mit nicht parametrisierten Abfragen.

Zum Abschluss möchte ich Sie noch auf eine kleine Merkwürdigkeit in Bezug auf die automatische Parametrisierung hinweisen. Schauen Sie sich bitte die folgenden vier Abfragen an, die allesamt für eine automatische Parametrisierung in Frage kommen:

```
select * from msdb.dbo.backupfile where backup_set_id=100
go
select * from msdb.dbo.backupfile where backup_set_id=1000
go
Select * From msdb.dbo.backupfile Where backup_set_id=10000
go
SELECT * from msdb.dbo.backupfile where backup_set_id=100000
```

Die Frage ist nun, wie viele Einträge im Plancache diese vier Abfragen jeweils erzeugen? Sicher ist auf jeden Fall, dass die Antwort irgendwo zwischen eins und vier liegt.

Nun, es werden drei Pläne erstellt. Warum das? Wieso reicht nicht ein einziger Plan aus?

Wenn Sie vermuten, dass der leicht unterschiedliche SQL-Text der Abfragen der Auslöser dafür ist, dass drei Ausführungspläne erstellt werden, dann liegen Sie mit dieser Vermutung falsch. Dieser Unterschied ist nicht maßgeblich, wenn eine Abfrage parametrisiert wird; dies kann also nicht die Ursache sein.

Der Grund wird sofort klar, wenn wir uns ansehen, in welcher Form der Text der Abfrage von der Funktion `sys.dm_exec_sql_text` zurückgegeben wird. Die folgende Abfrage ermittelt diese Information:

```
select t.text as SqlAnweisung
      ,execution_count as AnzahlAusführungen
  from sys.dm_exec_query_stats
       cross apply sys.dm_exec_sql_text(sql_handle) as t
```

Das Ergebnis sehen Sie in Abbildung 9.22.

SqlAnweisung	AnzahlAusführungen
(@1 tinyint)SELECT * FROM [msdb].[dbo].[backupfile] WHERE [backup_set_id]=@1	1
(@1 int)SELECT * FROM [msdb].[dbo].[backupfile] WHERE [backup_set_id]=@1	1
(@1 smallint)SELECT * FROM [msdb].[dbo].[backupfile] WHERE [backup_set_id]=@1	2

Abbildung 9.22: *Die drei unterschiedlichen Ausführungspläne*

Im Abfrageergebnis ist zunächst zu erkennen, dass für alle erstellten Abfragen der Text normalisiert wurde, sodass zum Beispiel die Groß-/Kleinschreibung tatsächlich keine Rolle spielt. Weiter ist zu sehen, dass die drei Pläne insgesamt vier Mal ausgeführt wurden, wobei der letzte Plan zwei Mal verwendet wurde. Wirklich interessant ist jedoch die Art und Weise der Parametrisierung, die jeweils für die Klausel `WHERE backup_set_id=<zahl>` durchgeführt wurde. Ganz offensichtlich findet hier eine implizite Typkonvertierung der Konstanten statt, wobei jeweils der kleinste passende Datentyp verwendet wird. Und so wird für den Wert 100 der Datentyp `TINYINT` angenommen, für 1.000 und 10.000 ist es `SMALLINT`, und für 100.000 wird der Datentyp `INTEGER` verwendet. Warum eine Typkonvertierung in dieser Form notwendig ist, kann ich leider nicht erklären. Der Datentyp der Vergleichsspalte `backup_set_id` ist `INTEGER`. Deshalb wäre es nach meiner Ansicht ausreichend, auch stets eine Konvertierung in den Datentyp `INTEGER` durchzuführen. Die Realität sieht aber anders aus – und somit erhalten Sie letztlich drei Pläne, wo eigentlich auch ein Plan ausgereicht hätte. Dieses Verhalten ist schon etwas merkwürdig, da doch die Parametrisierung gerade deswegen erfunden wurde, um den Plancache zu entlasten, also weniger Pläne im Cache zu speichern.

Selbstverständlich kann das Problem durch passenden SQL-Code umgangen werden. Sie müssen nur auf die implizite Typkonvertierung verzichten und die Konvertierung explizit durchführen, also zum Beispiel so:

```
select * from msdb.dbo.backupfile where backup_set_id=cast(100 as int)
go
select * from msdb.dbo.backupfile where backup_set_id=cast(1000 as int)
go
Select * From msdb.dbo.backupfile Where backup_set_id=cast(10000 as int)
go
SELECT * from msdb.dbo.backupfile where backup_set_id=cast(100000 as int)
```

Jetzt gibt es tatsächlich nur noch einen einzigen gespeicherten Ausführungsplan.

Sie können die gespeicherte Prozedur `sp_get_query_template` aufrufen, falls Sie wissen möchten, in welcher Weise eine Abfrage parametrisiert wird. Diese Prozedur erwartet den Text einer Abfrage als ersten Parameter. Dieser Parameter ist auch ein Ausgabeparameter, in welchem der parametrisierte Text der Abfrage zurückgegeben wird. Der zweite Parameter ist lediglich ein Ausgabeparameter, über den Sie die Parameterinformationen erhalten. Schauen Sie sich bitte das folgende Beispiel für die Verwendung der Prozedur an:

```
declare @query nvarchar(max)
       ,@params nvarchar(max)
exec sp_get_query_template N'select sum(y)
                              from T1
                             where x=12345
                               and b between 10.0 and 20
                               and z < 500'
```

```
                              ,@query out,@params out
-- Ergebnis zurückgeben
select @query as Abfrage, @params as Parameter
```

Abbildung 9.23 zeigt das (etwas überraschende) Ergebnis.

Abfrage	Parameter
select sum (y) from T1 where x = @0 and b between @1 and @2 and z < @3	@0 int,@1 numeric(38,1),@2 int,@3 int

Abbildung 9.23: *Abbildung 10.23:Vorhergesagtes Ergebnis der Parametrisierung*

Überraschend deshalb, weil die vorhergesagte implizite Typkonvertierung ganz offensichtlich anders ist als in Wirklichkeit. So wird zum Beispiel für den Wert 10.0 der Datentyp `NUMERIC(38,1)` verwendet. Dies ist irgendwie nicht der kleinste, sondern eher der größte passende Datentyp; und damit ist das Verhalten abweichend von dem, was wir gerade in unserem Beispiel gesehen haben. `sp_get_query_template` lässt also nur ungefähre Rückschlüsse darauf zu, wie die Parametrisierung in der Realität erfolgt.

Wenn Sie `sp_get_query_template` einmal für unsere vier Beispielabfragen vom Beginn dieses Abschnitts aufrufen, werden Sie herausfinden, dass für alle vier Abfragen das gleiche Muster ermittelt wird, wobei der Datentyp des einzigen Parameters stets `INTEGER` ist. Die Wirklichkeit sieht dann aber eben etwas anders aus.

9.3.2 Probleme mit der Parametrisierung

Die einleitenden Bemerkungen zur Parametrisierung aus Abschnitt 9.1.3 haben bereits angedeutet, dass SQL Server bei der Parametrisierung von Abfragen sehr »konservativ« vorgeht. Sobald die Möglichkeit besteht, dass die Parametrisierung einer Abfrage etwa aufgrund der Tabellenstruktur, der vorhandenen Statistiken oder der verwendeten Abfrageparameter zu Problemen hinsichtlich der Performance führt, wird keine automatische Parametrisierung durchgeführt.

Dieses Verhalten hat natürlich einen Grund. Immerhin wird durch eine Parametrisierung eine Vielzahl von Abfragen auf dasselbe Muster reduziert, also letztlich auf denselben Ausführungsplan. In Abschnitt 9.2 haben Sie aber gerade gesehen, dass die Verwendung von Statistiken eine Erstellung speziell angepasster Ausführungspläne für unterschiedliche Parameter gestattet – eine Verfahrensweise, die völlig konträr zur Parametrisierung ist. Statistiken ermöglichen beispielsweise die Verwendung eines Scan-Operators, falls ein Suchparameter verwendet wird, der während 98 Prozent der vorhandenen Zeilen selektiert, und die Verwendung eines Index Seek-Operators, wenn dieselbe Abfrage mit einem Suchparameter ausgeführt wird, der nur eine einzige Zeile liefert:

```
-- Diese Bedingung liefert 100.000 Zeilen.
where c1=2

-- Jede dieser Bedingungen liefert 1 Zeile.
where c1=1000
where c1=1001
...
where c=100000
```

Bei einer Parametrisierung geht diese Möglichkeit verloren. Hier existiert nur noch ein einziger Ausführungsplan, der für alle Parameter verwendet wird. Dieser Plan wird beim ersten Aufruf der Abfrage erstellt – und zwar mit den bei diesem Aufruf verwendeten Parameterwerten. Ein Problem ergibt sich dann, wenn die beim ersten Aufruf verwendeten Parameterwerte untypisch sind. In diesem Fall wird ein Ausführungsplan erstellt (und später auch wiederverwendet), der für alle anderen Parameterwerte nicht optimal ist.

Betrachten wir hierzu noch einmal das obige Beispiel. Angenommen, der Plancache ist leer und die Abfrage mit der Bedingung `WHERE c1=2` wird ausgeführt. Dadurch wird ein Abfrageplan erstellt, der sehr wahrscheinlich einen Scan der Tabelle oder eines Index durchführt. Wenn nun Abfragen mit einer der Bedingungen `WHERE c1=1000, WHERE c1=1001,` ... insgesamt 2.000.000 Mal ausgeführt werden, dann werden 2.000.000 Scan-Operationen durchgeführt, obwohl an dieser Stelle Index Seeks mit Sicherheit die bessere Wahl gewesen wären.

Das ist also der Preis, den Sie eventuell für eine Parametrisierung bezahlen müssen. In dem gerade durchgeführten Gedankenexperiment wäre die Alternative ohne Parametrisierung, dass der Plancache mit Ausführungsplänen »überflutet« würde, so wie das Beispiel in Abschnitt 9.3.1 gezeigt hat. Beide Varianten sind hier sicherlich nicht optimal. Eventuell wäre es angebracht, zwei Ausführungspläne für die beiden unterschiedlichen Parameterklassen zu haben – also einen Plan für `WHERE c1=2` sowie einen weiteren Plan für `WHERE c1=1000 ... 100000` –, aber diese Möglichkeit existiert leider nicht.

Nehmen wir noch einmal die zu Beginn dieses Kapitels in Abschnitt 9.2 angelegte Tabelle `T1`. Das folgende Skript ändert die Spalte `x` für alle Zeilen dieser Tabelle:

```
update T1 set x = 500
update top(1) T1 set x = 1000
```

Wir haben nun also eine einzige Zeile mit dem Wert `x=1000`. In allen anderen Zeilen hat die Spalte `x` den Wert 500. Diese Situation »schreit« ja wohl geradezu nach Problemen mit der Parametrisierung. Allerdings – oder besser gesagt: zum Glück – erkennt der Optimierer das Problem und parametrisiert die folgende Abfrage daher nicht – was ziemlich clever ist:

```
select y from T1 where x=1000
```

Allerdings ist es sehr wohl möglich, auch für diese Abfrage eine Parametrisierung zu veranlassen – absichtlich oder eher indirekt. Mit dieser Möglichkeit beschäftigen sich die nun folgenden Abschnitte.

9.3.3 Erzwungene Parametrisierung

In einigen Fällen kann es durchaus sinnvoll sein, den sehr »konservativen« automatischen Mechanismus zur Parametrisierung auszuhebeln und eine Parametrisierung von Abfragen zu erzwingen. Denken Sie noch einmal an das gerade am Ende des vorherigen Abschnitts präsentierte Beispiel mit unserer Testtabelle `T1`. Wenn Sie sich die beiden `UPDATE`-Anweisungen, die für eine extreme Datenverteilung der Spalte `x` in dieser Tabelle sorgen, einmal wegdenken, warum soll dann eine Anweisung wie

```
select y from T1 where x=100
```

nicht parametrisiert werden? Wenn Sie sich vorstellen, dass eine Abfrage in dieser Form 100.000 Mal am Tag mit unterschiedlichen Suchparametern aufgerufen wird, dann wäre eine solche Parametrisierung bei der existierenden Datenverteilung mit Sicherheit sinnvoll. Die Parametrisierung würde dafür sorgen, dass nur noch ein Plan existiert, der einen Index Seek für die Suche verwendet, weil entweder 100 Zeilen oder keine Zeile von der Abfrage zurückgeliefert werden. Durch eine Parametrisierung könnte in diesem Fall also sehr viel Speicher im Plancache eingespart werden.

Wir müssten demnach dafür sorgen, dass die Abfrage trotzdem parametrisiert wird und sozusagen die Entscheidung des Optimierers in dieser Hinsicht revidieren. Aber wie?

Es gibt generell zwei Möglichkeiten, eine Parametrisierung absichtlich zu veranlassen:

1. Erzwungene Parametrisierung für die gesamte Datenbank einschalten. Durch das Kommando

   ```
   alter database QueryTest set parameterization forced
   ```

   veranlassen Sie eine automatische Parametrisierung aller Abfragen, die bei ihrer Ausführung die Datenbank `QueryTest` im Kontext haben. Denken Sie bitte daran, dass die Datenbank im Kontext der Verbindung ausgewählt sein muss, andernfalls findet keine erzwungene Parametrisierung statt und es wird auf die automatische Parametrisierung zurückgefallen. Die folgende Anweisung bewirkt also keine Parametrisierung der gestellten Abfrage, falls die Option `PARAMETRIZATION FORCED` für die Datenbank `QueryTest` eingeschaltet wurde:

   ```
   use master;
   select y from QueryTest.dbo.T1 where x=100
   ```

   Die folgenden Abfragen hingegen führen beide zu einer Parametrisierung:

   ```
   use QueryTest;
   select y from QueryTest.dbo.T1 where x=100
   go
   select a,b from T1 where x=500
   ```

   Sie können die Einstellung auch über den Eigenschaftendialog der Datenbank vornehmen, indem Sie dort auf der Seite OPTIONEN den Wert für die Option PARAMETRISIERUNG auf EINFACH oder ERZWUNGEN ändern. Denken Sie bitte in jedem Fall daran, dass das Einstellen der erzwungenen Parametrisierung einen ganz erheblichen Eingriff in die Abfrageoptimierung bedeutet. Verwenden Sie die Option bitte nur, sofern Sie erfahren genug sind, um die Auswirkungen zu verstehen und zu beurteilen. Sie können darüber nachdenken, die erzwungene Parametrisierung zu konfigurieren, falls Ihr Plancache ständig voll ist, der Zähler für die Verwendung Ihrer Ausführungspläne in den meisten Fällen 1 ist, die Kompilierungs-Rate hoch ist, und die Prozessorauslastung im Wesentlichen durch Kompilierungen hervorgerufen wird.

2. Verwenden der Prozedur `sp_executesql`. Die Prozedur `sp_executesql` erlaubt die Ausführung von dynamischem SQL mit Parametern. Sie haben die Prozedur etwas weiter oben in diesem Kapitel bereits einmal in Aktion gesehen. `sp_executesql` verarbeitet eine

variable Anzahl Parameter. Der erste Parameter ist hierbei stets der Text der SQL-Abfrage, wobei Sie für die verwendeten Parameter Platzhalter der Form `@name` angeben. Der zweite Parameter der Prozedur muss die im SQL-Text verwendeten Parameter deklarieren. Über den dritten und alle folgenden Parameter legen Sie die Werte für alle in der Abfrage verwendeten Parameter fest. Sie sehen etwas weiter unten ein Beispiel. Falls Sie eine Abfrage über `sp_executesql` ausführen, wird diese Abfrage stets parametrisiert, wobei diese Parametrisierung sich nach den im SQL-Text deklarierten Parametern richtet.

Im nachfolgenden Abschnitt sehen Sie noch, wie mit gespeicherten Prozeduren eine Parametrisierung veranlasst werden kann.

Da wir gerade etwas weiter oben festgestellt haben, dass unsere Abfrage

```
select y from T1 where x=100
```

nicht automatisch parametrisiert wird, würde dies also bedeuten, dass für jede Abfrage dieser Form ein separater Ausführungsplan erstellt wird. Wir entscheiden, dass wir an dieser Stelle eine Parametrisierung veranlassen möchten, und verwenden hierfür die Prozedur `sp_executesql`:

```
exec sp_executesql N'select y from T1 where x=@p'
                  ,N'@p int'
                  ,@p=100
```

Die Abfrage wird nun parametrisiert. Dadurch würde für alle weiteren Abfragen der obigen Form, bei denen jeweils nur der Wert des Parameters `@p` geändert wird, der parametrisierte Plan verwendet werden, was in diesem Fall sicherlich eine gute Idee ist, solange bis ...

Ja, bis zum Beispiel die folgenden Aktualisierungen an der Tabelle vorgenommen werden:

```
update T1 set x = 100
update top(1) T1 set x = 1000
```

Für eine Suche nach dem Wert der Spalte `x` ist nun ein einziger parametrisierter Plan ganz bestimmt nicht mehr die optimale Lösung. Nehmen wir an, der Plancache sei leer und die obige `sp_executesql`-Anweisung würde erneut ausgeführt. Der Ausführungsplan wird also für eine Abfrage nach `x=100` erstellt. Da die Spalte `x` bis auf eine Zeile in allen restlichen 99.999 Zeilen der Tabelle den Wert 100 aufweist, ist ein Table Scan hier der optimale Operator für die Suche. Der Optimierer entscheidet sich folglich dafür (Abbildung 9.24).

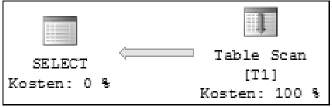


Abbildung 9.24:
Table Scan für die Abfrage aller Zeilen mit »x=100«

Soweit ist also alles in Ordnung. Wenn nun aber die folgende Abfrage

```
exec sp_executesql N'select y from T1 where x=@p'
                  ,N'@p int'
                  ,@p=1000
```

ausgeführt wird, wird der existierende (parametrisierte) Ausführungsplan gefunden und verwendet. Dadurch findet auch für diese Abfrage ein Table Scan statt – und das ist sicherlich nicht die optimale Lösung. Hier wäre ein Index Seek mit einem anschließenden RowId Lookup mit Sicherheit besser gewesen.

In Abbildung 9.25 sehen Sie den Ausführungsplan. Achten Sie bitte wieder auf den Unterschied zwischen geschätzter und tatsächlicher Zeilenanzahl. Er ist ein deutliches Indiz dafür, dass der Plan nicht optimal ist, weil der Optimierer falsche Kardinalitätsschätzungen zugrunde legt.

SELECT
Kosten: 0 %

Table Scan
[T1]
Kosten: 100 %

Table Scan
Scannt die Zeilen einer Tabelle.

Physischer Vorgang	Table Scan
Logischer Vorgang	Table Scan
Tatsächliche Anzahl von Zeilen	1
Geschätzte E/A-Kosten	0,412755
Geschätzte CPU-Kosten	0,110157
Anzahl von Ausführungen	1
Geschätzte Anzahl von Ausführungen	1
Geschätzte Operatorkosten	0,522912 (100 %)
Geschätzte Unterstrukturkosten	0,522912
Geschätzte Anzahl von Zeilen	99999

Abbildung 9.25: *Table Scan für das Prädikat »WHERE x=1000«*

Wenn wir im letzten Beispiel die Ausführung der beiden Abfragen vertauscht hätten, wäre für die Bedingung `WHERE x=1000` ein Plan mit einem Index Seek erstellt worden. Dieser Index Seek wäre dann auch für die Bedingung `WHERE x=100` verwendet worden, und das wäre natürlich ebenfalls nicht optimal.

Generell ist Parametrisierung keine gute Idee, falls die Verwendung unterschiedlicher Parameter für eine Abfrage zu einer sehr unterschiedlichen Anzahl von Zeilen im Ergebnis führt. In so einem Fall ist es ziemlich wahrscheinlich, dass unterschiedliche, also maßgeschneiderte, Abfragepläne besser sind.

Es kommt aber letztlich darauf an, wie häufig der nicht optimale Plan ausgeführt wird. Wenn der in Abbildung 9.25 gezeigte Table Scan für `x=1000` nur einmal täglich startet, die Abfrage nach `x=100` hingegen jede Stunde 1.000 Mal läuft, können Sie sicher sehr gut damit leben. Andernfalls müssen Sie eben Abfragen verwenden, die nicht parametrisiert sind. Wie Sie vorgehen können, um die Verwendung nicht optimaler Pläne im Plancache zu verhindern, erfahren Sie im folgenden Abschnitt.

Es gibt leider keine Möglichkeit, im Plancache gespeicherte Abfragen selektiv zu löschen, also etwa alle Pläne zu entfernen, die mehr als 1.000 Mal ausgeführt wurden und in denen für mindestens einen beteiligten Operator die Werte von geschätzter und tatsächlicher Zeilenanzahl um mehr als 70 Prozent voneinander abweichen.

9.4 Parameter Sniffing

Ich möchte diesen Abschnitt mit einem Beispiel beginnen, in dem eine gespeicherte Prozedur verwendet wird. Diese Prozedur soll alle Zeilen unserer Tabelle `T1` für einen einzelnen Wert `x` zurückliefern, also genau dasgleiche erledigen wie die durch `sp_executesql` ausgeführten Abfragen am Ende des vorherigen Abschnitts.

Der Code für die Prozedur ist denkbar einfach:

```
if (object_id('Proc1', 'P') is not null)
  drop procedure Proc1
go
create procedure Proc1(@p int) as
  select y from T1 where x=@p
go
```

Nach allem, was Sie nun über Parametrisierung wissen, ergeben sich die Fragen, was für ein Ausführungsplan (oder sind es vielleicht mehrere Pläne?) für diese Prozedur erstellt wird und wann die Erstellung des Ausführungsplans erfolgt.

Da für die Erstellung eines Plans Kardinalitätsschätzungen erforderlich sind, kann beim Anlegen der Prozedur über `CREATE PROCEDURE` sicherlich noch kein Plan erzeugt werden. Erst dann, wenn die Werte für die Parameter bekannt sind, ist die Erstellung eines Ausführungsplans möglich, denn erst zu diesem Zeitpunkt können anhand der konkreten Werte für die Parameter auch Kardinalitätsschätzungen durchgeführt werden. Deshalb wird ein Ausführungsplan beim ersten Aufruf der Prozedur erstellt. Für die Kardinalitätsschätzungen werden die bei diesem Aufruf übergebenen Parameter verwendet. SQL Server wendet hierbei eine Technik namens *Parameter Sniffing* an. Hierbei werden die Werte der an die Prozedur übergebenen Parameter verwendet und auf die in der Prozedur auszuführenden Anweisungen angewendet. Dadurch wird der erzeugte Plan auf der Basis dieser Parameterwerte für alle in der Prozedur enthaltenen Anweisungen erstellt.

Grundsätzlich ist Parameter Sniffing eine gute Sache, denn dadurch wird die Erstellung eines Plans für gespeicherte Prozeduren überhaupt erst möglich. Gespeicherte Prozeduren können natürlich erheblich komplexer sein als die in unserem Beispiel verwendete Prozedur. Wenn Sie sich vorstellen, dass eine solche Prozedur einige hundert Zeilen enthält, dann ist es sicherlich sinnvoll, für eine diese Prozedur einen wiederverwendbaren Plan zu haben und nicht bei jedem Aufruf erst einen Plan von Grund auf neu zu erstellen.

Grundsätzlich bringt diese Technik aber auch einige Probleme mit sich, die ich Ihnen im weiteren Verlauf dieses Abschnitts vorstellen möchte.

9.4.1 Probleme mit Parameter Sniffing

Für unsere oben erstellte Prozedur bedeutet Parameter Sniffing, dass beim Aufruf von

```
exec Proc1 1000
```

ein Ausführungsplan für

```
select y from T1 where x=1000
```

erstellt wird. Dieser Plan wird dann in der Folge stets verwendet, wenn die Prozedur aufgerufen wird, auch mit anderen Parametern, also etwa so:

```
exec Proc1 100
```

Damit haben wir natürlich dasselbe Problem wie am Ende von Abschnitt 9.3.3 bei der Verwendung von `sp_executesql`. Grundsätzlich kann für eine gespeicherte Prozedur nur ein gespeicherter Plan existieren, und dieser Plan ist immer parametrisiert. Wenn dieser Plan bei einem Aufruf mit untypischen Parametern erstellt wird, ist der Plan für alle Aufrufe mit »normalen« Parametern ineffizient. Etwas später werden Sie sehen, wie Probleme dieser Art gelöst werden können. Zuvor aber müssen wir noch über eine weitere Besonderheit in Bezug auf Parameter Sniffing sprechen, die oft übersehen wird.

Änderung von Parameterwerten in gespeicherten Prozeduren

Das »Ausschnüffeln« von Parametern funktioniert nur in erster Instanz. Dies bedeutet, dass der an eine Prozedur beim ersten Aufruf übergebene Parameterwert die Erstellung des Ausführungsplans bestimmt. Wenn Sie diesen Wert vor seiner Verwendung in Abfragen verändern, hat der geänderte Wert keinerlei Einfluss auf die Erstellung des Plans. Stattdessen wird der originale (also der übergebene) Wert verwendet.

Ich möchte Ihnen dieses Verhalten an einem einfachen Beispiel verdeutlichen, das ich in der Praxis schon häufig in dieser Form gesehen habe. Stellen Sie sich vor, es soll eine Suche nach einer Zeichenkette durchgeführt werden, wobei auch Platzhalter zulässig sind. Als Platzhalterzeichen hat sich allgemein das Zeichen * durchgesetzt. Die Benutzer sollen daher an der Oberfläche das Zeichen * als Platzhalter eingeben dürfen. In SQL-Abfragen kann dieses Zeichen als Platzhalter nicht verwendet werden, dort muss stattdessen das Prozentzeichen %, angegeben werden. Folglich muss irgendwo im Code aus * ein % werden, zum Beispiel in einer eigens für die Suche geschriebenen gespeicherten Prozedur:

```
use AdventureWorks2008;
if (object_id('SuchePersonen', 'P') is not null)
  drop procedure SuchePersonen
go
create procedure SuchePersonen(@n nvarchar(80)) as
 begin
   set @n = replace(@n, '*', '%') + '%'
   select LastName,FirstName,ModifiedDate
     from Person.Person
     where LastName like @n
 end
go
```

In der Prozedur erfolgt also die Umwandlung der Platzhalterzeichen in das SQL-Format. Nehmen wir nun an, dass nach der Erstellung der Prozedur alle Personen angezeigt werden sollen. Hierzu wird die Prozedur folgendermaßen aufgerufen:

```
exec SuchePersonen '*'
```

Tatsächlich werden alle Personen zurückgegeben, weil innerhalb der Prozedur letztlich eine Filterung nach `LastName LIKE '%%'` durchgeführt wird (die in Wirklichkeit keine Filterung ist). Für diese Suche ist mit Sicherheit ein Clustered Index Scan der optimale Operator. Der Ausführungsplan (Abbildung 9.26) zeigt jedoch einen Index Seek mit einer Schlüsselsuche (für die Spalte `ModifiedDate`).

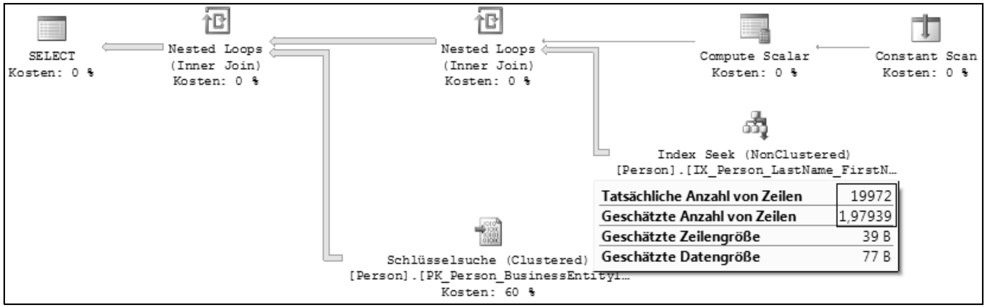


Abbildung 9.26: *Nicht optimaler Ausführungsplan durch Parameter Sniffing*

Sie sehen deutlich, dass die geschätzte Anzahl Zeilen ganz erheblich von der tatsächlichen Zeilenanzahl abweicht. Der übergebene Wert für den Parameter `@n` ist * und mit diesem Wert wird der Plan für die Abfrage erstellt. Die Tatsache, dass der Parameterwert innerhalb der Prozedur zuvor noch verändert wird, spielt überhaupt keine Rolle. Wenn Sie über dieses Verhalten nachdenken, dann ist dies auch durchaus logisch, denn der Plan für die Ausführung der Prozedur muss ja vor der eigentlichen Ausführung erstellt werden und der Wert von `@n` wird erst während der Ausführung verändert. Der Plan wird also für das Prädikat `WHERE LastName LIKE '*'` erstellt, und dafür werden 1,97939 Zeilen erwartet. Für diese Zeilenanzahl ist ein Index Seek sicherlich die optimale Wahl, für die tatsächlich zurückgelieferten 19.972 Zeilen ganz bestimmt nicht. Die Abfrage hat insgesamt 60.022 logische Lesevorgänge benötigt.

Sie können sich leicht davon überzeugen, dass ein Clustered Index Seek hier die bessere Wahl gewesen wäre, indem Sie die folgende Abfrage ausführen:

```
dbcc freeproccache
exec SuchePersonen '%'
```

Diese Abfrage gibt ebenfalls alle Zeilen zurück, nur wird diesmal der Parameterwert % für die Erstellung des Ausführungsplans verwendet. Abbildung 9.27 zeigt den Ausführungsplan.

SELECT
Kosten: 0 %

Clustered Index Scan (Clustered)
[Person].[PK_Person_BusinessEntityI…

Tatsächliche Anzahl von Zeilen	19972
Geschätzte Anzahl von Zeilen	19972
Geschätzte Zeilengröße	43 B
Geschätzte Datengröße	839 KB

Abbildung 9.27: *Optimaler Abfrageplan für die Rückgabe aller Zeilen*

Diesmal stimmen geschätzte und tatsächliche Zeilenanzahl überein und es sind lediglich 3.825 logische Lesevorgänge erforderlich. Dies sind nur kanpp 6 Prozent des vorigen Ergebnisses.

Das Problem ist allerdings etwas tiefergehender. Wenn Sie ein wenig darüber nachdenken, so werden Sie herausfinden, dass es in diesem Fall keine typischen Parameter gibt, für die ein allgemein gültiger Plan erstellt werden kann. Je nachdem, welche Zeichenkette für die Suche übergeben wird, ist der optimale Plan unterschiedlich; es gibt also in diesem Fall nicht den einen optimalen Plan. Dies ist einfach deshalb so, weil die Prozedur – je nach Suchkriterium – zwischen 0 und 19.972 Zeilen zurückliefert. Sie werden im weiteren Verlauf dieses Kapitels noch sehen, wie dieses Problem umgangen werden kann.

Bedingte Ausführung von Anweisungen in gespeicherten Prozeduren

Das gerade präsentierte Beispiel kann Ihnen etwas modifiziert zum Beispiel auch so begegnen:

```
use AdventureWorks2008;
if (object_id('SucheAdressen', 'P') is not null)
  drop procedure SucheAdressen
go
create procedure SucheAdressen(@city nvarchar(30), @postalCode nvarchar(15)) as
 begin
   if (@city is not null)
     select PostalCode, City, AddressLine1, AddressLine2, ModifiedDate
       from Person.Address
      where City like @city
   else if (@postalCode is not null)
     select PostalCode, City, AddressLine1, AddressLine2, ModifiedDate
       from Person.Address
      where PostalCode like @postalCode
   else
     select PostalCode, City, AddressLine1, AddressLine2, ModifiedDate
       from Person.Address
 end
go
```

Die Prozedur `SucheAdressen` führt – wie ihr Name verrät – eine Suche nach Adresseinträgen in der Tabelle `Person.Address` durch. Die Art der Suche ist hierbei abhängig von den übergebenen Parametern. Wird für den Parameter `@city` ein Wert übergeben, der nicht `NULL` ist, dann führt die Prozedur eine Suche nach dem Namen der Stadt durch. Gleiches gilt für den Parameter `@postalCode`, durch den eine Suche nach der Postleitzahl durchgeführt wird. Schließlich gibt es auch noch die Möglichkeit, dass keiner der beiden Parameter von `NULL` verschieden ist. In diesem Fall werden einfach alle Zeilen der Tabelle zurückgegeben. Die Werte der übergebenen Parameter werden innerhalb der Prozedur nicht geändert. Trotzdem gibt es ein ähnlich geartetes Problem wie im vorherigen Abschnitt.

Stellen Sie sich einmal vor, die Prozedur würde folgendermaßen ausgeführt:

```
exec SucheAdressen 'Dulu%', null
```

Wir durchsuchen also die Adress-Tabelle nach Einträgen, bei denen der Name der Stadt mit »Dulu« beginnt. Der Aufruf der Prozedur führt dazu, dass zunächst ein Ausführungsplan erstellt wird. Da der Wert für den Parameter `@city` angegeben wurde, wird dieser Zweig der Prozedur ausgeführt:

```
if (@city is not null)
  select PostalCode, City, AddressLine1, AddressLine2, ModifiedDate
    from Person.Address
   where City like @city
```

Deshalb findet eine Optimierung hinsichtlich des Parameterwertes »Dulu%« für den Mustervergleich durch `LIKE` statt. Für diesen Vergleich enthält die Tabelle zwei Zeilen, weshalb völlig korrekt ein Index Scan mit einer anschließenden Schlüsselsuche ausgeführt wird (Abbildung 9.28).

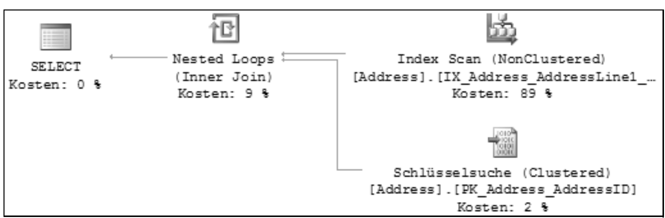


Abbildung 9.28: *Ausführungsplan für die Suche nach »City like 'Dulu%'«*

Eine Suche im Indexbaum kann hier nicht verwendet werden, da die Spalte `City` nicht an erster Stelle im Index steht. Der Optimierer entscheidet sich für einen Index Scan, weil der Index alle erforderlichen Spalten enthält (er ist abdeckend), und dadurch weniger Lesevorgänge erforderlich sind als bei einem Clustered Index Scan.

Die geschätzte Zeilenanzahl ist ca. 1,9; die tatsächliche Zeilenanzahl ist 2 – soweit ist also alles in Ordnung. Sie wissen bereits, dass der erstellte Plan in der Folge etwa für eine Suche nach allen Städten, die mit dem Buchstaben »A« beginnen, sehr wahrscheinlich nicht optimal ist, aber dieser Umstand soll uns hier nicht weiter interessieren.

Wir wollen uns stattdessen der folgenden Frage zuwenden: Wenn beim ersten Aufruf der Prozedur ein Ausführungsplan für die Prozedur erstellt wird, welcher Plan wird dann für die anderen beiden `IF`-Zweige erstellt, die also beim ersten Aufruf nicht ausgeführt werden? Was passiert, wenn beim zweiten Aufruf der Prozedur nach der Postleitzahl gesucht wird, zum Beispiel so:

```
exec SucheAdressen null, '558%'
```

Für den bei diesem Aufruf entscheidenden `IF`-Zweig muss ja auch beim Erstellen des Ausführungsplans bereits ein entsprechender Plan erzeugt worden sein. Da zu diesem Zeitpunkt aber der Wert der Variablen `@postalCode` nicht bekannt gewesen ist, muss der Optimierer irgendwelche Annahmen treffen – die mehr oder weniger realitätsnah sind.

Diese Annahmen werden letztlich aus den beteiligten Operatoren abgeleitet. In unserem Beispiel ist dies der LIKE-Operator. Der Optimierer kann nicht anders, als anhand dieses Operators eine Kardinalitätsschätzung vorzunehmen, die auf Erfahrungswerten und existierenden Statistiken beruht. In Abbildung 9.29 sehen Sie den Ausführungsplan für den Prozeduraufruf mit einer Suche nach der Postleitzahl.

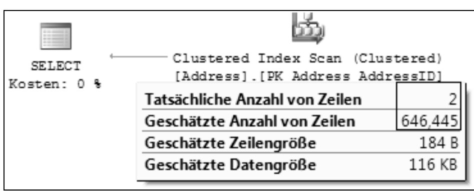


Abbildung 9.29:
Clustered Index Scan für eine Suche nach zwei Zeilen

Es werden ca. 646 Zeilen erwartet, und deshalb wird ein Clustered Index Scan ausgeführt. Für die tatsächlich gelieferten zwei Zeilen wäre ein Index Scan, so wie in Abbildung 9.28 zu sehen, sicherlich die bessere Wahl gewesen, aber der gespeicherte Ausführungsplan ist für diesen Fall eben nicht optimal. Allein der LIKE-Operator hat die Auswahl des Operators bestimmt. Der Optimierer legt für unseren Fall eine Selektivität dieses Operators von 646,445/19614 – 3 % zugrunde: ein Mittelwert, der aus der bestehenden Statistik ermittelt wird. Etwas weiter unten werden wir auf diese Art der Schätzung noch einmal zurückkommen.

Sie können sich davon überzeugen, dass ein anderer Plan gewählt würde, wenn Sie den zweiten Prozeduraufruf zuerst ausführten und zuvor den Plancache leerten:

```
dbcc freeproccache
exec SucheAdressen null, '558%'
```

Der Ausführungsplan sieht von seiner Struktur so aus, wie in Abbildung 9.28 zu sehen.

Das Dilemma ist also, dass immer nur für einen IF-Zweig ein optimaler Plan erstellt werden kann.

Das Problem kann Ihnen in leicht abgewandelter Form auch so begegnen:

```
if (object_id('SucheAdressen', 'P') is not null)
  drop procedure SucheAdressen
go
create procedure SucheAdressen(@city nvarchar(30), @postalCode nvarchar(15)) as
 begin
   select PostalCode, City, AddressLine1, AddressLine2, ModifiedDate
     from Person.Address
    where ((@city is null) or (City like @city))
      and ((@postalCode is null) or (PostalCode like @postalCode))
 end
go
```

Oder etwa so:

```
if (object_id('SucheAdressen', 'P') is not null)
  drop procedure SucheAdressen
go
create procedure SucheAdressen(@city nvarchar(30), @postalCode nvarchar(15)) as
 begin
   select PostalCode, City, AddressLine1, AddressLine2, ModifiedDate
     from Person.Address
    where City like isnull(@city, '##')
       or PostalCode like isnull(@postalCode,'##')
 end
go
```

Hier werden die einzelnen IF-Zweige sozusagen in das Prädikat der SELECT-Anweisung verlagert, und dadurch wird der Code besonders kompakt – eine scheinbar clevere Lösung.

Die Lösung ist aber eben nur scheinbar clever, weil auch hier wieder der erste Prozeduraufruf zur Planerstellung führt. An der Tatsache, dass die bei diesem Aufruf verwendeten Parameter die Optimierung bestimmen, ändert sich dadurch natürlich nichts. In der zweiten Lösung kommt noch hinzu, dass die übergebenen Parameter nicht direkt verwendet werden, sondern durch eine Funktion verändert werden. Dies verhindert das »Ausschnüffeln« der Parameterwerte und die Erstellung eines optimalen Plans.

Eine mögliche Lösung dieses Problems ist die Verwendung von separaten Prozeduren für jeden IF-Zweig. Für jede dieser Prozeduren wird ein separater Plan erstellt, der jeweils auf der Basis der beim ersten Aufruf übergebenen Parameter optimiert wird. Weitere Lösungen finden Sie etwas weiter unten.

Verwenden lokaler Variablen

Eine weitere Besonderheit mit Parameter Sniffing ergibt sich bei der Verwendung lokaler Variablen. Stellen Sie sich bitte vor, dass Sie die an eine Prozedur übergebenen Parameter für die Festlegung von lokalen Variablen verwenden und diese lokalen Variablen in einer Abfrage benutzen, also etwa so:

```
if (object_id('SucheAdressen', 'P') is not null)
  drop procedure SucheAdressen
go
create procedure SucheAdressen(@city nvarchar(30)) as
 begin
   declare @citySearch nvarchar(80)
   set @citySearch = replace(@city, '*', '%') + '%'
   select PostalCode, City, AddressLine1, AddressLine2, ModifiedDate
     from Person.Address
    where City like @citySearch
 end
go
```

Innerhalb der Prozedur wird eine lokale Variable deklariert, deren Wert anhand des übergebenen Parameters festgelegt wird. Diese lokale Variable wird dann in der Abfrage verwendet. Wenn nun der folgende Aufruf

```
exec SucheAdressen 'Kingsport'
```

der Prozedur ausgeführt wird, ergibt sich die Frage, welcher Wert von `@citySearch` für die Erstellung des Plans herangezogen wird? Ein Parameter Sniffing kann hier sicherlich nicht stattfinden, denn der übergebene Parameter `@city` wird ja in der Abfrage überhaupt nicht verwendet. Auf welcher Basis findet hier also eine Kardinalitätsschätzung statt?

Es funktioniert genau so, wie im vorangegangenen Abschnitt bei der Verwendung bedingter Ausführungen, also unterschiedlicher `IF`-Zweige, bereits erklärt. Prägen Sie sich hierzu bitte Folgendes ein:

Falls der Wert einer verwendeten Variable zur Übersetzungszeit der Abfrage nicht bekannt ist, verwendet der Optimierer Annahmen über die Kardinalität, wobei er die bestehenden Statistiken und die in der Abfrage verwendeten Operatoren zur Beurteilung heranzieht.

Wenn wir den obigen Prozeduraufruf, der eine Suche nach der Stadt »Kingsport« durchführt, ausführen, ergibt sich der in Abbildung 9.30 gezeigte Ausführungsplan.

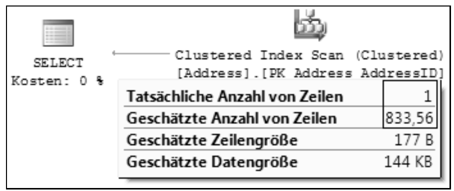


Abbildung 9.30: *Ausführungsplan für die Suche nach »City LIKE 'Kingsport%'«*

Die vom Optimierer vorgenommene Kardinalitätsschätzung geht von ca. 833 Zeilen aus. Deshalb wird ein Clustered Index Scan ausgeführt. Tatsächlich wird aber nur eine Zeile geliefert; ein Index Scan wäre hier also mit Sicherheit besser gewesen. Der Optimierer kann die Kardinalität eben nur grob einschätzen, weil der Wert der Variablen `@citySearch` zum Zeitpunkt der Planerstellung unbekannt ist.

Denken Sie bitte an dieses Verhalten, wenn Sie lokale Variablen in T-SQL-Skripten verwenden. Das gerade beschriebene Phänomen ist nicht nur auf gespeicherte Prozeduren beschränkt, sondern gilt ganz allgemein in beliebigen T-SQL-Skripten!

Schauen Sie sich bitte einmal das folgende Skript an, welches zwei Abfragen enthält:

```
use QueryTest;
select * from Person where Id < 0
go

declare @id int
set @id = 0
select * from Person where Id < @id
```

Wir verwenden für die Abfragen die etwas weiter oben in unserer Datenbank `QueryTest` angelegte Tabelle `Person`. Beide Abfragen sind logisch äquivalent, liefern also dasselbe Ergebnis (nämlich keine einzige Zeile). Und doch sehen die Ausführungspläne für beide Abfragen komplett anders aus (Abbildung 9.31).

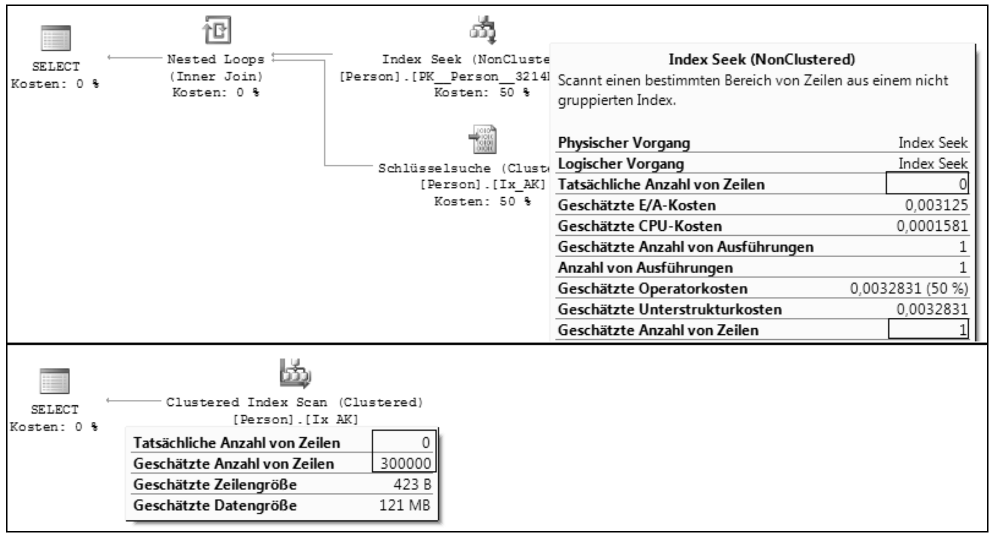


Abbildung 9.31: *Zwei unterschiedliche Pläne für dieselbe logische Abfrage. Der gruppierte Index existiert für die Spalten »(Altersklasse, Gehalt)«*

Der obere Teil in Abbildung 9.31 zeigt hierbei den Ausführungsplan ohne Verwendung einer lokalen Variablen, der untere Teil den Plan der Abfrage mit Verwendung der Variablen `@id`.

Der obere Plan ist absolut in Ordnung. Im unteren Plan erkennen Sie deutlich, dass die Schätzung der Zeilenanzahl bei der Verwendung einer lokalen Variablen ganz erheblich von der Realität abweicht. Woher kommen die geschätzten 300.000 Zeilen?

Der Optimierer legt für den Operator `<` eine allgemeine Selektivität von 30 Prozent zugrunde, wenn aus der Vergleichsbedingung der Wert für den Vergleich nicht ermittelt werden kann. Dies ist in unserem Beispiel der Fall. Unsere Tabelle `Person` hat 1.000.000 Zeilen, und das ergibt dann geschätzte 300.000 Zeilen, also 30 Prozent der Zeilenanzahl für einen `<`-Vergleich mit einem unbekannten Wert. Für die anderen Vergleichsoperatoren werden ähnliche Annahmen getroffen. So ist zum Beispiel die allgemeine Selektivität für den `<`-Operator ebenfalls 30 Prozent. Für den exakten Vergleich über den =-Operator wird die geschätzte Zeilenanzahl aus der Dichte der zugehörigen Statistik (siehe Abschnitt 9.2) multipliziert mit der Zeilenanzahl abgeleitet. Bei Verwendung des `LIKE`-Operators gibt es in diesem Fall keine allgemeine Festlegung der Selektivität. Die Statistiken für Zeichenketten sind etwas detaillierter als reine »Zahlen«-Statistiken, und die geschätzte Selektivität hängt in diesem Fall von der entnommenen Stichprobe ab.

Der Unterschied in den benötigten Lesevorgängen für die beiden Abfragen ist hierbei sehr ausgeprägt: Die Abfrage unter Verwendung des Index Seek benötigt drei logische Lesevorgänge, gegenüber 55.750 erforderlichen logischen Lesevorgängen beim Clustered Index Seek! Dieser Unterschied schlägt sich auch markant in der Ausführungszeit nieder. Also merken Sie sich bitte die folgende einfache Regel:

Die Verwendung von lokalen Variablen in »reinen« T-SQL-Skripten ist keine gute Idee.

9.4.2 Lösung von Parameter Sniffing-Problemen

Nachdem Sie nun wissen, welche Stolperfallen auf Sie lauern, möchte ich Ihnen im Folgenden mögliche Lösungen für diese Fallen präsentieren.

Einsatz von dynamischem SQL

Dynamisches SQL ist allgemein verpönt – und in den meisten Fällen auch zu Recht. Mit Bedacht eingesetzt, kann dynamisches SQL aber durchaus sinnvoll sein, wie das folgende Beispiel beweist.

Schauen Sie sich bitte einmal den folgenden geänderten Code unserer gespeicherten Prozedur `SucheAdressen` an:

```
if (object_id('SucheAdressen', 'P') is not null)
  drop procedure SucheAdressen
go
create procedure SucheAdressen(@city nvarchar(30), @postalCode nvarchar(15)) as
 begin
   declare @cmd nvarchar(300)
   set @cmd = 'select PostalCode, City
                    ,AddressLine1, AddressLine2, ModifiedDate
                from Person.Address where (1=1)'
   if (@city is not null)
     set @cmd = @cmd + ' and (City like @city)'
   if (@postalCode is not null)
     set @cmd = @cmd + ' and (PostalCode like @postalCode)'
     print @cmd
   exec sp_executesql @cmd, N'@city nvarchar(30), @postalCode nvarchar(15)'
                    ,@city = @city
                    ,@postalCode = @postalCode
 end
go
```

Die Prozedur verwendet nun dynamisches SQL. Für jede Kombination von Parametern wird ein eigener Aufruf von `sp_executesql` ausgeführt, der jeweils speziell an diese Parameter angepasst wird. Erst beim Aufruf von `sp_executesql` wird hierbei ein Plan für die

entsprechende Anweisung erstellt, also nicht bereits beim ersten Aufruf der Prozedur SucheAdressen. Diese Verfahrensweise löst das Problem, dass für unterschiedliche Kombinationen von Parametern unterschiedliche Ausführungspläne optimal sind.

Die folgenden vier Prozeduraufrufe bekommen nun alle einen eigenen Ausführungsplan:

```
exec SucheAdresse null, '5%'
exec SucheAdresse 'Dulu%', null
exec SucheAdresse 'Berlin%', '909%'
exec SucheAdresse null, null
```

Das Problem, dass unterschiedliche Parameterwerte verschiedene optimale Pläne benötigen, wird jedoch nicht gelöst. sp_executesql erstellt jeweils einen parametrisierten Plan für die beim ersten entsprechenden Aufruf übergebenen Parameterwerte. Damit bleibt das Problem bestehen, dass die Erstellung dieses Plans mit untypischen Parameterwerten einen nicht optimalen Plan für alle anderen Parameterwerte darstellt.

Verwenden des Abfragehinweises OPTIMIZE FOR

Sofern Sie eine gespeicherte Prozedur erstellen, haben Sie keine Kontrolle darüber, mit welchen Parameterwerten diese Prozedur das erste Mal von einer Anwendung ausgeführt wird, und daher auch keinen Einfluss auf den generierten Ausführungsplan – oder etwa doch?

Sie werden es sicher schon ahnen. – Es gibt tatsächlich Möglichkeiten, die Planerstellung zu beeinflussen – und zwar direkt im SQL-Code einer Abfrage. Zu diesem Zweck können Sie sogenannte Abfragehinweise verwenden, um den Optimierer in eine bestimmte Richtung zu lenken.

Ich bin kein Freund von Abfragehinweisen, da ich denke, dass der Optimierer am besten selber entscheiden kann, welcher Plan erstellt wird – und in den meisten Fällen benötigt der Optimierer auch keine weitere Unterstützung. Abfragehinweise, welche die Planerstellung beeinflussen, sind einigermaßen gefährlich, denn sie schränken die Möglichkeiten der Planerstellung ein und sollten daher mit entsprechender Vorsicht verwendet werden. Denken Sie bei der Verwendung von Abfragehinweisen bitte stets daran, dass eine Änderung an den Daten oder Strukturen jederzeit dazu führen kann, dass ein ehemals sinnvoller Hinweis nun kontraproduktiv wird. Im Allgemeinen ist es sehr schwierig, Probleme aufzudecken, die auf ungeeignete oder veraltete Abfragehinweise zurückzuführen sind. Die im Folgenden vorgestellten Abfragehinweise dienen im Wesentlichen dazu, Parameter Sniffing auszuschalten bzw. die im Zusammenhang mit Parameter Sniffing auftretenden Probleme zu umgehen.

Kommen wir also zu einer ersten Möglichkeit, die Planerstellung mit einem Abfragehinweis zu beeinflussen. Falls Sie ein typisches Muster für einen Prozeduraufruf kennen, können Sie durch den Hinweis OPTIMIZE FOR die Parameterwerte angeben, für welche der Optimierer den Plan erstellen soll. Schauen Sie sich noch einmal unsere Prozedur SucheAdressen an, deren Abfrage hier einen entsprechenden Abfragehinweis enthält:

```
if (object_id('SucheAdressen', 'P') is not null)
  drop procedure SucheAdressen
go
create procedure SucheAdressen(@city nvarchar(30), @postalCode nvarchar(15)) as
 begin
   select PostalCode, City, AddressLine1, AddressLine2, ModifiedDate
     from Person.Address
    where ((@city is null) or (City like @city))
      and ((@postalCode is null) or (PostalCode like @postalCode))
     option (optimize for (@city='Dulu%', @postalCode='86541%'))
 end
go
```

Über die Klausel `OPTION (OPTIMIZE FOR(...))` geben Sie die Werte für die verwendeten Parameter an, die für die Planerstellung verwendet werden sollen. Wenn Sie die Prozedur so erzeugen, ist es für die Planerstellung unerheblich, welche Parameterwerte beim ersten Aufruf übergeben werden; es werden stets die im Abfragehinweis verwendeten Parameter für die Optimierung verwendet. Dadurch vermeiden Sie eine Situation, in welcher der erste Aufruf einer Prozedur mit untypischen Parameterwerten die Erstellung eines Ausführungsplans bewirkt, der für die meisten folgenden Aufrufe nicht optimal ist.

Egal, mit welchen Parameterwerten die Prozedur nun aufgerufen wird, der Plan wird immer für die im Optimierungshinweis angegebenen Parameterwerte erstellt. Falls Sie einen allgemeinen Plan erzeugen möchten, der nicht auf der Basis konkreter Parameterwerte generiert wird, können Sie für jeden Parameterwert auch den reservierten Bezeichner `UNKNOWN` verwenden. Der Abfragehinweis in unserer geänderten Prozedur `SucheAdressen` kann also auch so aussehen:

```
option (optimize for (@city unknown, @postalCode unknown))
```

In diesem Fall findet die Optimierung auf der Basis einer Kardinalitätsschätzung statt, die nur von den verwendeten Operatoren und den Statistiken bestimmt wird. Dies ist also genau dasselbe Verfahren wie bei der Verwendung lokaler Variablen in T-SQL-Skripten, das weiter oben erläutert wurde. In unserer Prozedur `SucheAdressen` führt dies stets zu einem Clustered Index Scan, weil insgesamt ca. 2.089 Ergebniszeilen erwartet werden. Dies gilt auch dann, wenn die Prozedur mit sehr selektiven Parametern aufgerufen wird, die zum Beispiel nur eine Zeile zurückliefern, und für die deshalb eine Suche im Index (in unserem Fall ein Index Scan wegen des Fehlens eines geeigneten Index für die Suche im Index-Baum) günstiger gewesen wäre.

Mit `OPTIMIZE FOR` können Sie den Optimierer also veranlassen, einen Plan zu erzeugen, der für bestimmte Parameterwerte optimal ist. Falls es einen solchen Plan nicht gibt (etwa weil keine typischen Parameterwerte existieren), hilft Ihnen `OPTIMIZE FOR` natürlich nicht wirklich. Das Problem ist auch hier wieder, dass Sie sich für den einen Plan entscheiden müssen, der alle Belange abdeckt – und dies ist in einigen Fällen schlichtweg nicht möglich.

Verwenden von RECOMPILE

Wie gerade gesagt, ist die Tatsache, dass für eine parametrisierte Abfrage oder auch eine gespeicherte Prozedur nur ein Plan existieren kann, oft störend, wenn die parametrisierte Abfrage mit sehr unterschiedlichen Parameterwerten ausgeführt wird.. In vielen Fällen wären für unterschiedliche Werte der Parameter letztlich separate Pläne besser als ein einziger Plan für alle Parameterwerte. Um so etwas zu erreichen, kann der Abfragehinweis `RECOMPILE` in verschiedenen Varianten verwendet werden. Wie üblich, möchte ich Ihnen auch diesmal ein Beispiel präsentieren, um diese unterschiedlichen Varianten zu erklären.

Wir verwenden auch hier wieder die gespeicherte Prozedur `SucheAdressen`, zunächst in der schon bekannten Form:

```
if (object_id('SucheAdressen', 'P') is not null)
  drop procedure SucheAdressen
go
create procedure SucheAdressen(@city nvarchar(30), @postalCode nvarchar(15)) as
 begin
   select PostalCode, City, AddressLine1, AddressLine2, ModifiedDate
     from Person.Address
    where ((@city is null) or (City like @city))
      and ((@postalCode is null) or (PostalCode like @postalCode))
 end
go
```

Noch einmal zur Wiederholung: Je nachdem, welcher der beiden folgenden Aufrufe zuerst erfolgt, wird der für diese Prozedur erstellte Plan entweder einen Clustered Index Scan oder einen Index Scan mit anschließender Schlüsselsuche im gruppierten Index beinhalten:

```
-- Clustered Index Scan
exec SucheAdressen null, null

-- Index Scan mit Schlüsselsuche
exec SucheAdressen 'Kingsport%', '37%'
```

Je nach erstelltem Plan wird entweder nur der erste oder nur der zweite Aufruf auf der Basis eines optimalen Plans ausgeführt.

Der `RECOMPILE`-Hinweis kann verwendet werden, um dafür zu sorgen, dass stets ein möglichst optimaler Plan für die Ausführung erstellt wird. Sie können den `RECOMPILE`-Hinweis prinzipiell auf vier Arten angeben:

RECOMPILE bei der Ausführung einer gespeicherten Prozedur oder Abfrage. Sie können bei der Ausführung einer gespeicherten Prozedur die Option `RECOMPILE` als Zusatz angeben. Ein entsprechender Aufruf sieht dann so aus:

```
exec SucheAdressen 'Kingsport%', '37%' with recompile
```

Dadurch wird ein eventuell existierender Ausführungsplan nicht verwendet. Stattdessen wird ein neuer Plan erstellt, der nur für diesen Aufruf verwendet wird, also auch nicht in den Plancache gelangt. Ein bereits existierender Ausführungsplan wird dabei nicht verändert.

RECOMPILE bei der Erstellung einer gespeicherten Prozedur Es ist auch möglich, dass Sie die Option `RECOMPILE` bereits bei der Erstellung einer gespeicherten Prozedur, also als Option für die Anweisung `CREATE PROCEDURE` verwenden:

```
create procedure SucheAdressen(@city nvarchar(30), @postalCode nvarchar(15))
         with recompile as
 begin
   ...
 end
go
```

In diesem Fall wird für die Prozedur niemals ein Ausführungsplan gespeichert. Jede Ausführung der Prozedur erfordert daher zunächst die Erstellung eines Ausführungsplans, der dann exakt auf die verwendeten Parameter abgestimmt ist. Dieser Ausführungsplan wird nicht im Plancache gespeichert, ist also nur für den einen Aufruf gültig.

Gezieltes RECOMPILE für eine Anweisung innerhalb einer gespeicherten Prozedur Seit SQL Server 2005 können Sie den `RECOMPILE`-Hinweis auch für einzelne Abfragen innerhalb einer gespeicherten Prozedur verwenden. Dadurch wird nicht bei jeder Ausführung der Prozedur ein komplett neuer Plan für die gesamte Prozedur erstellt. Vielmehr kann ein existierender Plan wiederverwendet werden, und nur die durch `RECOMPILE` gekennzeichneten Anweisungen werden für die Ausführung kompiliert. Dadurch wird insbesondere bei größeren Prozeduren, die viele Anweisungen enthalten, Zeit für die Übersetzung eingespart. Prinzipiell sieht die Syntax so aus:

```
create procedure SucheAdressen(@city nvarchar(30), @postalCode nvarchar(15)) as
 begin
   select PostalCode, City, AddressLine1, AddressLine2, ModifiedDate
     from Person.Address
    where ((@city is null) or (City like @city))
      and ((@postalCode is null) or (PostalCode like @postalCode))
    option( recompile)
 end
go
```

Da unsere Beispielprozedur nur eine einzige Abfrage enthält, hat die Verwendung von `OPTION RECOMPILE` hier letztlich die gleiche Wirkung wie das Anlegen der Prozedur mit der Klausel `WITH RECOMPILE`.

RECOMPILE für eine beliebige Abfrage Genau so, wie Sie den `RECOMPILE`-Abfragehinweis in einzelnen Anweisungen einer gespeicherten Prozedur verwenden können, ist es möglich, jede beliebige Abfrage mit diesem Abfragehinweis zu versehen. Dadurch wird ein eventuell existierender Ausführungsplan ignoriert, und für die entsprechende Abfrage wird ein eigener Ausführungsplan erstellt. Dieser erstellte Ausführungsplan ist wiederum nur für die eine Abfrage gültig und wird nicht im Plancache gespeichert.

Sofern Sie die obigen Punkte aufmerksam gelesen haben, haben Sie sicherlich bereits festgestellt, dass der Name RECOMPILE nicht besonders glücklich gewählt ist.

Ein Re-Compile bezeichnet üblicherweise eine Neu-Übersetzung eines bestehenden Ausführungsplans, etwa auf Grund von veralteten Statistiken oder Schemaänderungen, wobei der neue Plan den alten Plan im Plancache ersetzt. Diese Aktion wird bei Angabe der Option RECOMPILE nicht ausgeführt. Ein Re-Compile in diesem Sinne ist in Wirklichkeit ein Compile, insbesondere deshalb:

- Es ist nicht ausschlaggebend, ob bereits ein entsprechender Plan vorhanden ist. Bei Angabe von RECOMPILE findet stets eine Planerstellung, also ein Compile statt.
- Der durch die Option RECOMPILE erstellte Plan wird nicht im Plancache gespeichert, sondern nur für die einmalige Ausführung der Abfrage verwendet.
- Der Name hätte also besser COMPILE lauten sollen

Dem Vorteil, dass die Verwendung der Option RECOMPILE stets zu einem möglichst optimalen Plan führt, steht natürlich der Nachteil gegenüber, dass die Wiederverwendung eines existierenden Plans verhindert wird. Dadurch wird zwar der Plancache in einigen Fällen entlastet, Sie bezahlen dies jedoch mit einer erhöhten CPU-Belastung und möglicherweise auch mit einer längeren Ausführungszeit. Die Ausführungszeit verlängert sich immer dann, wenn ein existierender optimaler Plan nicht verwendet wird. In diesem Fall kostet die durch eine RECOMPILE-Option angestoßene Erstellung des Ausführungsplans unnötig Zeit.

Sie müssen also sehr genau abwägen, ob sich ein Einsatz von RECOMPILE lohnt. Auf jeden Fall sollten Sie nicht den rigorosen Ansatz wählen und RECOMPILE in einem OLTP-System »auf breiter Front« einsetzen. Beobachten Sie mit Hilfe des Profilers oder des Windows Systemmonitors Ihre Kompilierungs- und Re-Kompilierungsrate, um festzustellen, ob an dieser Stelle Probleme auftreten (siehe auch Kapitel 4). Verwenden Sie RECOMPILE nur gezielt, um die Leistung problematischer Abfragen zu verbessern.

Arbeiten mit Planhinweislisten (Plan Guides)

Alle bislang auf der Basis von Abfragehinweisen präsentierten Lösungen haben gemein, dass Sie gezielt für konkrete Abfragen eingesetzt werden. Diese Verfahrensweise hat prinzipiell zwei Nachteile:

- Sie werden nicht in allen Fällen die erforderlichen SQL-Anweisungen anpassen können. Falls Sie zum Beispiel eine gekaufte Software verwenden, die Abfragen gegen eine Datenbank ausführt, haben Sie sehr wahrscheinlich keine Möglichkeit, die von der Anwendung erzeugten SQL-Anweisungen zu verändern. Diese Aussage gilt in gewisser Art auch für gespeicherte Prozeduren. Deren Text können Sie zwar verändern oder mit Abfragehinweisen versehen, beim oder nach dem nächsten Software-Update gibt es dann aber wahrscheinlich Probleme. Einmal ganz abgesehen davon, wie sich so etwas auf Ihre Garantie- oder Supportansprüche auswirkt. Auch wenn Sie problematische Abfragen finden, können Sie in diesem Fall keine Abfragehinweise hinzufügen.

- Die Anpassung einzelner Anweisungen kann sehr mühselig sein. Oftmals ist es hilfreich, wenn Anweisungen gruppiert bzw. klassifiziert werden können, und sich somit Abfragehinweise für eine Gruppe ähnlich gearteter Anweisungen spezifizieren lassen.

In diesen Fällen sind Planhinweislisten ein geeignetes Mittel. In einer Planhinweisliste können Sie einen einzelnen Abfragetext oder auch ein Muster für Abfragen hinterlegen, wobei Sie für jeden Eintrag auch entsprechende Optimierungshinweise angeben. Bei der Erstellung eines Abfrageplans berücksichtigt der Optimierer die existierenden Planhinweislisten und arbeitet die dort hinterlegten Hinweise in den erstellten Abfrageplan ein.

Für das Anlegen einer Planhinweisliste verwenden Sie die gespeicherte Systemprozedur `sp_create_plan_guide`. Es gibt drei Kategorien von Planhinweislisten, die ich Ihnen zunächst nennen möchte. Im Anschluss an die Auflistung präsentiere ich Ihnen zu jeder Kategorie ein Beispiel:

- **OBJECT**-Planhinweislisten. Dies sind Planhinweislisten, die sich auf Abfragen in gespeicherten Prozeduren oder Funktionen beziehen. Eine solche Planhinweisliste muss unter Angabe des Abfragetextes und des Namens der gespeicherten Prozedur bzw. der Funktion angelegt werden. Der Optimierer vergleicht bei der Optimierung sowohl den eigentlichen Abfragetext als auch den Prozedur- oder Funktionsnamen. Wird eine Übereinstimmung festgestellt, werden die angegebenen Abfragehinweise berücksichtigt.
- **SQL**-Planhinweislisten. Eine Planhinweisliste dieses Typs bezieht sich auf eine konkrete Abfrage, die als eigenständige Abfrage ausgeführt wird, also nicht innerhalb einer Prozedur oder Funktion. Es ist auch möglich, ein Muster für den Abfragetext anzugeben. In diesem Fall werden die angegebenen Optimierungshinweise für alle Abfragen, deren Text dem angegebenen Muster entspricht, angewendet.
- **TEMPLATE**-Planhinweislisten. Eine TEMPLATE-Planhinweisliste spezifiziert das Parametrisierungsverhalten einer Abfrage bzw. einer Gruppe von Abfragen. Über diesen Hinweis können Sie die für die Datenbank angegebene Option der erzwungenen oder einfachen Parametrisierung (`PARAMETRIZATION FORCED` oder `PARAMETRIZATION SIMPLE`) für bestimmte Abfragen außer Kraft setzen und so eine einfache oder erzwungene Parametrisierung auf der Basis einzelner Abfragen einstellen.

Über `sp_create_plan_guide` können Sie nicht nur Abfragehinweise für die Optimierung angeben. Die Prozedur erlaubt auch die Spezifizierung eines konkreten Abfrageplans, der im XML-Format an die Prozedur übergeben werden muss.

Kommen wir nun zu einigen Verwendungsbeispielen von `sp_create_plan_guide`.

Im ersten Beispiel wird eine Planhinweisliste für eine Anweisung in einer gespeicherten Prozedur erzeugt. Diese Planhinweisliste ist also vom Typ `OBJECT`. Hierfür benötigen wir den Namen der Prozedur sowie den Text der entsprechenden SQL-Anweisung. Nehmen wir an, dass die Prozedur `SucheAdressen` bereits in der Datenbank `AdventureWorks2008` existiert:

```
create procedure SucheAdressen(@city nvarchar(30), @postalCode nvarchar(15)) as
 begin
   select PostalCode, City, AddressLine1, AddressLine2, ModifiedDate
     from Person.Address
```

```
    where ((@city is null) or (City like @city))
      and ((@postalCode is null) or (PostalCode like @postalCode))
 end
```

Wir wollen weiter davon ausgehen, dass Sie keine Möglichkeit haben, den Code der Prozedur zu verändern. Falls Sie sicher sind, dass für Ihre Anwendung eine Optimierung für den Parameterwert `@city='Dulu%'` erfolgen soll, dann können Sie also zum Text der Prozedur keinen entsprechenden Optimierungshinweis hinzufügen. Eine Planhinweisliste bietet in dieser Situation einen Ausweg. Diese Hinweisliste können Sie wie folgt erstellen:

```
use AdventureWorks2008;
exec sp_create_plan_guide
   @name = N'PlanGuide_AdrSuche'
  ,@stmt = N'select PostalCode, City
                  ,AddressLine1, AddressLine2, ModifiedDate
            from Person.Address
           where ((@city is null) or (City like @city))
             and ((@postalCode is null) or (PostalCode like @postalCode))'
  ,@type = N'OBJECT'
  ,@module_or_batch = N'dbo.SucheAdressen'
  ,@params = NULL
  ,@hints = N'option (optimize for (@city=N''Dulu%''))'
```

Die obige Anweisung erstellt eine Planhinweisliste für die durch den Parameter `@stmt` festgelegte `SELECT`-Anweisung innerhalb der Prozedur `SucheAdressen`. Wann immer für die Prozedur ein Ausführungsplan erstellt wird, ist der übergebene Wert für den Parameter `@city` für die Planerstellung nicht von Bedeutung. Der Plan wird stets für den Parameterwert `@city='Dulu%'` optimiert. Der eigentliche Code der Prozedur musste hierfür nicht modifiziert werden.

Sie können sich die gerade erstellte Planhinweisliste auch im Objekt-Explorer des Management Studios ansehen. Einen entsprechenden Ordner finden Sie unter *AdventureWorks2008/Programmierbarkeit/Planhinweislisten* (siehe Abbildung 9.32).

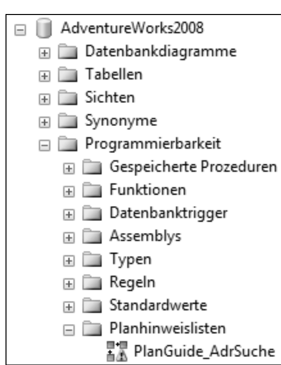


Abbildung 9.32:
Die erzeugte Planhinweisliste »PlanGuide_AdrSuche«

Für die aufgelisteten Planhinweislisten können Sie sich dann beispielsweise auch die Eigenschaften anzeigen lassen. Es ist sogar möglich, Planhinweislisten mit der grafischen Benutzeroberfläche hinzuzufügen. Der zugehörige Dialog ist allerdings nicht sehr komfortabel, sodass Sie sehr wahrscheinlich anstelle des Dialogs die entsprechenden gespeicherten Prozeduren verwenden werden – so wie in diesem Abschnitt gezeigt.

Das zweite Beispiel erzeugt eine Planhinweisliste für eine Ad-Hoc-SQL-Abfrage. Diese Art von Planhinweislisten wird für Abfragen benutzt, die über `sp_executesql` ausgeführt werden. In aller Regel sind dies Abfragen von Client-Anwendungen, die parametrisierte Abfragen an den Server senden. Schauen Sie sich die folgende Abfrage für eine Suche nach Personen an, wobei als Suchkriterium der Nachname verwendet wird:

```
exec sp_executesql
  @stmt = N'select FirstName, MiddleName, LastName
                 ,EmailPromotion
             from Person.Person
            where LastName like @p'
 ,@params = N'@p nvarchar(40)'
 ,@p = '%'
```

Wird die Abfrage so ausgeführt wird, wie in der obigen Anweisung dargestellt, so wird im Ausführungsplan ein Clustered Index Scan verwendet, da alle Zeilen der Tabelle `Person.Person` zurückgeben werden. Falls Sie wissen, dass in Ihrer Anwendung in aller Regel sehr detailliert nach dem Nachnamen gesucht wird, dass also in den meisten Fällen nur wenige Zeilen zurückgegeben werden, wäre es sicher besser, generell einen Index Seek über den auf der Spalte `LastName` vorhandenen Index durchzuführen. Die wenigen nicht optimalen Abfragen, in denen diese Indexsuche dann verwendet würde, obwohl der Index nicht selektiv genug ist, sollen einfach vernachlässigt werden.

Falls der Abfragetext von einer Anwendung generiert wird, auf deren Quellcode Sie keinen Einfluss haben, können Sie den Text der Abfrage nicht modifizieren. Über eine Planhinweisliste vom Typ SQL können Sie jedoch eine Indexverwendung für die Abfrage erzwingen:

```
exec sp_create_plan_guide
   @name = N'PlanGuide_PersSuche'
  ,@stmt = N'select FirstName, MiddleName, LastName
                  ,EmailPromotion
             from Person.Person
            where LastName like @p'
  ,@type = N'SQL'
  ,@module_or_batch = null
  ,@params = N'@p nvarchar(40)'
  ,@hints = N'option (table hint(Person.Person, forceseek))'
```

Über den Tabellenhinweis `TABLE HINT(Person.Person, FORCESEEK)` wird festgelegt, dass für die angegebene Abfrage stets eine Indexsuche durchgeführt wird.

Der Text der Abfrage muss dabei genau übereinstimmen. Hierbei zählen auch Leerzeichen sowie Zeilenumbrüche und die Groß-/Kleinschreibung. Auch der Name des Parameters ist relevant. In der obigen Darstellung wurden wegen der besseren Lesbarkeit Zeilenumbrüche eingefügt. Dadurch ist der in der Planhinweisliste verwendete Text der Abfrage verschieden vom Text der eigentlichen Abfrage, und die Planhinweisliste würde nicht gefunden werden.

Falls Sie eine Planhinweisliste für eine parametrisierte Abfrage erstellen möchten, ist die Ermittlung der Parameter nicht immer ganz einfach. Sie können die gespeicherte Systemprozedur `sp_get_query_template` verwenden, um die parametrisierte Form einer Abfrage zu ermitteln. Lesen Sie hierzu gegebenenfalls noch einmal in Abschnitt 9.3.1 nach und sehen Sie sich Abbildung 9.23 an.

Zum Abschluss möchte ich Ihnen noch ein Beispiel für eine Planhinweisliste des Typs `TEMPLATE` präsentieren. Bei Planhinweislisten dieses Typs geht es ausschließlich darum, die in der Datenbank vorhandene Einstellung für die Parametrisierung (`FORCED` oder `SIMPLE`, siehe Abschnitt 9.3.3) für eine bestimmte Abfrage zu überschreiben und für diese Abfrage eine Parametrisierung zu erzwingen oder eben nicht zu erzwingen, und zwar unabhängig von der Einstellung in den Datenbankoptionen. Bei Planhinweislisten vom Typ `TEMPLATE` dürfen deshalb nur die Hinweise `OPTION (PARAMETERIZATION FORCED)` bzw. `OPTION (PARAMETERIZATION SIMPLE)` verwendet werden.

Angenommen sie möchten, dass Abfragen der folgenden Form parametrisiert werden:

```
select sum(sod.LineTotal) as Bestellwert
 from Sales.SalesOrderHeader as soh
      inner join Sales.SalesOrderDetail as sod
        on sod.SalesOrderID = soh.SalesOrderID
 where soh.OrderDate between '20040101' and '20041231'
```

Diese Abfrage der Bestellwertsumme für einen Datumsbereich würde normalerweise nicht parametrisiert werden, da sie sich über mehrere Tabellen erstreckt. Sie können jedoch eine Parametrisierung erzwingen, indem Sie eine entsprechende Planhinweisliste erzeugen:

```
declare @stmt nvarchar(max)
       ,@params nvarchar(max)
exec sp_get_query_template
     N'select sum(sod.LineTotal) as Bestellwert
          from Sales.SalesOrderHeader as soh
               inner join Sales.SalesOrderDetail as sod
                       on sod.SalesOrderID = soh.SalesOrderID
        where soh.OrderDate between ''20040101'' and ''20041231'''
  ,@stmt out
  ,@params out

exec sp_create_plan_guide
```

```
 @name = N'Plan_Guide_Bestellwert'
,@stmt = @stmt
,@type = N'TEMPLATE'
,@module_or_batch = null
,@params = @params
,@hints = N'option (parameterization forced)'
```

Das obige Skript ermittelt zunächst die parametrisierte Form der Abfrage durch die gespeicherte Prozedur `sp_get_query_template`. Diese parametrisierte Form wird anschließend an `sp_create_plan_guide` übergeben. In der Folge würden dann alle Abfragen, die den Bestellwert für einen Datumsbereich ermitteln (so wie in der oben dargestellten Abfrage), parametrisiert.

Um eine Planhinweisliste zu löschen, können Sie den Objekt-Explorer oder die Prozedur `sp_control_plan_guide` verwenden. Auch das zeitweilige Deaktivieren und das erneute Aktivieren von Planhinweislisten können Sie über `sp_control_plan_guide` oder den Objekt-Explorer erledigen.

Zur Beobachtung der Verwendung von Planhinweislisten können Sie den Profiler verwenden. In der Gruppe *Performance* finden Sie diese beiden Ereignisse:

- **Plan Guide Successfull**. Der Optimierer hat eine existierende Planhinweisliste für die Planerstellung verwendet.
- **Plan Guide Unsuccessfull**. Der Optimierer hat eine zur Abfrage passende Planhinweisliste gefunden, konnte diese Planhinweisliste jedoch nicht verwenden. Dies kann zum Beispiel passieren, wenn im Abfragehinweis der Planhinweisliste ein bestimmter Index verwendet wird, der inzwischen nicht mehr existiert.

In der Ereignisspalte *TextData* wird dann jeweils der Name der Planhinweisliste ausgegeben.

Auch der Systemmonitor bietet zwei Indikatoren zur Beobachtung der Verwendung von Planhinweislisten an. In der Kategorie *SQL Statistics* finden Sie diese beiden Indikatoren:

- **Guided plan executions/sec**. Dieser Indikator zeigt die Anzahl der verwendeten Planhinweislisten pro Sekunde an.
- **Misguided plan executions/sec**. Dieser Wert gibt an, wie oft eine existierende Planhinweisliste nicht verwendet werden konnte, weil sie ungültig war.

9.5 Physikalische JOIN-Operatoren

JOIN-Operatoren werden Sie sehr häufig in Abfrageplänen finden. Zum einen sind natürlich bei Abfragen über mehrere Tabellen JOIN-Operatoren erforderlich. Aber auch dann, wenn eine Abfrage nur Daten aus einer einzigen Tabelle holt, kann es sein, dass der Ausführungsplan einen JOIN-Operator enthält. Dies ist zum Beispiel dann der Fall, wenn eine Indexsuche mit einer anschließenden Schlüsselsuche durchgeführt wird, so wie in Abbildung 9.31 zu sehen.

Es ist daher wichtig, dass Sie die unterschiedlichen physikalischen JOIN-Operatoren kennen und verstehen. Darüber hinaus ist es sehr hilfreich, wenn Sie erkennen, warum der Optimierer sich für einen bestimmten physischen JOIN-Operator entscheidet. Deshalb ist dieser Thematik hier auch ein eigener Abschnitt gewidmet.

Sie haben bereits in Kapitel 4 erfahren, dass der Optimierer prinzipiell drei physische JOIN-Operatoren kennt:

- MERGE JOIN
- HASH JOIN
- NESTED LOOPS

Allen diesen Operatoren ist zunächst gemein, dass Sie aus zwei Eingabetabellen eine Ausgabetabelle erzeugen. Bei einem JOIN über mehrere Tabellen werden die JOIN-Operatoren einfach kaskadierend angewendet. Bei drei verknüpften Tabellen benötigt die Abfrage also zwei JOIN-Operatoren. Ganz allgemein: bei *n* miteinander verknüpften Tabellen *n-1* JOIN-Operatoren.

Wir wollen nun die drei physischen JOIN-Operatoren untersuchen und verwenden hierfür zwei Testtabellen. Außerdem beschränken wir uns in den Experimenten auf INNER JOINs. Dies ist kein Problem, da die erhaltenen Ergebnisse prinzipiell ebenso auf OUTER JOINs zutreffen. Der folgende Teil zeigt auch, welche Auswirkungen die Auswahl eines nicht optimalen JOIN-Operators (zum Beispiel durch einen entsprechenden Abfragehinweis) haben kann.

Unsere beiden Testtabellen werden folgendermaßen erzeugt:

```
use QueryTest
go
-- Zwei Tabellen anlegen
-- T1: 1.000.000 Zeilen
if (object_id('T1', 'U') is not null)
  drop table T1
go
create table T1
 (
   c1 int not null identity(1,1)
  ,c2 char(200) not null default '*'
 )
go
create unique clustered index ix1 on t1(c1)
go
-- T2: 20.000 Zeilen
if (object_id('T2', 'U') is not null)
  drop table T2
go
create table T2
 (
   c1 int not null
  ,c2 char(800) not null default '#'
```

```
 )
create clustered index Ix2 on t2(c1)
go
-- Testdaten einfügen.
-- 1.000.000 Zeilen in T1 und 20.000 Zeilen in T2
insert T1 default values
go
insert T1(c2) select '*' from Numbers where n <= 1000000
go
insert T2(c1) select top 20000 c1 from t1 order by newid()
```

Das Skript erzeugt eine Tabelle `T1` mit 1.000.000 Zeilen und eine Tabelle `T2` mit 20.000 Zeilen, wobei die Werte in der Spalte `T2.c1` per Zufall aus dem Wertebereich von 1 bis 1.000.000 ausgewählt werden. Außerdem wird für beide Tabellen ein gruppierter Index angelegt.

9.5.1 MERGE JOIN

Ein MERGE JOIN ist immer dann optimal, falls die an der Verknüpfung beteiligten Tabellen bereits in sortierter Form vorliegen und außerdem sehr viele Zeilen verknüpft werden müssen.

In der folgenden Abfrage sind diese Bedingungen erfüllt:

```
select *
  from t1 inner join t2
                on t2.c1=t1.c1
```

Beide Tabellen verfügen über einen entsprechenden gruppierten Index, der die Daten in sortierter Reihenfolge im Sinne der JOIN-Bedingung enthält. Daneben enthält die Abfrage keine weiteren Filterkriterien, es werden demnach prinzipiell alle Zeilen aus beiden Tabellen zusammengefügt. Den grafischen Ausführungsplan sehen Sie in Abbildung 9.33.

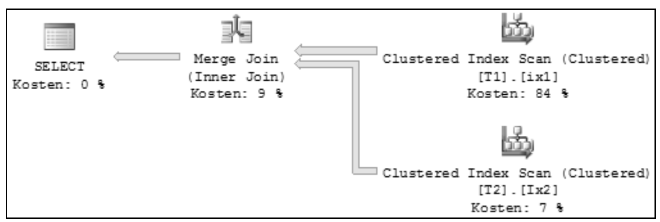


Abbildung 9.33: *Ausführungsplan mit MERGE JOIN*

Sowohl die äußeren (oberen) Eingabezeilen als auch die inneren (unteren) Eingabezeilen werden jeweils in sortierter Form verarbeitet. Hierzu wird die äußere Eingabe durchlaufen und für jede Zeile der erste passende Wert der inneren Eingabe gesucht. Danach kann die innere Schleife einfach so lange Zeile für Zeile abarbeiten, bis sich das Sortierkriterium ändert. Sobald dies der Fall ist, wiederholt sich der Vorgang für die nächste äußere Eingabezeile. Dies wird so lange fortgesetzt, bis in der äußeren Schleife alle Zeilen verarbeitet wurden. Da für jede äußere Zeile jeweils nur die erste innere Zeile gesucht werden muss, ist ein MERGE JOIN für eine Verarbeitung von vielen Zeilen sehr effizient. Der

Optimierer kann sich sogar dann für einen MERGE JOIN entscheiden, wenn die Eingabetabellen überhaupt nicht im Sinne der JOIN-Bedingung sortiert sind. Unter Umständen erfolgt dann die Sortierung in einem zusätzlichen Schritt vor dem eigentlichen JOIN. Schauen Sie sich hierzu bitte das folgende Beispiel an:

```
drop index Ix2 on T2
go
select *
  from t1 inner join t2 on t2.c1=t1.c1
```

Durch das Löschen des gruppierten Index der Tabelle T2 wird auch die Sortierung nach der Spalte c1 entfernt. Dadurch sind die Zeilen für die Tabelle T2 nicht mehr so sortiert, dass sie für den MERGE JOIN unmittelbar verwendet werden können. Dennoch wählt der Optimierer einen MERGE JOIN, wie der Abfrageplan in Abbildung 9.34 zeigt.

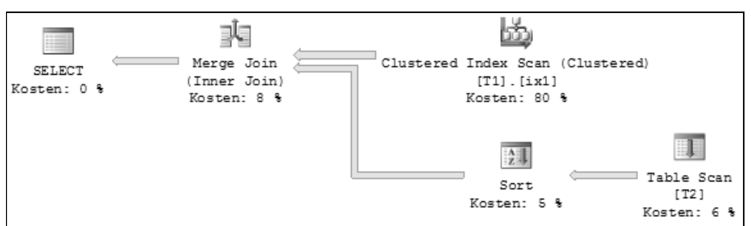


Abbildung 9.34: *MERGE JOIN mit vorgeschalteter Sortierung*

Im Plan ist recht deutlich zu sehen, dass die für den MERGE JOIN erforderliche Sortierung einfach hinzugefügt wird. Diese Sortierung ist relativ kostengünstig, weil die Tabelle T2 nur 20.000 Zeilen enthält. Daher ist auch hier der MERGE JOIN die optimale Lösung.

9.5.2 HASH JOIN

Ein HASH JOIN verwendet einen Hash Match-Operator für die JOIN-Implementierung. Dies ist immer dann günstig bzw. erforderlich, falls viele Zeilen, die nicht entsprechend der JOIN-Bedingung sortiert sind, miteinander verknüpft werden sollen. Schauen Sie sich dazu bitte das folgende Beispiel an:

```
create clustered index Ix2 on t2(c1)
go
drop index ix1 on T1
go
select *
  from t1 inner join t2 on t2.c1=t1.c1
```

Wir erzeugen hier den zuvor gelöschten Index auf der Tabelle T2 erneut und löschen dafür den gruppierten Index auf der Tabelle T1. Nun sind die 1.000.000 Zeilen der Tabelle T1 nicht mehr im Sinne der JOIN-Bedingung sortiert. Ein MERGE JOIN mit einer vorherigen Sortierung der Tabellenzeilen in einem zusätzlichen Schritt (so wie im Beispiel zuvor) ist auf Grund der Zeilenanzahl jetzt zu teuer. Der Optimierer entscheidet sich an dieser Stelle für einen HASH JOIN, wie Sie im Ausführungsplan sehen können.

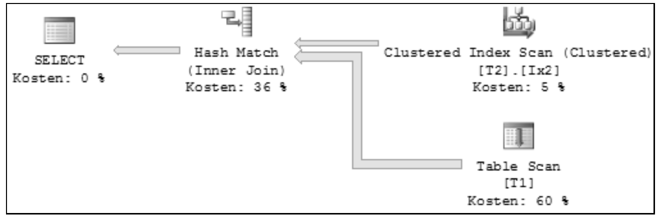


Abbildung 9.35:
Hash Match-Operator für die Implementierung eines JOIN

Ein HASH JOIN verwendet eine Hash-Tabelle als Hilfskonstrukt. Über diese Hash-Tabelle werden dann Suchoperationen und Verknüpfungen durchgeführt. Der obere Eingabezweig ist die äußere Schleife, über die die Hash-Werte für die Hash-Tabelle bestimmt werden. Für jeden Wert in der äußeren Schleife wird eine Suchoperation für den ermittelten Hash-Wert entsprechend des unteren Zweigs durchgeführt. Wird der Wert gefunden, stimmen beide Zeilen überein, und die Zeile wird in die Hash-Tabelle aufgenommen.

In unserem Beispiel ist die Hash-Tabelle übrigens so groß, dass sie nicht im Hauptspeicher abgearbeitet werden kann (zumindest ist dies auf meiner Maschine der Fall). Wenn Sie vor der Ausführung der Abfrage `SET STATISTICS IO ON` ausführen, so sehen Sie in den ausgegebenen Statistiken einen entsprechenden Hinweis:

```
'Worktable'-Tabelle. Scananzahl 0, logische Lesevorgänge 0, physische Lesevorgänge 0,
Read-Ahead-Lesevorgänge 0, logische LOB-Lesevorgänge 0, physische LOB-Lesevorgänge 0,
Read-Ahead-LOB-Lesevorgänge 0.
```

Die Hash-Tabelle wird in einer temporären Tabelle geführt, die in der Systemdatenbank *tempdb* verwaltet wird. Leider enthält die Ausgabe keine Information darüber, wie viele Lese- und Schreibvorgänge gegen diese Tabelle durchgeführt wurden.

Es ist möglich, Abfragehinweise zum Erzwingen eines bestimmten physischen JOIN-Operators zu verwenden. Falls Sie Zweifel haben, dass ein HASH JOIN unter Verwendung einer *WorkTable* wirklich die beste Wahl ist, können Sie durch einen solchen Hinweis dafür sorgen, dass zum Beispiel ein MERGE JOIN ausgeführt wird. Der Text der Abfrage sieht dann so aus:

```
select *
  from t1 inner merge join t2 on t2.c1=t1.c1
```

Für den MERGE JOIN müssen die Zeilen der Tabelle `T1` zuvor sortiert werden, so wie im MERGE JOIN-Beispiel aus dem vorherigen Abschnitt. Wie erwartet, enthält der Ausführungsplan einen entsprechenden Sort-Operator (siehe Abbildung 9.36).

Auffällig ist, dass die Abfrage nun teuer genug für eine parallele Verarbeitung ist, was natürlich darauf schließen lässt, dass der erforderliche Einsatz von Ressourcen (wie zum Beispiel CPU und E/A) erheblich gegenüber der vorherigen Version gestiegen ist. Insgesamt sind die prognostizierten Abfragekosten etwa sechs Mal so hoch wie zuvor. Der Abfragehinweis hat sich also in keiner Weise bezahlt gemacht!

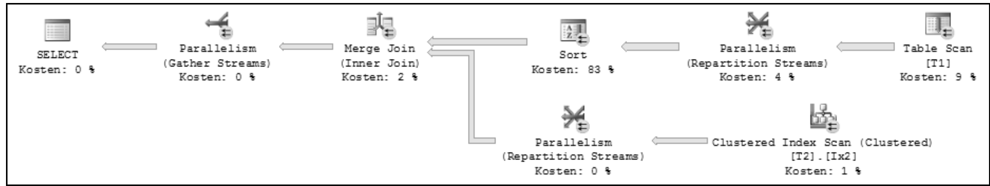


Abbildung 9.36: *Erzwungener MERGE JOIN mit vorgeschalteter Sortierung*

Auch hier wird übrigens eine *WorkTable* benötigt, diesmal für die Sortierung der 1.000.000 Zeilen der Tabelle `T1`. Diese Sortierung ist denn auch mit 83 Prozent der Gesamtkosten der bestimmende Operator im Hinblick auf die Kosten der Abfrage.

9.5.3 NESTED LOOPS

Ein NESTED LOOP JOIN ist immer dann sinnvoll, wenn in der äußeren Schleife nur wenige Zeilen enthalten sind und die innere Schleife durch einen unterstützenden Index durchsucht werden kann. Es ist auch möglich, dass der Optimierer einen temporären Index erstellt, um die innere Suche zu beschleunigen.

Natürlich wollen wir auch hier ein Beispiel betrachten. Schauen Sie sich hierzu bitte die folgende Abfrage an:

```
create unique clustered index ix1 on t1(c1)
go
select *
  from t1 inner join t2 on t2.c1=t1.c1
                       and t1.c1 <= 100
```

Zunächst wird dafür gesorgt, dass die beiden beteiligten Tabellen jeweils über einen gruppierten Index verfügen. Die `SELECT`-Anweisung sieht im Prinzip so aus wie im ersten Beispiel, in dem der Optimierer sich für einen MERGE JOIN entschieden hatte. Der Unterschied zum ersten Fall ist hier die `WHERE`-Klausel, durch die aus der Tabelle `T1` lediglich 100 Zeilen herausgefiltert werden. Damit ist hier ein NESTED LOOP-Operator letztlich effizienter als ein MERGE JOIN.

Den erzeugten Ausführungsplan sehen Sie in Abbildung 9.37.

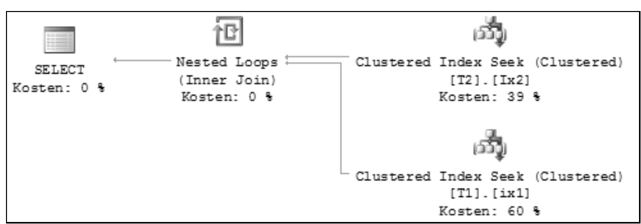


Abbildung 9.37: *NESTED LOOP JOIN*

Die äußere (obere) Eingabe für den NESTED LOOP JOIN wird über eine Suche im gruppierten Index der Tabelle `T2` ermittelt. Der Optimierer »weiß« anhand der Verknüpfungsbedingung, dass für die Tabelle `T2` nur Zeilen mit einem Wert von `c1 <= 100` verarbeitet werden sollen. Von den existierenden 20.000 Zeilen in der Tabelle `T1` sind dies in meinen Beispieldaten nur zwei Zeilen, und daher ist eine Suche im Index hier die beste Option. Für jede in der Tabelle `T2` gefundene Zeile wird dann ein Clustered Index Seek mit dem gefundenen Wert für `c1` in der Tabelle `T1` ausgeführt. Die übereinstimmenden Zeilen werden anschließend durch den NESTED LOOP-Operator miteinander verknüpft.

9.6 Auffinden geeigneter Indizes

Bereits in den Kapiteln 5 und 6 haben Sie gesehen, welche enorme Auswirkung passende Indizes auf die Abfrageleistung haben. Aus diesem Grund ist es natürlich wesentlich, dass Sie für Ihre Datenbank(en) die passenden Indizes erzeugen. In diesem Abschnitt werden wir uns mit dieser Thematik intensiv auseinandersetzen. Dabei werden Sie erfahren, welche Strategie Sie verfolgen können, und welche Faktoren für das Anlegen von Indizes zu beachten sind. Wir beginnen zunächst ohne den Einsatz technischer Hilfsmittel, wie zum Beispiel Ausführungspläne. Das Ziel ist das Erzeugen eines möglichst optimalen Satzes von Indizes, wobei wir zunächst davon ausgehen wollen, dass lediglich eine mehr oder weniger detaillierte Kenntnis über die zu erwartenden Abfragen existiert. Darüber hinaus ist natürlich auch das Datenbankschema bekannt. Wir wollen also diese beiden Informationsquellen verwenden, um dem optimalen Satz von initial benötigten Indizes möglichst nahe zu kommen.

Dieser Abschnitt enthält auch Beispiele, die zeigen, dass die zur Verfügung stehenden Hilfsmittel, wie zum Beispiel Ausführungspläne mit Informationen über fehlende Indizes, nicht in jedem Fall unüberlegt eingesetzt werden sollten.

Im nachfolgenden Kapitel 11 greifen wir das Thema Indexoptimierung noch einmal auf. Dort erfahren Sie, wie ein laufendes System überwacht werden kann, um sowohl fehlende als auch überflüssige Indizes herauszufinden.

Da Indizes generell nur redundante Daten enthalten, die bei jeder Aktualisierung von Tabellendaten ebenfalls aktualisiert werden müssen, liegt die Schlussfolgerung nahe, dass Indizes nur für die Beschleunigung von Leseoperationen nützlich sind und Aktualisierungen generell behindern. Im Prinzip ist gegen diese Aussage nichts einzuwenden, wenn wir bedenken, dass auch an der Aktualisierung von Daten oftmals implizite Leseoperationen beteiligt sind. So arbeiten zum Beispiel viele Operationen, die Daten verändern, auch selektiv. Nehmen wir hier nur einmal eine `UPDATE`-Anweisung. Diese wird in aller Regel zusammen mit einer `WHERE`-Klausel verwendet, weil ja nicht bei jedem `UPDATE` alle Zeilen einer Tabelle geändert werden sollen. In einem solchen Fall kann ein Index auf die in der `WHERE`-Klausel verwendete(n) Spalte(n) durchaus nützlich sein, auch wenn der Index selbst durch das `UPDATE` ebenfalls aktualisiert werden muss. Darüber hinaus müssen bei der Aktualisierung von Tabellendaten oftmals Überprüfungen von Fremdschlüsselbeziehungen durchgeführt werden. Auch hierfür sind entsprechende Indizes durchaus hilfreich.

Die Kunst beim Indexentwurf besteht darin, den für Ihre Anwendung optimalen Satz von Indizes zu erzeugen. Hiermit ist gemeint, dass tatsächlich nur die wirklich erforderlichen Indizes in der Datenbank angelegt werden. Diese Aufgabe ist in aller Regel recht anspruchsvoll, da durch Änderungen an der Anwendung und damit letztlich an den Abfragen auch die Menge der benötigten beziehungsweise nicht benötigten Indizes variiert. Selbstverständlich sollte eine Indexoptimierungsstrategie zunächst abschätzen, welche Spalten zur Filterung oder Sortierung herangezogen werden und für diese Spalten entsprechende Indizes planen. Hierbei werden Sie in vielen Fällen vor recht einfache Aufgaben gestellt. So ist zum Beispiel in einer Tabelle mit Kundendaten die Suche nach einer Kundennummer mit Sicherheit ein häufiges Szenario. Folglich wird ein Index auf der Spalte, welche die Kundennummer enthält, Abfragen dieser Art definitiv beschleunigen und kann daher mehr oder weniger bedenkenlos angelegt werden.

Nicht in allen Fällen ist es aber so einfach oder offensichtlich. Sie werden in diesem Abschnitt einige Szenarien und Beispiele sehen, in denen die Auswahl der »richtigen« Indizes nicht ganz so leicht ist. Index-Tuning ist eine Herausforderung, die sorgfältig geplant und angegangen werden sollte.

9.6.1 Suchargumente (SARGs)

Der Optimierer untersucht die in einer Abfrage angegebenen Prädikate in der `WHERE`-Klausel, um herauszufinden, ob für diese Bedingungen geeignete Indizes für die Unterstützung der Suche bzw. Filterung existieren. Ob diese Indizes dann später auch tatsächlich für die eigentliche Abfrageausführung verwendet werden, ist zu diesem Zeitpunkt noch nicht klar. Dies hängt beispielsweise von entsprechenden Kardinalitätsschätzungen ab. Voraussetzung dafür, dass ein entsprechender Index überhaupt in Betracht gezogen wird, ist der Aufbau der `WHERE`-Klausel.

Falls der Optimierer für die `WHERE`-Klausel Indizes verwenden kann, dann spricht man im Englischen davon, dass die `WHERE`-Bedingung *search arguments*, oder kurz: *SARGs* (im Deutschen leider eine etwas unglückliche Abkürzung) enthält. Es muss also zunächst einmal das Ziel sein, die in der `WHERE`-Klausel formulierten Filterbedingungen so zu verfassen, dass SARGs existieren. Andernfalls wird eine Indexsuche von vornherein unterbunden.

Nicht jede `WHERE`-Klausel enthält automatisch SARGs. Insbesondere sind die folgenden Konstrukte keine SARGs und führen deshalb dazu, dass ein Index für die Suche im Indexbaum unter Umständen nicht verwendet werden kann.

Mustervergleiche mit führendem Platzhalter

Der `LIKE`-Operator kann nur dann über einen Index suchen, wenn keine führenden Platzhalter verwendet werden. Die folgende Bedingung ist daher kein SARG:

```
where Name like '%der'
```

Diese Bedingung hingegen erfüllt das SARG-Kriterium:

```
where Name like 'Mei%'
```

Oder-Verknüpfungen

Eine Verknüpfung von Bedingungen mit OR kann dazu führen, dass existierende Indizes nicht verwendet werden. Gleiches gilt für Operatoren, die Vergleiche mit Elementen einer Menge durchführen: wie SOME, ANY, ALL oder IN.

Negative Vergleiche

Vergleiche durch <>, NOT EXISTS, NOT IN oder NOT LIKE können in vielen Fällen nicht von einem Index profitieren. Um herauszufinden, dass eine bestimmte Zeile nicht existiert, muss jede Zeile einer Tabelle untersucht werden. Solch eine Suche kann durch einen Index oft nicht unterstützt werden.

Spalten-Ausdrücke

Vermeiden Sie Ausdrücke mit Spalten, auf denen ein Index existiert. Die folgende Bedingung ist kein SARG:

```
where Bestelldatum + 30 > current_timestamp
```

Ein existierender Index auf der Spalte Bestelldatum wird nicht für eine Suche in Betracht gezogen. Falls Sie Ausdrücke dieser Art benötigen, so versuchen Sie, diese umzuformulieren. Zum Beispiel so:

```
where Bestelldatum > current_timestamp - 30
```

Diese Bedingung ist logisch mit der vorherigen identisch, verwendet jedoch die indizierte Spalte in keinem Ausdruck. Daher handelt es sich bei diesem Ausdruck um ein SARG.

Falls Sie in Abfragen Ausdrücke für indizierte Spalten benötigen, können Sie eine berechnete Spalte zu Ihrer Tabelle hinzufügen. Berechnete Spalten dürfen auch in Indizes verwendet werden. Dadurch können Sie in vielen Fällen Ausdrücke mit indizierten Spalten vermeiden.

Verwenden von Funktionen

Ebenso wie bei einem Spaltenausdruck ist auch ein Scan erforderlich, falls Sie indizierte Spalten in Funktionen verwenden. Schauen Sie sich hierzu das folgende Beispiel an:

```
where left(LastName,1) = 'M'
```

Ein existierender Index auf der Spalte LastName kann hier für die Suche nicht verwendet werden. Die Suchbedingung kann jedoch umformuliert werden, sodass ein SARG entsteht:

```
where LastName like 'M%'
```

Die folgende WHERE-Klausel, die alle Bestellungen des Monats Oktober im Jahr 2008 abfragt, enthält ein weiteres Beispiel für ein »Nicht-SARG«:

```
where year(Bestelldatum) = 2008
  and month(Bestelldatum) = 10
```

Ebenso wenig ist das folgende, logisch äquivalente Prädikat ein SARG:

```
where convert(varchar(6), Bestelldatum, 112) = '200810'
```

Die Bedingung kann aber so formuliert werden, dass die indizierte Spalte nicht in einer Funktion verwendet wird und somit von einem existierenden Index profitieren kann:

```
where Bestelldatum between '20081001' and '20081031'
```

Bitte denken Sie daran, dass die Verwendung von Indizes durch die Formulierung von Suchbedingungen als »Nicht-SARGs« nicht unterbunden wird. Die Rede ist lediglich davon, dass im Falle der Verwendung eines »Nicht-SARG«-Ausdrucks kein Index für eine Suche über den Indexbaum verwendet werden kann. Ein Scan eines nicht gruppierten Index kann durchaus durchgeführt werden, wenn der Optimierer entscheidet, dass dadurch weniger Lesevorgänge entstehen. Dies ist dann der Fall, wenn der nichtgruppierte Index abdeckend ist und insgesamt weniger Spalten enthält als die Tabelle bzw. der gruppierte Index. Möglich ist auch die Verwendung eines Index für Sortierungen. Es ist also nicht gesagt, dass ein entsprechender Index niemals verwendet wird, falls Sie keine SARGs benutzen.

Ein Index Scan ist letztlich aber nicht der Grund für das Anlegen eines Index. Wenn Sie einen Index erstellen, dann tun Sie dies bitte immer in der Absicht, dass dieser Index Such- oder Sortieroperationen unterstützt. Sollte Ihr Index für diese Operationen nicht verwendet werden, ist sein Nutzen zumindest fragwürdig.

9.6.2 Auswahl des gruppierten Index für eine Tabelle

Für jede Tabelle können Sie einen gruppierten Index erstellen, der stets alle Tabellendaten enthält. Diese Daten werden im gruppierten Index in sortierter Form entsprechend der Indexspalten abgelegt. Da es je Tabelle nur einen gruppierten Index geben kann, sollten Sie diesen Index entsprechend sorgfältig auswählen.

SQL Server legt für den Primärschlüssel einer Tabelle einen gruppierten Index an, sofern Sie nichts anderes angeben. In vielen Fällen ist dieses Standardverhalten akzeptabel – weswegen es ja auch als Standard festgelegt wurde. Dennoch möchte ich Ihnen raten, dass Sie darüber nachdenken, ob der Primärschlüssel Ihrer Tabelle wirklich der geeignete Kandidat für den gruppierten Index ist.

Die richtige Wahl des gruppierten Index ist allein deshalb wichtig, weil eine nachträgliche Änderung – vor allem bei großen Tabellen – fast immer mit einem enormen Ressourcenverbrauch verbunden ist. Wenn Sie einen gruppierten Index verändern, werden alle Daten der Tabelle umsortiert und auch alle nichtgruppierten Indizes neu erstellt (siehe Kapitel 5). Dieser Vorgang benötigt natürlich Zeit und erzeugt Blockierungen auf der betroffenen Tabelle.

Ein gruppierter Index ist besonders für Bereichssuchen optimal. Wenn Sie also für eine Suche nach Tabellendaten oft den `BETWEEN`-Operator oder Vergleichsoperatoren wie < oder > verwenden, haben Sie hier eventuell einen Kandidaten für einen gruppierten Index. Darüber hinaus sind Spalten, nach denen gruppiert wird, um Aggregate, wie beispielsweise `MIN`, `MAX` oder `AVG`, zu berechnen, sehr gut als gruppierter Index geeignet. In beiden Fällen kann eine Suche über den gruppierten Index die erste passende Zeile finden und dann einfach die in sortierter Form vorliegenden Daten durchlaufen. Ein gruppierter Index ist auch für eine Sortierung nach den Indexspalten über die `ORDER BY`-Klausel perfekt geeignet.

Schlagen Sie bitte noch einmal in Kapitel 5 nach, wenn Ihnen der Aufbau eines gruppierten Index nicht mehr geläufig ist.

Auf der anderen Seite sollten Sie bedenken, dass ein gruppierter Index die Daten in sortierter Reihenfolge auf der Festplatte verwaltet, wenn Sie die Indexspalten auswählen. Falls die Spaltenwerte des gruppierten Index mit der Zeit nicht dauerhaft wachsen, sind Einfügeoperationen (also `INSERT`s) in nicht sequenzieller Reihenfolge erforderlich. In diesem Fall muss der gruppierte Index beim Einfügen neuer Daten umsortiert werden. Dies kann dazu führen, dass beim Einfügen von Daten die Blattseiten des gruppierten Index auseinandergerissen werden müssen, damit die sortierte Reihenfolge der Tabellendaten eingehalten werden kann. Diese sogenannten Page Splits sind eine relativ teure Operation.

Verwenden Sie also für den gruppierten Index eine Spalte, deren Wert beständig wächst und sich nicht ändert, die also nicht in `UPDATE`-Operationen verwendet wird.

Aber selbst dann ist es nicht immer ganz einfach. Stellen Sie sich bitte vor, dass Sie einen Primärschlüssel auf einer `IDENTITY`-Spalte anlegen und hierfür einen gruppierten Index erstellen. Die Spaltenwerte für den Primärschlüssel wachsen beständig an; jede neue Zeile enthält einen höheren Wert als die vorhergehende Zeile. In diesem Fall ist ein Page Split maximal am Ende des Index beim Einfügen neuer Daten erforderlich, und diese Operation ist nicht allzu kostspielig. Auf der anderen Seite wird ein solcher Primärschlüssel sehr wahrscheinlich ein künstlicher Schlüssel sein, der für die Benutzer einer entsprechenden Anwendung keinerlei Bedeutung hat. Die Anwender werden diesen Schlüssel also nie zu Gesicht bekommen und somit auch nicht in Abfragen verwenden. Daher ist es fraglich, ob ein solcher Schlüssel wirklich ein geeigneter Kandidat für den gruppierten Index ist.

Programmierer tendieren oftmals dazu, GUIDs als Primärschlüssel zu verwenden. Wenn Sie nicht aufpassen, kann es leicht passieren, dass Sie einen gruppierten Index auf einer `UNIQUEIDENTIFIER`-Spalte bekommen – und das ist meistens nicht die optimale Lösung. Schauen Sie sich bitte die folgende Tabellendefinition an:

```
create table Verkauf
 (
   VerkaufID uniqueidentifier not null default newid() primary key
  ,Verkaufsdatum date not null default current_timestamp
  ,...
 )
```

Die Definition der Spalte `VerkaufID` bewirkt hier, dass der gruppierte Index für die Tabelle `Verkauf` auf dieser Spalte erzeugt wird. Da die Funktion `NEWID()` zufällige GUID-Werte erzeugt, führen `INSERT`-Operationen auf dieser Tabelle zu massiven Umsortierungen in Kombination mit Page Splits. Vermeiden Sie so etwas unbedingt! Verwenden Sie zumindest die Funktion `NEWSEQUENTIALID()` zum Erzeugen von GUIDs. Diese Funktion erzeugt monoton wachsende Werte für die zurückgegebenen GUIDs, wodurch Page Splits minimiert werden. Im vorliegenden Fall wäre aber sehr wahrscheinlich die Spalte `Verkaufsdatum` ein besserer Kandidat für den gruppierten Index. Die Werte für diese Spalte werden in der Regel anwachsen, außerdem wird die Spalte sehr wahrscheinlich für Bereichssuchen, Gruppierungen und Sortierungen verwendet werden.

Denken Sie also bitte sorgfältig darüber nach, welche Spalten Sie für den gruppierten Index verwenden.

9.6.3 Selektivität und Sortierung

Sie wissen bereits, dass ein Index generell sowohl für Such- als auch für Sortieroperationen nützlich ist. In diesem Abschnitt wollen wir die Kriterien, die eine Verwendung oder auch Nichtbeachtung eines Index beeinflussen, etwas näher untersuchen.

Erinnern Sie sich zunächst bitte noch einmal daran, wie eine Suche über einen nicht abdeckenden (und folglich nichtgruppierten) Index abläuft. In so einem Fall werden über den Index zunächst nur Zeiger auf die eigentlichen Daten ermittelt. Welcher Art ein solcher Zeiger ist, hängt davon ab, ob die entsprechende Tabelle über einen gruppierten Index verfügt. Existiert ein gruppierter Index, dann ist der Zeiger ein Schlüsselwert im gruppierten Index. Über diesen Schlüsselwert können dann die erforderlichen Daten durch eine Suche im gruppierten Index ermittelt werden. Dieser Vorgang wird als *Schlüsselsuche* bezeichnet und ist recht teuer. Falls kein gruppierter Index existiert, enthalten die Zeiger direkte Verweise auf die Datenseiten. In diesem Fall kann eine entsprechende Datenseite direkt über einen sogenannten *RowId Lookup* ermittelt werden. Dieser RowId Lookup ist nicht ganz so kostspielig wie eine Schlüsselsuche im gruppierten Index, aber immer noch relativ teuer.

Bei der Indexsuche über einen nicht abdeckenden Index tritt ein recht interessantes Phänomen auf: Es ist nämlich sehr gut möglich, dass erforderliche Schlüsselsuchen oder RowId Lookups die Abfragekosten soweit verteuern, dass eine Indexsuche keinerlei Vorteil gegenüber einem Table- oder Clustered Index Scan mit sich bringt. Im Gegenteil – eine Indexsuche kann sogar teurer werden als eine vergleichbare Scan-Operation. Dies ist immer dann der Fall, wenn über den nicht abdeckenden Index relativ viele Zeilen zurückgegeben werden, sodass in Folge viele Schlüsselsuchen oder RowId Lookups erforderlich sind. Dies bedeutet, dass die Formulierung von Suchbedingungen in einer »SARG-Form« nicht automatisch die Verwendung eines existierenden Index bewirkt. Ob

ein nicht abdeckender Index für die Suche verwendet wird, hängt letztlich von der Kardinalitätsschätzung ab. Ein zum SARG passender Index wird nur dann verwendet, wenn er auch selektiv genug ist, wenn also die Anzahl der Zeilen genügend klein ist.

Es ist im Übrigen erstaunlich, wie hoch die Selektivität sein muss, damit der Optimierer einen nicht abdeckenden Index verwendet. Schauen wir uns hierzu ein einfaches Beispiel an. Zunächst wird eine Tabelle angelegt und mit Daten aus der Tabelle `AdventureWorks2008.Person.Person` gefüllt:

```
use QueryTest;
-- Anlegen einer Tabelle für den Test
if (object_id('Person', 'U') is not null)
  drop table Person
go
select BusinessEntityID as Id
      ,LastName         as Nachname
      ,FirstName        as Vorname
  into Person
  from AdventureWorks2008.Person.Person
go
```

Diese Tabelle enthält 19.972 Zeilen. Für unsere Experimente wollen wir nach der Spalte `ID` suchen und sehen, wie hoch die Selektivität der Abfrage sein muss, damit ein Index auf dieser Spalte verwendet wird.

Wir legen zunächst einen nichtgruppierten Index auf der Spalte `ID` an und schließen die Spalte `Vorname` in den Index mit ein:

```
create index IX_PTP_Id on Person(Id) include (Vorname)
```

Die folgende Abfrage verwendet eine Filterbedingung, die eine Suche über den Index veranlassen könnte:

```
select Nachname from Person where Id<12000
```

Der Optimierer entscheidet sich allerdings für einen Table Scan (siehe Abbildung 9.38), weil dadurch weniger Leseoperationen erforderlich sind.

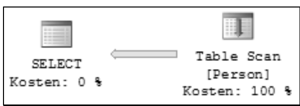


Abbildung 9.38:
Table Scan-Operation wegen zu geringer Selektivität des Index

Die Ursache hierfür ist, dass durch die `WHERE`-Bedingung insgesamt 11.194 Zeilen der Tabelle zurückgegeben werden. Dies sind ca. 56 Prozent der in der Tabelle enthaltenen Zeilen. Bei einer Suche über den Index müsste der Wert für die Spalte `Nachname`, der ja nicht direkt über den Index ermittelt werden kann, dann durch 11.194 zusätzliche RowId Lookups ermittelt werden: Damit wäre eine Indexsuche teurer als ein Table Scan.

Durch einen entsprechenden Indexhinweis können wir diese Aussage überprüfen. Die folgende Abfrage erzwingt die Verwendung des Index:

```
select Nachname from Person with (index=IX_PTP_Id) where Id<12000
```

Und tatsächlich ist im Ausführungsplan nun ein Index Seek mit einem entsprechenden RowId Lookup zu erkennen (Abbildung 9.39).

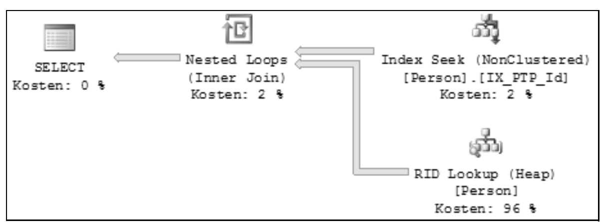


Abbildung 9.39:
Erzwungener Index Seek

Wenn Sie `SET STATISTICS IO ON` einschalten, erhalten Sie für beide Abfragen die Information über die erforderlichen Lesevorgänge. Das Resultat ist recht erstaunlich:

Für den Table Scan wurden 105 logische Leseoperationen benötigt. Bei der erzwungenen Indexsuche sind es 11.239. Damit ist der Table Scan in dieser Hinsicht um mehr als den Faktor 107 besser!

Es ist sicher leicht einzusehen, dass bei der Rückgabe vieler Zeilen ein Table Scan besser geeignet ist als eine Suche im nicht abdeckenden Index mit anschließendem Lookup. Die interessante Frage ist aber, wie selektiv ein Index sein muss, damit er für eine Suche verwendet wird. Das Ergebnis ist einigermaßen erstaunlich.

Ich habe die obige Abfrage mit kleiner werdenden Werten für den Vergleich wiederholt in der folgenden Weise ausgeführt:

```
select Nachname from Person where Id<40 option (recompile)
```

Der Abfragehinweis `OPTION (RECOMPILE)` dient hier dazu, einen eventuell vorhandenen parametrisierten Ausführungsplan zu ignorieren. Andernfalls würde stets der bei der ersten Ausführung erstellte Plan (also der Table Scan) für alle Ausführungen verwendet.

Tatsächlich ist es so, dass der Optimierer erst ab der Bedingung `ID < 40` eine Indexsuche verwendet. In diesem Fall werden 39 von 19.972 Zeilen zurückgegeben. Das bedeutet, dass der Index erst verwendet wird, sobald weniger als 39/19.972 ≈ 0,2 Prozent der Zeilen zurückgegeben werden!

Forscht man etwas nach, so findet man allerdings heraus, dass bereits ab ca. 100 Zeilen die Verwendung des Index Vorteile bringt. Dies sind ungefähr 0,5 Prozent der Zeilen. Der Optimierer verschätzt sich hier ein wenig, ohne dass dies allerdings dramatische Auswirkungen hat.

Sie können diese Ungenauigkeit leicht nachprüfen, indem Sie das folgende Skript ausführen:

```
set statistics io on
select Nachname from Person where Id<50
select Nachname from Person with (index=IX_PTP_Id) where Id<50
```

Im ersten Fall (ohne Indexhinweis) wird ein Table Scan ausgeführt, der die bereits aus dem ersten Experiment bekannten 105 logischen Leseoperationen benötigt. Die Suche über den Index benötigt hingegen nur 51 logische Leseoperationen, ist also ca. 50 Prozent besser. Wenn Sie das obige Skript mit verschiedenen Vergleichswerten ausführen und die erforderlichen Leseoperationen beobachten, werden Sie feststellen, dass die benötigten Leseoperationen für beide Abfragen etwa ab dem Wert 100 zu Gunsten der Indexsuche umschlagen.

Ob ein nicht abdeckender Index oder ein Table Scan für die Suche verwendet wird oder nicht, hängt neben der Selektivität auch davon ab, wie viele Daten eine Tabellenzeile insgesamt enthält. Ist eine Tabellenzeile sehr breit, dann ist ein Table Scan sehr teuer und damit auch dann unwahrscheinlich, falls eine Indexsuche viele Daten zurückgibt. Die Verwendung eines Index zahlt sich in diesem Fall auch bereits dann aus, wenn relativ viele Zeilen zurückgegeben werden.

Wenn wir unsere Tabelle `Person` um die folgende Spalte

```
alter table Person
  add BreiteSpalte nchar(2000) not null default'#'
```

ergänzen, dann ist ein Table Scan generell sehr teuer, weil die `BreiteSpalte` sehr viele Leseoperationen erfordert. Daher wird auch schon bei einer Rückgabe von relativ vielen Zeilen eine Indexsuche verwendet. Die folgende Abfrage gibt insgesamt 4.194 Zeilen zurück:

```
select Nachname from Person where Id<5000 option (recompile)
```

Dies sind nun immerhin ca. 21 Prozent der Zeilen. Trotzdem wird eine Indexsuche durchgeführt (siehe Abbildung 9.40), weil die erforderlichen 4.194 RowId Lookups immer noch günstiger sind als ein Table Scan.

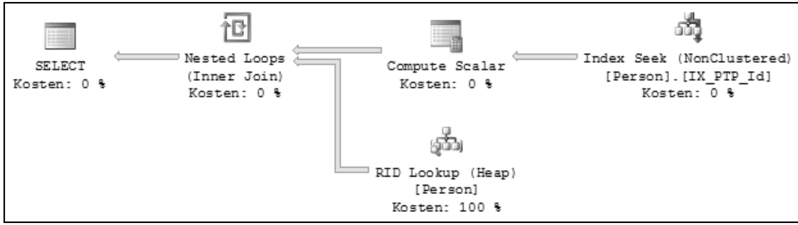


Abbildung 9.40: *Index Seek trotz geringer Selektivität*

Sie sehen also, dass es nicht ganz leicht ist vorherzusagen, wie hoch die Selektivität eines nicht abdeckenden Index sein muss, damit der Index für eine Suche verwendet wird. Dies hängt tatsächlich nicht nur von der Selektivität, sondern auch von den Tabellendaten ab.

Merken Sie sich bitte auch, dass ein Table Scan nicht grundsätzlich schlecht ist. Falls eine Abfrage viele Zeilen einer Tabelle mit wenigen oder schmalen Spalten zurückgibt, ist ein Table Scan oftmals die effizienteste Möglichkeit, Daten zu suchen.

Ich möchte allerdings noch einmal betonen, dass die Abwägung zwischen Tabellen- bzw. Clustered Index Scan oder Indexsuche nur bei nicht abdeckenden Indizes stattfindet. Ein abdeckender Index wird immer dann verwendet, wenn die Abfrage ein entsprechendes SARG enthält.

Schauen Sie sich hierzu die folgende Abfrage an:

```
select Vorname from Person where Id<12000
```

Die Spalte `Vorname` ist in den Index als `INCLUDED`-Spalte aufgenommen worden. Der Index enthält nun alle Informationen für die Suche und auch die zurückzugebenden Spalten. Daher wird der Index verwendet, obgleich er nicht selektiv genug ist (siehe Abbildung 9.41).

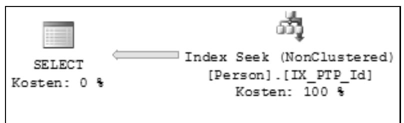


Abbildung 9.41:
Index Seek bei abdeckendem Index

Der Index Seek ist in diesem Fall einfach die günstigste Möglichkeit für eine Minimierung der erforderlichen Leseoperationen.

Wenn Sie nun darüber nachdenken, verstärkt abdeckende Indizes zu verwenden, dann ist gegen diesen Ansatz zunächst einmal nichts einzuwenden. Bedenken Sie aber bitte die folgenden Besonderheiten in Bezug auf abdeckende Indizes:

- Abdeckende Indizes benötigen zusätzlichen Speicherplatz und Verwaltungsaufwand, da sie gegenüber nicht abdeckenden Indizes mehr Daten enthalten.
- Abdeckende Indizes sind normalerweise maßgeschneidert für Abfragen. Dies bedeutet, dass Sie nur diejenigen Spalten in den Index einschließen, die auch wirklich in einer Abfrage ausgegeben werden. Ein nützlicher abdeckender Index kann beim Aufnehmen zusätzlicher Spalten in eine Abfrage zum nicht abdeckenden Index werden. In diesem Fall haben Sie ein Problem, weil der abdeckende Index nun in Wirklichkeit nicht abdeckend ist, und der höhere Speicherbedarf bzw. Verwaltungsaufwand damit nicht mehr gerechtfertigt ist. Ein solcher Fall ist schwer zu entdecken, weil der Index eventuell trotzdem noch verwendet wird, nur eben nicht mehr im »abdeckenden Sinn«. Denken Sie bitte an unser obiges Beispiel, in dem der Index für eine Suche über die Spalte `ID` verwendet wurde, ohne dass er für die Abfrage abdeckend war.
- Verwenden Sie niemals `SELECT *` in Produktivumgebungen. Durch `SELECT *` ist es sehr unwahrscheinlich, dass eine Abfrage von einem abdeckenden Index profitieren kann, denn dieser Index müsste ja alle Spalten enthalten – und das ist normalerweise nur beim gruppierten Index der Fall.

Ich möchte Ihnen nun anhand eines weiteren Beispiels zeigen, dass es nicht immer ganz einfach ist, sich für einen »richtigen« Index zu entscheiden. Das Beispiel zeigt auch, dass ein vom Optimierer bemängelter fehlender Index nicht in jedem Fall wirklich optimal ist.

Wir verwenden für dieses Beispiel eine Tabelle mit Kraftfahrzeugdaten, die folgenden Aufbau hat:

```
use QueryTest;
go
create table Kfz
 (
   FgstNr char(36) not null primary key
  ,Kennzeichen char(10) not null unique
  ,Erstzulassung date not null
  -- Platzhalter. Steht stellvertretend für weitere Spalten
  ,Platzhalter char(500) null
 )
```

In diese Tabelle werden nun 1000.000 Zeilen eingefügt:

```
-- Füge 1.000.000 Zeilen in die Tabelle ein
insert Kfz(FgstNr,Kennzeichen,Erstzulassung)
 select newid()                                            as FgStNr
       ,char(65 + abs(checksum(newid())) % 26)
          + char(65 + abs(checksum(newid())) % 26) + '-'
          + right('000000' + cast(n as varchar(7)), 6)      as Kennzeichen
       ,dateadd(d,-abs(checksum(newid())) % 1000,'20081231') as Erstzulassung
   from Numbers
  where n <= 1000000
```

Die Tabelle hat bislang nur einen gruppierten Index auf der Spalte `FgstNr`, die als Primärschlüssel festgelegt wurde. Nehmen wir nun an, dass eine Bereichssuche nach dem Datum der Erstzulassung durchgeführt werden soll. Das Ergebnis soll nach dem Kennzeichen sortiert ausgegeben werden. Die folgende Abfrage erledigt diese Aufgabe:

```
-- Alle Fahrzeuge, die im Mai 2006 zugelassen wurden
select Kennzeichen, FgstNr, Erstzulassung
  from Kfz
 where Erstzulassung between '20060501' and '20060531'
 order by Kennzeichen
```

Der Ausführungsplan zeigt einen Clustered Index Scan (siehe Abbildung 9.42) mit insgesamt 7.985 logischen Leseoperationen. Außerdem ist die Abfrage so teuer, dass sie parallel ausgeführt wurde. Sie erkennen im Ausführungsplan auch, dass ein fehlender Index entdeckt wurde, der die Abfrageleistung um über 94 Prozent verbessern würde. (In der Abbildung ist die komplette Indexdefinition leider aus Platzgründen nicht zu erkennen.)

```
Fehlender Index (Auswirkung 94.6648): CREATE NONCLUSTERED INDEX [<Name of Missing

SELECT          Parallelism          Sort          Clustered Index Scan (Clustered)
Kosten: 0 %     (Gather Streams)     Kosten: 1 %   [Kfz].[PK__Kfz__59962F612D47B39A]
                Kosten: 1 %                        Kosten: 98 %
```

Abbildung 9.42: *Ausführungsplan für die Suche nach dem Zulassungsdatum*

Wie zu erwarten, ist es ein Index auf der Spalte `Erstzulassung`, der außerdem die Spalten `FgstNr` und `Kennzeichen` einschließt. Wir legen diesen Index einmal an:

```
create nonclustered index Ix_Kfz_EZ_FgStNr_KZ
    on Kfz (Erstzulassung) include (FgstNr, Kennzeichen)
```

Wenn die Abfrage nun erneut ausgeführt wird, sieht der Ausführungsplan erheblich besser aus (siehe Abbildung 9.43), und es wird auch kein fehlender Index mehr beanstandet.

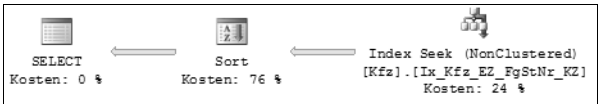


Abbildung 9.43:
Suche nach Erstzulassung mit passendem Index

Die erforderlichen E/A-Operationen haben sich durch den Index auf 24 gegenüber ursprünglich erforderlichen 7.985 verringert. Das sind etwa 330 Mal weniger!

Bitte bedenken Sie jedoch stets, dass der Index nur für die obige Abfrage die geschätzte Verbesserung bewirkt. Sobald sich die Daten oder die Parameter der Abfrage ändern, kann der Optimierer sich dazu entscheiden, den Index nicht zu verwenden.

Die vom Optimierer gelieferten Hinweise über fehlende Indizes bieten sicherlich eine enorm hilfreiche Möglichkeit für das Ergänzen nützlicher Indizes. Allerdings sind diese Ratschläge nicht in jedem Fall in dem Sinne korrekt, dass sie eine optimale Lösung ermöglichen. In der Online-Dokumentation finden Sie eine Reihe von Hinweisen über Ungenauigkeiten der Vorhersage. Implementieren Sie also die Empfehlungen auf keinen Fall ohne zu überlegen, sondern prüfen Sie bitte, ob die vom Optimierer vorgeschlagenen Indizes in dieser Form tatsächlich hilfreich sind.

Falls Sie das nicht glauben mögen, schauen Sie sich bitte das folgende Beispiel an. Wir entfernen zunächst den etwas weiter oben angelegten Index wieder:

```
drop index Ix_Kfz_EZ_FgStNr_KZ on Kfz
```

Wenn wir nun diese Abfrage starten, die alle Kfz-Zulassungen der Jahre 2005 bis 2007 zurückliefert,

```
select Kennzeichen, FgstNr, Erstzulassung
  from Kfz
 where Erstzulassung between '20050101' and '20071231'
 order by Kennzeichen
```

ist der im Ausführungsplan angegebene Indexhinweis exakt derselbe, wie bereits weiter oben angegeben. Dies ist aus meiner Sicht sehr fragwürdig, weil die Abfrage insgesamt knapp 64.000 Zeilen zurückliefert, also mehr als die Hälfte der Tabellenzeilen. Es fällt auf, dass die Sortierung nach der Spalte `Kennzeichen` im Index nicht berücksichtigt wird – und das erscheint doch einigermaßen seltsam.

Legen wir den vorgeschlagenen Index erneut an und führen die Abfrage anschließend noch einmal aus, so ergibt sich der in Abbildung 9.44 gezeigte Ausführungsplan.

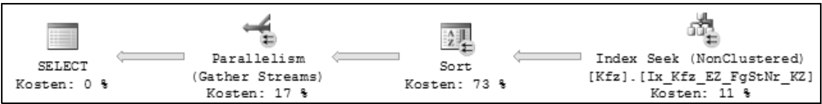


Abbildung 9.44: *Ausführungsplan mit parallelem Index Seek*

Die Kosten für die obige Abfrage betragen nun lediglich noch 3,21 gegenüber 8,26 ohne Index – immerhin eine Verbesserung um den Faktor 2,5. Allerdings werden im Ausführungsplan auch die folgenden Punkte deutlich:

1. Der Index führt tatsächlich zu der prognostizierten Verbesserung der Abfrageleistung.
2. Die Kosten der Abfrage werden größtenteils durch die erforderliche Sortierung verursacht.
3. Im Ausführungsplan wird kein weiterer fehlender Index angegeben. Das erscheint anhand der hohen Kosten für die erforderliche Sortieroperation und der parallelen Ausführung der Abfrage sehr fraglich.

Es liegt die Vermutung nahe, dass ein Index auf der Spalte `Kennzeichen`, der die Spalten `FgstNr` und `Erstzulassung` beinhaltet, hier wesentlich angebrachter wäre. Wir legen den Index also einmal an:

```
create index IxKfz_Kennzeichen
    on Kfz(Kennzeichen) include (FgStNr,Erstzulassung)
```

Wenn die Abfrage nun noch einmal ausgeführt wird, ergibt sich ein Ausführungsplan, der ohne die teure Sort-Operation auskommt (siehe Abbildung 9.45).

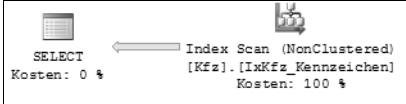


Abbildung 9.45: *Ausführungsplan mit Index, der die Sortierung unterstützt*

Die Kosten für die Abfrage betragen nunmehr nur noch 0,59! Im Ausführungsplan bekommen Sie nun wieder den Hinweis über einen fehlenden Index auf der Spalte `Erstzulassung`, so wie am Beginn dieses Experiments. Legen Sie den Index an, so wird er tatsächlich auch verwendet, und der Ausführungsplan enthält wieder die teure Sort-Operation. Unser Index auf der Spalte `Kennzeichen` wird dann tatsächlich nicht mehr berücksichtigt.

Was ist da passiert? Wenn wir uns die benötigten E/A-Operationen ansehen, wird deutlich, dass bei einer Verwendung unseres Index auf der Spalte `Kennzeichen` 654 logische Leseoperationen erforderlich sind. Wird der Index auf der Spalte `Erstzulassung` benutzt, werden nur 458 logische Leseoperationen benötigt. Irgendwie wird der fehlende Index also offensichtlich in Bezug auf die erforderlichen E/A-Operationen geprüft. Die teure Sortierung hingegen wird hierbei nicht berücksichtigt. Dies wird vor allem deutlich, wenn Sie über `SET STATISTICS TIME ON` die erforderliche Ausführungszeit und CPU-Verwendung beobachten. Bei Verwendung des die Sortierung unterstützenden Index auf der Spalte `Kennzeichen` ist die CPU-Belastung um ca. den Faktor 13 geringer als bei einer Verwendung des vorgeschlagenen Index auf der Spalte `Erstzulassung`.

Der im Abfrageplan ausgegebene Hinweis für einen fehlenden Index ist in diesem Fall also nicht völlig korrekt. Sicherlich bewirkt der dort vorgeschlagene Index eine Verbesserung der Abfrageleistung, er ist jedoch nicht optimal. Die manuelle Analyse der Abfrage und des Abfrageplans sowie die Erzeugung des aus dieser Analyse hervorgegangenen Index hat in unserem Fall nochmals eine Verbesserung um mehr als den Faktor 5 bewirkt.

Denken Sie bitte stets daran, wenn Sie automatisch erzeugte Hinweise anwenden, und vertrauen Sie diesen Hinweisen nicht blind. Da Sie dieses Buch in der Hand halten und lesen, haben Sie ja längst erkannt, dass Sie einer automatischen Optimierung nicht in jedem Fall glauben können und menschliches Eingreifen erforderlich ist.

Interessant ist hier auch die Tatsache, dass der Optimierer für den Fall, dass beide Indizes existieren, einen Plan erstellt, der den Index auf der Spalte `Erstzulassung` verwendet. Es wird also der Plan mit der höheren CPU-Belastung ausgewählt!

Eine tiefergehende Analyse zu fehlenden Indizes führt der Datenbankmodul-Optimierungsratgeber, der seit der Version 2008 auch als Datenbankoptimierungsratgeber bezeichnet wird, durch. Wie Sie den Datenbankoptimierungsratgeber verwenden, erfahren Sie in Kapitel 11.

9.6.4 Verknüpfungen und Fremdschlüssel (Foreign Keys)

SQL Server legt in einigen Fällen selbstständig Indizes an. So wird zum Beispiel für eine `PRIMARY KEY`-Einschränkung automatisch ein Index erzeugt. Gleiches gilt für `UNIQUE`-Einschränkungen, deren Überprüfung ebenfalls durch einen automatisch erzeugten Index unterstützt wird.

Bedenken Sie aber bitte, dass dies nicht für Fremdschlüsselbeziehungen gilt. Für `FOREIGN KEY`-Einschränkungen wird nicht automatisch auch ein Index erzeugt. Es ist unter Umständen nicht ganz einfach zu entscheiden, ob für eine Fremdschlüsselbeziehung ein Index hilfreich ist oder nicht. Oftmals werden Fremdschlüsselbeziehungen in JOINs verwendet, um Abfragen über mehrere Tabellen auszuführen. Bei Tabellen, für die eine Fremdschlüsselbeziehung existiert, sind diese JOINs in den meisten Fällen sogenannte EQUI JOINs. Dies bedeutet, dass auf Gleichheit hin – in der Regel in zwei Spalten – getestet wird. Meist sind dies 1:n-Beziehungen, wie zum Beispiel im Fall einer Tabelle `Kunde` und einer Tabelle `Rechnung`. Diese beiden Tabellen werden wahrscheinlich über eine Spalte `KundenNr` miteinander verknüpft sein. Falls Sie nun Rechnungen und Kunden abfragen wollen, sieht eine solche Abfrage ungefähr so aus:

```
select Kunde.Name, Rechnung.Betrag, Rechnung.Datum
  from Kunde
       inner join Rechnung
             on Rechnung.KundenNr = Kunde.KundenNr
```

Die Verknüpfung testet also auf Gleichheit in der Spalte `KundenNr` beider Tabellen. Stellen Sie sich nun vor, dass Ihre Datenbank 30.000 Kunden und insgesamt 2.000.000 Rechnungen enthält. In diesem Fall ist ein Index auf der Spalte `KundenNr` in der Tabelle `Rechnung` sicherlich hilfreich. (In der Tabelle `Kunde` wird ein entsprechender Index auf der Spalte `KundenNr` mit Sicherheit existieren, denn dies ist der Primärschlüssel der Tabelle.) Durch

einen Index auf Rechnung(KundenNr) kann der obige JOIN als MERGE JOIN ausgeführt werden, was beim Verknüpfen vieler Zeilen optimal ist. Falls die beiden verknüpften Tabellen also viele Zeilen enthalten, ist ein Index auf einer Fremdschlüsselspalte sicherlich hilfreich. Lesen Sie noch einmal in Abschnitt 9.5.1 nach, in dem MERGE JOINS erklärt werden.

Etwas komplizierter wird es, wenn Sie Tabellen verwenden, die eigentlich Datentypen repräsentieren. Es ist oft der Fall, dass die in SQL existierenden Datentypen für die Beschreibung der Eigenschaften einer Entität nicht ausreichen. In einer solchen Situation wird häufig eine Tabelle angelegt, die als erweiterter Datentyp dient. Beispiele sind etwa ein Rechnungstyp oder eine Verkaufsregion. Diese Tabellen haben in der Regel nur sehr wenige Zeilen. Wenn Sie nun in einer Tabelle mit Rechnungen auf einen Rechnungstyp verweisen, so wird ein Index auf der Spalte RechnungsTypID in der Tabelle Rechnung nicht selektiv sein. Normalerweise haben Sie vielleicht 10 unterschiedliche Rechnungstypen. Haben Sie 1.000.000 Rechnungen im System, so gibt es also im Schnitt 100.000 Rechnungen je Rechnungstyp. Eine Suche nach dem Rechnungstyp wird also von einem Index nicht profitieren können.

Es gibt jedoch einen anderen Fall zu bedenken, den ich Ihnen an einem Beispiel verdeutlichen möchte. Wir verwenden hierzu unsere etwas weiter oben angelegte Tabelle mit KFZ-Zulassungsdaten. Unser Datenmodell soll nun auch einen KFZ-Typ aufnehmen können, den wir in einer eigenen Tabelle unterbringen. Diese Tabelle wird so erzeugt:

```
create table KfzTyp
 (
  KfzTypID int identity(1,1) not null primary key
 ,TypName nvarchar(100) not null
 ,MaximalGewicht int not null
 )
```

Zu Beginn sollen drei Typen existieren, die in die Tabelle eingefügt werden:

```
insert KfzTyp(TypName, MaximalGewicht)
  values ('PKW', 2300)
        ,('Kleintransporter', 3500)
        ,('LKW', 35000)
```

Was nun noch fehlt, ist die Fremdschlüsselbeziehung zwischen den Tabellen Kfz und Kfz-Typ. Hierzu fügen wir zur Tabelle Kfz eine Spalte KfzTypID hinzu und füllen diese Spalte mit zufälligen Werten:

```
alter table Kfz
  add KfzTypID int null
go
update Kfz set KfzTypID = 1 + abs(checksum(newid())) % 3
```

Nachdem dies erledigt ist, kann die Fremdschlüsselbeziehung erzeugt werden. Zuvor legen wir noch fest, dass für jede Zeile in der Tabelle Kfz ein entsprechender Typ angegeben werden muss. NULL-Werte sind in der Spalte Kfz.KfzTypID also nicht erlaubt:

```
alter table Kfz
  alter column KfzTypID int not null
go
alter table Kfz
  add constraint FK_Kfz_KfzTyp
     foreign key (KfzTypID) references KfzTyp(KfzTypID)
```

Wenn nun die folgende Abfrage ausgeführt wird, die alle Kraftfahrzeuge vom Typ »LKW« zurückgibt,

```
select Kfz.*
  from Kfz
       inner join KfzTyp on KfzTyp.KfzTypID = Kfz.KfzTypID
 where KfzTyp.TypName = 'LKW'
```

dann wäre ein eventuell existierender Index auf `Kfz(KfzTypID)` für die Ausführung sicherlich nicht hilfreich, es sei denn, er ist abdeckend, das heißt, er enthält die restlichen Spalten der Tabelle `Kfz` als eingeschlossene Spalten. Die Filterung nach der `KfzTypID` ist nicht selektiv genug für die Verwendung eines entsprechenden Index. Auf der Tabelle `Kfz` ist also immer ein Clustered Index Scan erforderlich, wie der Ausführungsplan in Abbildung 9.46 zeigt.

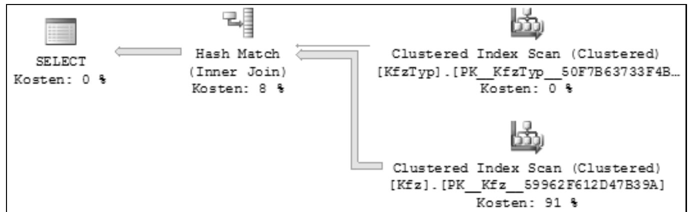


Abbildung 9.46: *Abbildung 10.46:IINNER JOIN mit Clustered Index Scan*

Wozu also einen Index auf der Spalte `Kfz.KfzTypID` anlegen? Es wird interessant, wenn Sie aus der Tabelle `KfzTyp` Zeilen löschen möchten. Angenommen, die folgende Zeile wurde versehentlich eingefügt:

```
insert KfzTyp(TypName, MaximalGewicht)
 values ('Fahrrad', 10)
```

Wenn Sie diese Zeile wieder löschen, muss sichergestellt sein, dass die existierende Fremdschlüsselbeziehung nicht verletzt wird. Folglich müssen alle Zeilen der Tabelle `Kfz` dahingehend überprüft werden, ob nicht ein Verweis auf die zu löschende Zeile in der Tabelle `KfzTyp` existiert. Wenn nun ein Index auf `Kfz(KfzTypID)` angelegt wurde, ist eine solche Prüfung sehr schnell erledigt, da über den Index sofort die Nichtexistenz einer solchen Zeile festgestellt werden kann. Ohne einen entsprechenden Index muss diese Prüfung durch einen Table Scan bzw. Clustered Index Scan erfolgen.

Beim Löschen der »Fahrrad«-Zeile aus der Tabelle `KfzTyp` werden ohne Index auf `Kfz(KfzTypID)` 7.227 logische Lesevorgänge benötigt. Mit einem entsprechenden Index sind es dann lediglich noch drei logische Lesevorgänge – eine Verbesserung um den Faktor 2.400!

Leider kann ich Ihnen an dieser Stelle keine allgemeine Lösung für dieses Problem präsentieren. Es hängt tatsächlich von Ihren Anwendungen und Ihren Daten ab, in welchen Fällen Sie für Fremdschlüsselbeziehungen Indizes erzeugen sollten. Wenn Sie vor der Frage stehen, ob Sie einen Index für eine Fremdschlüsselbeziehung anlegen sollen oder nicht, dann erinnern Sie sich bitte an das in diesem Abschnitt Gesagte.

9.7 Zusammenfassung

In diesem Kapitel haben Sie einen umfassenden Einblick in die Ausführung von Abfragen durch SQL Server erhalten. Durch diese Kenntnisse sind Sie nun in der Lage, Ihre Abfrageausführung zu analysieren und Problemen auf den Grund zu gehen.

Sie wissen jetzt, wie der Plancache funktioniert und in welcher Weise Abfragepläne in diesem Cache verwaltet werden.

Darüber hinaus haben Sie erfahren, wie der Optimierer Ausführungspläne erstellt und welche Kriterien der Optimierer bei der Wahl geeigneter physischer Operatoren berücksichtigt. Sie wissen außerdem, wie Abfragen parametrisiert werden und welche Vor- und Nachteile Parameter Sniffing bietet. Ebenso sind Sie in der Lage, die sich möglicherweise ergebenden Probleme in Bezug auf Parameter Sniffing mit diversen Methoden zu lösen.

Sie können nun die unterschiedlichen physischen JOIN-Operatoren auseinanderhalten und wissen, wie diese Operatoren funktionieren, und welcher JOIN-Operator in welchem Fall optimal ist.

Zum Abschluss sind wir noch einmal darauf eingegangen, wie Sie einen möglichst optimalen Satz von Indizes für Ihre Datenbank(en) herausfinden können.

Wir haben in diesem Kapitel sehr intensiv mit grafischen Ausführungsplänen gearbeitet und eine Reihe von Problemen allein durch die Untersuchung dieser Ausführungspläne aufgedeckt. Als wichtigste Erkenntnis sollten Sie auf jeden Fall Folgendes mitnehmen:

Falls Sie im grafischen Ausführungsplan Unterschiede zwischen geschätzten und tatsächlichen Werten finden, ist dies ein Indiz dafür, dass der Optimierer anfängt zu »raten«, anstatt zu schätzen. Diese Aussage gilt insbesondere für die geschätzte und tatsächliche Zeilenanzahl. Falls diese beiden Werte stark voneinander abweichen, sollten Sie unbedingt die Ursache hierfür herausfinden.

In diesem Kapitel haben Sie das notwendige Rüstzeug für das Aufspüren derartiger Probleme erhalten. Oftmals sind es nicht aktuelle Statistiken. Es kann jedoch beispielsweise auch sein, dass Sie einen gespeicherten und parametrisierten Plan verwenden, der für die Abfrage nicht optimal ist.

10 Auffinden problematischer Abfragen

Im vorherigen Kapitel lagen die Schwerpunkte auf der Analyse und der Optimierung problematischer Abfragen. Dabei taucht natürlich unmittelbar die Frage auf, woran man eigentlich erkennen kann, dass eine Abfrage problematisch ist – sodass eine entsprechende Analyse und Optimierung sich letztlich überhaupt lohnt. Wie findet man Abfragen, die von einer Optimierung profitieren können bzw. für die eine Optimierung erforderlich ist?

Die Beantwortung dieser Frage steht im Mittelpunkt dieses Kapitels. Um problematische Abfragen zu finden, müssen Sie das laufende System überwachen und Informationen über die Abfrageausführung prüfen. Der Ansatz aus dem vorherigen Kapitel, bei dem wir in den meisten Fällen einfach eine Online-Überwachung durchgeführt haben – in der Regel durch das Einschließen des tatsächlichen Ausführungsplans bei der Ausführung einer Abfrage – ist hierfür nicht ausreichend. Eine solche Vorgehensweise ist für die Überwachung eines Produktivsystems schlichtweg nicht möglich. Vielmehr ist es erforderlich, die Historie der ausgeführten Abfragen für eine Überwachung heranzuziehen.

In diesem Kapitel lernen Sie unterschiedliche Möglichkeiten für die Protokollierung der Abfrageausführung und natürlich auch für die Auswertung der protokollierten Informationen kennen. Hierbei begeben wir uns auf bereits bekanntes Terrain und verwenden für die Protokollierung und Auswertung die folgenden Techniken:

- **Dynamische Verwaltungssichten**. In einem ersten Ansatz, sozusagen im kleineren Maßstab, können Sie einfach die dynamischen Verwaltungssichten verwenden, in denen ja auch Abfragestatistiken und Ausführungspläne gespeichert werden. Diese Informationen sind jedoch flüchtig, also nicht unbegrenzt haltbar. Wenn Sie Ihr System dauerhaft überwachen möchten und eine Historie über einen längeren Zeitraum benötigen, werden Sie nicht umhinkommen, entsprechende permanent gespeicherte Protokolle aufzuheben. Bei einer Verwendung dynamischer Verwaltungssichten entfällt die Notwendigkeit, eine Protokollierung zu konfigurieren. Die Protokollierung wird bereits durch SQL Server selber automatisch erledigt. Dies ist einerseits natürlich sehr bequem; der Preis, den wir für diese Bequemlichkeit zahlen müssen, ist aber eben die Vergänglichkeit der protokollierten Informationen. Wir haben dynamische Verwaltungssichten in den vorangegangenen Kapiteln bereits sehr häufig verwendet, um Informationen über ausgeführte Abfragen zu erhalten. Daher werden Sie in diesem Kapitel nur noch einmal eine kurze Wiederholung finden, die speziell darauf ausgelegt ist, Abfragen mit einem hohen Ressourcenverbrauch aufzufinden. Diese Wiederholung dient letztlich auch dazu, dass Sie ein tieferes Verständnis für das Verwaltungs-Data Warehouse entwickeln.

- **SQL Server Profiler**. Eine Ablaufverfolgung des Profilers kann so konfiguriert werden, dass relevante Informationen zur Abfrageleistung in ihr enthalten sind. Es ist auch möglich, eine solche Ablaufverfolgung in einer Datei zu speichern. Dadurch haben Sie die Möglichkeit, Informationen zur Abfrageleistung – auch über deren Lebensdauer in den dynamischen Verwaltungssichten hinaus – zu speichern.
- **Verwaltungs-Data Warehouse (VDWH)**. Dies ist sozusagen der »große Hammer«, mit dem alle erdenklichen Protokollierungs- und Auswertungsszenarien an zentraler Stelle verwaltet werden können. Im VDWH können Informationen zur Abfrageleistung, zu E/A-Operationen und sogar SQL Server-Ablaufverfolgungen gespeichert werden. Mit einem Wort: Alle in Bezug auf das Auffinden problematischer Abfragen relevanten Informationen können an diesem zentralen Ort aufgezeichnet und abgefragt werden. Darüber hinaus erhalten Sie dadurch eine gewisse Standardisierung Ihrer Messungen. Der Einsatz des VDWH ist daher sicherlich die eleganteste Möglichkeit der Systemüberwachung und wird sich in der Zukunft sehr wahrscheinlich dementsprechend etablieren. Daher wird der Schwerpunkt in diesem Kapitel auch auf der Verwendung des VDWH zur Systemüberwachung liegen.

10.1 Überwachung durch dynamische Verwaltungssichten

Im Rahmen dieses Buches sind Ihnen dynamische Verwaltungssichten das erste Mal bereits in Kapitel 2 begegnet. In Kapitel 4 haben Sie dann gesehen, wie Sie E/A-Operationen je Datenbank und aufgetretene Wartezustände abfragen können. Die dort vorgestellten Abfragen sind eine sehr gute Möglichkeit, um sehr schnell einen Überblick über eventuelle Probleme in einem laufenden System zu gewinnen. Über die aufgetretenen Wartezustände können Sie Rückschlüsse auf mögliche Performance-Engpässe ziehen. Hierzu müssen Sie einfach die Ursache für die in der »Hitliste« oben stehenden Wartezustände herausfinden und erklären können. Dies betrifft allerdings mehr den Bereich der Fehlersuche und liegt daher außerhalb des Rahmens dieses Buches. Thema dieses Kapitels ist das Auffinden problematischer Abfragen – und dafür ist eine Untersuchung der Wartezustände nur sehr bedingt geeignet. Wartezustände dienen mehr zur Inspektion des allgemeinen Zustands eines Systems.

10.1.1 Auswertung der E/A-Operationen

Etwas anders sieht es mit den E/A-Operationen je Datenbank aus. Abbildung 10.1 zeigt die E/A-Rangliste meines Systems, die durch die in Kapitel 4 vorgestellte Abfrage ermittelt wurde.

Datenbank	Typ	Gelesen/MB	Geschrieben/MB	EA Summe/MB	Wartezeit (sek)	Rang
tempdb	Daten	1007.19	1097.80	2104.98	24687.78	1
QueryTest	Protokoll	1.93	5664.78	5666.71	7972.97	2
QueryTest	Daten	2121.98	4911.13	7033.12	4860.99	3
VDWH	Protokoll	0.88	196.93	197.80	313.35	4
VDWH	Daten	87.74	177.26	265.00	228.44	5
tempdb	Protokoll	0.66	14.05	14.71	169.75	6
msdb	Protokoll	0.50	5.58	6.08	73.34	7
msdb	Daten	47.23	3.77	51.01	73.33	8
AdventureWorks2008	Daten	149.14	1.75	150.89	57.66	9
master	Daten	17.15	0.02	17.16	10.71	10
AdventureWorks2008	Protokoll	0.44	0.16	0.60	5.91	11

Abbildung 10.1: *E/A-Operationen je Datenbank*

Diese Rangliste eignet sich hervorragend als Ausgangspunkt für eine weiterführende Analyse. Betrachten Sie insbesondere die erforderlichen Lesevorgänge und konzentrieren Sie sich in weiteren Analysen zunächst auf die Datenbank, die in der Rangliste der Leseoperationen an erster Stelle steht. Die Ursache für massive Leseoperationen in OLTP-Systemen sind meist Scans. In OLTP-Systemen sind Scan-Operationen normalerweise unabsichtlich und ein Indiz für Folgendes:

- **Fehlende Indizes**. Indexsuchen minimieren Lesevorgänge. Falls keine passenden Indizes existieren, müssen sequenzielle Suchen, also Scan-Operationen, durchgeführt werden. Dadurch steigt die Anzahl erforderlicher Leseoperationen um einige Größenordnungen an.
- **Fehlende oder ungeeignete Filterbedingungen**. Überprüfen Sie, ob Ihre Abfragen nicht zu viele Zeilen zur Anwendung übertragen, und die Filterung erst auf der Client-Seite anstatt bereits auf der Datenbank erfolgt. Oft können auch »selbstgebaute« JOINs die Ursache sein. In so einem Fall werden die beteiligten Tabellen separat abgefragt und die Zeilen dann erst in der Client-Anwendung verknüpft. Denken Sie daran, dass nicht jeder Anwendungsentwickler über entsprechende Kenntnisse des mengenorientierten Ansatzes verfügt, der mit SQL zur Verfügung steht. Oftmals werden SQL-Abfragen von Softwareentwicklern entworfen, die eine prozedurale Denkweise gewöhnt sind. Ineffiziente SQL-Anweisungen sind dann häufig die Folge.

  Schauen Sie sich bitte das folgende Beispiel an. Angenommen, aus den Tabellen `Kunde` und `Rechnung` sollen für alle Kunden der Kategorie »A« die Umsätze für den Monat Oktober 2008 in einer Tabelle der Anwendung dargestellt werden. Die entsprechende Abfrage sieht so aus:

  ```
  select Kunde.KundenNr, Kunde.Name
        ,sum(Rechnung.Betrag()) as Betrag
    from Rechnung
         inner join Kunde on Kunde.KundenNr = Rechnung.KundenNr
   where Rechnung.Datum between '20081001' and '20081031'
     and Kunde.Kategorie = 'A'
   group by Kunde.KundenNr, Kunde.Name
  ```

  Es ist jedoch nicht ungewöhnlich, dass Ihnen in einer Anwendung zum Beispiel die folgende Implementation dieser Aufgabe begegnet.

Zunächst werden *alle* Rechnungen des Monats Oktober 2008 geholt:

```
select KundenNr
      ,sum(Betrag()) as Betrag
  from Rechnung
 where Datum between '20081001' and '20081031'
 group by KundenNr
```

Danach wird dann in der Anwendung in einer Schleife für jeden gefundenen Kunden die Kategorie ermittelt:

```
select Kategorie, Name
  from Kunde
 where KundenNr = <VergleichsNr>
```

Erst dann erfolgt die Auswertung der Kategorie und die Aufnahme des Kunden in die Tabelle, sofern der Kunde ein A-Kunde ist.

Natürlich ist eine solche Vorgehensweise alles andere als effizient. Sie sollten daher darauf achten, dass so etwas unterbunden wird. Falls Sie keinen Einfluss auf die Anwendungsentwicklung haben, etwa weil es sich bei der entsprechenden Software um ein zugekauftes Produkt handelt, existiert hier jedoch keine Möglichkeit, etwas zu ändern.

Wenn Anwendungen von Entwicklern erstellt werden, die über ungenügende Erfahrung mit SQL verfügen, werden Sie normalerweise auch ungewöhnlich viele Cursor antreffen, weil die Anwendungsentwickler der Meinung sind, dass eine bestimmtes Problem nicht mit SQL gelöst werden kann und daher prozedural angegangen werden muss. Cursor sind oftmals ebenfalls ein Indiz dafür, dass JOINs ausprogrammiert werden, anstatt sie einfach in einer SELECT-Anweisung hinzuschreiben, und daher in den meisten Fällen ebenso ineffizient.

- **Nicht optimale Ausführungspläne**. Die vielfältigen Ursachen für nicht optimale Ausführungspläne haben wir in Kapitel 9 bereits eingehend untersucht. Nicht aktuelle Statistiken oder die Parametrisierung mit untypischen Parameterwerten sind häufige Ursachen für die Existenz nicht optimaler Ausführungspläne.

In OLAP-Systemen sieht es übrigens etwas anders aus. Dort sind große, durch Scans verursachte E/A-Operationen durchaus nichts Ungewöhnliches. Wenn also Ihre OLAP-Datenbank in der Rangliste weit oben steht, dann ist dieser Umstand nicht beunruhigend. Mit anderen Worten: In einem OLAP-System ist die Rangliste der E/A-Operationen als Ausgangspunkt für eine tiefere Analyse nur bedingt geeignet.

Falls die Systemdatenbank *tempdb* in der Rangliste sehr weit oben steht (so wie in Abbildung 10.1 zu sehen), dann sollten Sie ebenfalls der Ursache auf den Grund gehen. Die Datenbank *tempdb* ist letztlich eine Ressource, die sich alle bestehenden Verbindungen teilen müssen. Aus diesem Grund kann die *tempdb* durchaus zu einem Engpass werden. Hierfür kommen im Wesentlichen drei Punkte in Frage:

- **Manuell erzeugte temporäre Objekte in T-SQL-Stapeln**. Sie können im T-SQL-Code ganz bewusst temporäre Objekte verwenden, die in der Datenbank *tempdb* gespeichert werden. Dies betrifft alle Objekte, deren Name mit dem Zeichen # beginnt. Auf diese Weise können zum Beispiel Cursor oder auch Tabellen angelegt werden. Oftmals ist die eigentliche Ursache für den Einsatz solcher temporärer Objekte ebenfalls eine gewisse Unerfahrenheit mit der Sprache SQL.

- **Automatisch erzeugte temporäre Objekte**. In Kapitel 9 haben Sie Beispiele gesehen, in denen für die Abfrageausführung eine *Worktable* verwendet wurde, weil der verfügbare Hauptspeicher etwa für eine Sortierung oder Hash-Tabelle nicht ausreichte. Die in einem solchen Fall angelegte temporäre Tabelle, die *Worktable* eben, wird ebenfalls in der Datenbank *tempdb* erzeugt. In einigen Fällen kann das Erzeugen einer *Worktable* durch einen passenden Index vermieden werden. Sie sollten also überprüfen, warum eine Abfrage eine *Worktable* benötigt. Hierauf kommen wir etwas weiter unten noch einmal zurück. Oftmals ist das automatische Erzeugen eines temporären Index oder einer temporären Tabelle aber auch die beste Möglichkeit, eine Abfrage auszuführen. Daher sind zum Beispiel *Worktable*-Objekte nicht grundsätzlich schlecht. Wenn für die Ausführung einer Abfrage automatisch temporäre Objekte angelegt werden, dann sollten Sie den entsprechenden Ausführungsplan aber zumindest einmal näher inspizieren und verstehen, warum temporäre Objekte – wie zum Beispiel ein Table- oder ein Index Spool – erforderlich sind.
- **Anlegen von Zeilenkopien in der Isolierungsstufe SNAPSHOT**. Wenn Sie Transaktionen in der Isolierungsstufe `SNAPSHOT` mit `READ COMMITTED` ausführen, werden Kopien der von einer Transaktion betroffenen Zeilen erzeugt, damit Lese- und Schreiboperationen sich in der Isolierungsstufe `READ COMMITTED` nicht blockieren. Diese Zeilenkopien werden in der Datenbank *tempdb* verwaltet. Aus diesem Grund kann die *tempdb* in der Rangliste sehr weit oben stehen, obwohl für die eigentliche Abfrageausführung keine nennenswerten Ressourcen in der *tempdb* benötigt werden.

10.1.2 Ermitteln fehlender Indizes

Wenn Sie Abfragen für die in der Rangliste der E/A-Operationen führenden Datenbanken näher untersuchen, sollten Sie herausfinden, ob der Optimierer für diese Abfragen Indizes vermisst. In Kapitel 6 haben Sie zwei Wege zur Ermittlung fehlender Indizes kennengelernt. Die dort vorgestellten Möglichkeiten können Sie nun anwenden, um zu prüfen, ob Indizes für in der Rangliste führende Datenbanken fehlen.

Denken Sie dabei bitte daran, dass die vom Optimierer erzeugten Hinweise über fehlende Indizes eine gewisse Ungenauigkeit aufweisen, und dass Sie die in diesen Hinweisen enthaltenen Empfehlungen nur nach einer manuellen Prüfung anwenden und eventuell noch modifizieren sollten.

10.1.3 Auswerten der im Plancache gespeicherten Ausführungspläne

Die am Beginn von Kapitel 9 vorgestellte Abfrage zur Untersuchung der Abfragestatistiken können Sie einsetzen, um zu prüfen, welche der im Plancache enthaltenen Ausführungspläne für die Mehrzahl der E/A-Operationen verantwortlich sind. Die Abfrage gibt auch den Namen der Datenbank zurück, sodass Sie leicht ermitteln können, welche Abfragen für eine bestimmte Datenbank die meiste E/A-Belastung erzeugt haben.

Allerdings gibt es hier auch noch ein kleines Problem mit nicht parametrisierten Abfragen. Wenn Abfragen, die in ihrer Struktur identisch sind, nicht parametrisiert werden, so finden Sie für jede dieser Abfragen einen separaten Plan im Plancache. Denken Sie bitte an das Beispiel aus Kapitel 9, in dem wir den Plancache mit derartigen Abfragen »überflutet« haben.

Schauen Sie sich bitte die folgende Abfrage an, in der alle Bestellsummen für einen bestimmten Kunden summiert werden:

```
select sum(sod.LineTotal)
  from Sales.SalesOrderHeader as soh
       inner join Sales.SalesOrderDetail as sod
               on sod.SalesOrderID = soh.SalesOrderID
 where soh.CustomerID=11015
```

Diese Abfrage wird nicht parametrisiert, weil sie sich über mehrere Tabellen erstreckt. Daher werden Sie für die Abfrage einen Eintrag im Plancache finden, der genau für die `CustomerID` 11.015 erzeugt wurde. Wenn Sie nun die identische Abfrage für einen anderen Kunden, also mit einem anderen Vergleich für die `CustomerID`, ausführen, wird erneut ein Eintrag im Plancache erstellt, diesmal mit einem anderen Wert für den Vergleich. Angenommen, die Abfrage benötigt jedesmal zehn logische Lesevorgänge und wird am Tag 1.000.000 Mal ausgeführt. Damit läge die Abfrage in der Rangliste sicherlich weit oben – und doch würden Sie sie nicht sofort entdecken, weil sie im Plancache insgesamt 1.000.000 Mal mit jeweils zehn Leseoperationen enthalten ist. Falls Sie also nur auf die Leseoperationen schauen, entgeht Ihnen, dass diese Abfrage – oder besser gesagt der Abfragetyp – die Ursache für Performance-Probleme darstellen kann.

Wir benötigen also eine Möglichkeit, Abfragen desselben Typs zusammenzufassen. Zu diesem Zweck kann die Spalte `query_hash` der dynamischen Verwaltungssicht `sys.dm_exec_query_stats` verwendet werden. Der Optimierer bestimmt diesen Hash-Wert für jede Abfrage. Abfragen, die dasselbe Muster besitzen und sich nur in Literalwerten unterscheiden, haben einen identischen `query_hash`-Wert. Sie können also einfach die Leseoperationen je `query_hash`-Wert summieren und außerdem die Anzahl der im Plancache vorhandenen Einträge für einen `query_hash`-Wert bestimmen. Wird dieses Ergebnis nach E/A-Vorgängen sortiert, erhalten Sie die Abfrage, welche in Summe die meiste E/A-Belastung erzeugt.

Schauen wir uns noch einmal das Beispiel aus Kapitel 9 an, in dem Abfragen innerhalb einer Schleife in nicht parametrisierter Form ausgeführt werden:

```
set nocount on
dbcc freeproccache
dbcc dropcleanbuffers
go
declare @i int
       ,@cmd nvarchar(200)
set @i = 1
while (@i <= 1000)
 begin
   set @cmd = 'declare @x int;
               select @x=checksum_agg(checksum(*))
```

```
                from sys.all_columns
               where object_id=' + cast(@i as nvarchar(30))
   exec (@cmd)
   set @i = @i + 1
 end
go
```

Nach der Ausführung des obigen T-SQL-Skripts werden im Plancache 1.000 nahezu identische Pläne gespeichert sein. Diese Pläne haben alle denselben Wert für `query_hash`, denn sie unterscheiden sich lediglich in dem für die `WHERE`-Klausel verwendeten Vergleichswert. Mit der folgenden Abfrage können Sie eine solche Situation herausfinden:

```
select query_hash
      ,count(*) as Anzahl
      ,sum(total_worker_time) as CPU
      ,sum(total_elapsed_time) as Dauer
      ,sum(total_logical_reads) as Gelesen
      ,count(distinct query_plan_hash) as UnterschiedlichePlaene
  from sys.dm_exec_query_stats as qs
 group by query_hash
 order by Anzahl desc
```

Ein Beispielergebnis der obigen Abfrage sehen Sie in Abbildung 10.2.

query_hash	Anzahl	CPU	Dauer	Gelesen	UnterschiedlichePlaene
0xFAB427FA2B1B11BA	1000	248000	378006	4012	1

Abbildung 10.2: *Abfragen mit identischem »query_hash«*

Sie können dort sofort ablesen, wie viel CPU-Last die Abfragen insgesamt erzeugt haben oder wie viele Leseoperationen erforderlich waren. Über den zurückgegebenen Wert für `query_hash` können Sie dann den Plancache weiter untersuchen und feststellen, zu welcher Abfrage der Hash-Wert gehört. Hierzu verwenden Sie die Abfrage des Plancache aus Kapitel 9 und filtern dort einfach nach einem bestimmten Wert für `query_hash`.

Sicherlich ist Ihnen bereits aufgefallen, dass die obige Abfrage auch eine Auswertung der Spalte `query_plan_hash` vornimmt. In dieser Spalte steht ebenfalls ein Hash-Wert, diesmal aber nicht für den Abfragetext, sondern für den erstellten Ausführungsplan. Immerhin ist es ja möglich, dass für eine Abfrage eines bestimmten Typs – also für Abfragen mit demselben `query_hash`-Wert – unterschiedliche Ausführungspläne erzeugt werden, da unterschiedliche Literalwerte zu unterschiedlichen Kardinalitätsschätzungen und damit letztlich auch zu unterschiedlichen physischen Operatoren führen. Dadurch können in `sys.dm_exec_query_stats` unterschiedliche Werte für `query_plan_hash` für denselben Wert von `query_hash` existieren. In unserem Beispiel ist dies nicht der Fall – und diese Information ist sehr wertvoll.

Wir wissen durch das in Abbildung 10.2 präsentierte Ergebnis, dass für alle Abfragen mit dem in der ersten Spalte dargestellten Hash-Wert ein identischer Ausführungsplan verwendet wurde. Diese Information steht in der letzten Spalte. Weiter können wir ablesen, dass die Abfrage insgesamt 1.000 Mal im Plancache existiert. Wenn Sie sich an das in Kapitel 9 Gesagte erinnern, erkennen Sie sofort, dass an dieser Stelle ein Problem mit der Nicht-Parametrisierung dieser Abfrage besteht. Die Abfrage mit dem in der ersten Spalte

dargestellten Hash-Wert sollte parametrisiert werden, um das System insgesamt zu entlasten. Hierzu können Sie entweder den Code der Anwendung anpassen, indem Sie zum Beispiel die gespeicherte Systemprozedur `sp_executesql` einsetzen oder auch parametrisierte Abfragen durch die Verwendung des ADO.NET-`Parameter`-Objekts erzeugen, so wie in Kapitel 9 gezeigt. Wenn Sie diese Möglichkeiten nicht haben, weil Sie den Code der Anwendung nicht ändern können, hilft Ihnen die Erstellung einer Planhinweisliste des Typs TEMPLATE, mit der Sie eine Parametrisierung für die entsprechende Abfrage erzwingen können.

Die Spalten `query_hash` und `query_plan_hash` sind auch nützlich, um herauszufinden, in welchen Fällen eine Parametrisierung die Abfrageleistung negativ beeinträchtigt. Wenn für beide Spalten ein Wert nur einmal auftritt, so gibt es auch nur einen Abfragetext und einen Ausführungsplan. Damit ist es wahrscheinlich, dass eine solche Abfrage parametrisiert wurde. Untersuchen Sie für diese Pläne die Spalten `execution_count`, `min_logical_reads` und `max_logical_reads`. Wenn der Wert für `execution_count` größer als 1 ist und sich die Spaltenwerte für `min_logical_reads` und `max_logical_reads` stark unterscheiden, haben Sie möglicherweise eine Abfrage entdeckt, die parametrisiert wurde, obwohl sie von einer Parametrisierung nicht profitiert.

Die folgende Anweisung findet solche Abfragen heraus:

```
select text
      ,query_hash, query_plan_hash
      ,execution_count
      ,min(min_logical_reads) as min_logical_reads
      ,max(max_logical_reads) as max_logical_reads
  from sys.dm_exec_query_stats
```

Die Abfrage sucht diejenigen Zeilen heraus, bei denen der Wert von `max_logical_reads` mehr als zehn Mal so groß ist wie der Wert von `min_logical_reads`. Passen Sie die `WHERE`-Klausel dementsprechend an, um diese Bedingung zu ändern; etwa wenn Sie zu Beginn einer Analyse erst einmal nach größeren Unterschieden in beiden Werten suchen möchten.

10.1.4 Permanentes Speichern der Informationen aus dynamischen Verwaltungssichten

Wie bereits mehrfach erwähnt, haben dynamische Verwaltungssichten den Nachteil, dass die in ihnen gespeicherte Information flüchtig ist und daher spätestens bei dem nächsten Neustart von SQL Server verloren geht. Es gibt auch noch einen weiteren Nachteil: Die von dynamischen Verwaltungssichten zurückgelieferte Information präsentiert stets eine Momentaufnahme des entsprechenden Systemzustands zum Zeitpunkt der Abfrage der dynamischen Verwaltungssicht. So erhalten Sie zum Beispiel bei der Abfrage der Sicht `sys.dm_io_virtual_file_stats` stets die kumulierte Information über die E/A-Operationen je Datenbank seit dem letzten Start von SQL Server. Eine Information über den zeitlichen Verlauf der E/A-Operationen, etwa nach Tagen oder Stunden, erhalten Sie durch eine Abfrage dieser Sicht nicht.

In vielen Fällen ist aber gerade diese Information sehr wesentlich. Oftmals ist es durchaus interessant, zum Beispiel den zeitlichen Verlauf der E/A-Operationen über einen Tag hinweg zu beobachten, wobei etwa zu jeder vollen Stunde ein Snapshot erstellt wird. Falls diese Information für Sie von Bedeutung ist, Sie also den zeitlichen Verlauf bestimmter Indikatoren darstellen möchten, dann müssen Sie in zyklischen Abständen Momentaufnahmen der entsprechenden Verwaltungssichten archivieren. Ich möchte Ihnen hierzu ein Beispiel für die E/A-Vorgänge präsentieren, das Sie jedoch auch auf andere dynamische Verwaltungssichten anwenden können.

Die folgende Abfrage gibt die seit dem letzten Start von SQL Server durchgeführten E/A-Operationen je Datenbank zurück:

```
select db_name(mf.database_id) as DbName
      ,case mf.type when 0 then 'data' else 'log' end as FileType
      ,sum(fs.num_of_bytes_written)/1024.0/1024.0 as MBWritten
      ,sum(fs.num_of_bytes_read)/1024.0/1024.0 as MBRead
  from sys.dm_io_virtual_file_stats(null,null) as fs
       inner join sys.master_files as mf
               on mf.file_id = fs.file_id
              and mf.database_id = fs.database_id
 group by mf.database_id, mf.type
```

Falls diese Information permanent gespeichert werden soll, kann das Ergebnis der obigen Abfrage in eine Tabelle geschrieben werden. Diese Tabelle muss neben der eigentlich von der Abfrage zurückgegebenen Information auch eine Identifikation des Snapshots enthalten. Wir nehmen hierfür einfach den Zeitpunkt der Protokollierung und legen die Tabelle in der Datenbank *msdb* wie folgt an:

```
use msdb;
-- Protokolltabelle
create table dbo.FileIO
   (
     RecordedAt datetime not null default getdate()
    ,DbName sysname not null
    ,FileType nvarchar(10) not null
    ,MBWritten decimal(12,2) not null
    ,MBRead decimal(12,2) not null
   )
```

Sie können nun in zyklischen Abständen Daten in diese Tabelle einfügen. Hierzu erstellen Sie zum Beispiel einen SQL Server Agent-Auftrag, der etwa jede Stunde einen Protokolleintrag hinzufügt:

```
insert msdb.dbo.FileIO(DbName, FileType, MBWritten, MBRead)
  select...
```

Wenn Sie nun die protokollierten Snapshots auswerten möchten, müssen Sie die Werte zweier Snapshots voneinander abziehen, um die E/A-Belastung für den Zeitraum zwischen den beiden Snapshots zu erhalten. Hierfür können Sie die folgende Sicht erstellen:

```
use msdb
go
create view dbo.FileIOCourse(SnapshotNo, StartTime, EndTime
                            ,DbName, FileType
                            ,MBWritten, MBRead, MBTotal, PercentIO) as
  with OrderedFileIO as
   (
    select row_number() over(order by DbName,FileType,RecordedAt) as rn
          ,convert(datetime,convert(varchar(16)
          ,RecordedAt,120), 120) as RecordedAt
          ,DbName, FileType
          ,MBWritten, MBRead
      from msdb.dbo.FileIO
   )
   ,DiffFileIO as
   (
    select ss1.RecordedAt as StartTime, ss2.RecordedAt as EndTime
          ,ss1.DbName as DbName, ss1.FileType as FileType
          ,ss2.MBWritten-ss1.MBWritten as MBWritten
          ,ss2.MBRead-ss1.MBRead as MBRead
      from OrderedFileIO as ss1
           inner join OrderedFileIO as ss2
                   on ss2.rn = ss1.rn+1
                  and ss2.DbName = ss1.DbName
                  and ss2.FileType = ss1.FileType
   )
   select dense_rank() over (order by StartTime)
         ,StartTime,EndTime
         ,DbName, FileType
         ,MBWritten, MBRead
         ,MBWritten + MBRead
         ,cast(100.0* (MBWritten + MBRead)
            /sum(MBWritten + MBRead)
                    over (partition by StartTime) as decimal(12,2))
     from DiffFileIO
```

Diese Sicht zieht jeweils die Werte benachbarter Snapshots voneinander ab. Daher können Sie die Sicht verwenden, um den Verlauf der E/A-Operationen über einen Zeitraum hinweg zu beobachten. Um zum Beispiel die Leseoperationen für die Datenbank `QueryTest` zu erhalten, starten Sie einfach die folgende einfache Abfrage:

```
select SnapshotNo, EndTime, MBRead
  from dbo.FileIOCourse
 where DbName = 'QueryTest'
   and FileType = 'data'
 order by SnapshotNo
```

In Abbildung 10.3 sehen Sie einen Ausschnitt des Ergebnisses.

SnapshotNo	EndTime	MBRead
1	2008-10-01 01:00:00.000	0.00
2	2008-10-01 02:00:00.000	5.00
3	2008-10-01 03:00:00.000	3.00
4	2008-10-01 04:00:00.000	0.00
5	2008-10-01 05:00:00.000	2.00
6	2008-10-01 06:00:00.000	2.00
7	2008-10-01 07:00:00.000	9.00
8	2008-10-01 08:00:00.000	20.00
9	2008-10-01 09:00:00.000	21.00
10	2008-10-01 10:00:00.000	47.00

Abbildung 10.3:
Leseoperationen für die Datenbank »QueryTest« über die Zeit hinweg

Natürlich können Sie auch eine entsprechende Sicht erstellen und die in Abbildung 10.3 dargestellten Daten in Excel importieren. Dadurch ist es auf einfache Weise möglich, den Verlauf der E/A-Operationen über die Zeit grafisch darzustellen. Abbildung 10.4 zeigt ein Beispiel.

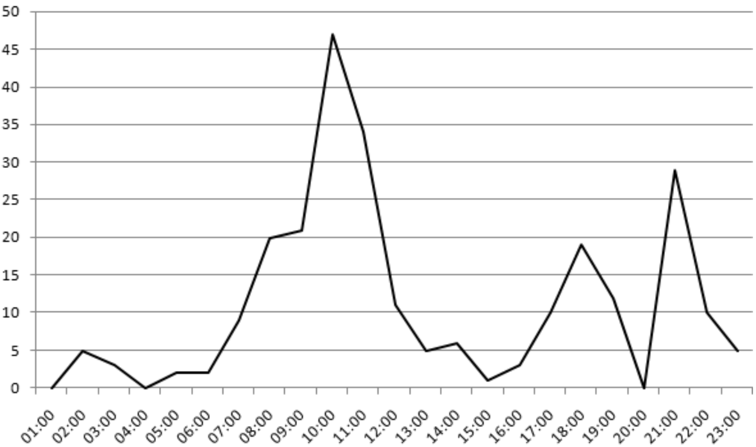


Abbildung 10.4: *E/A-Operationen für eine Datenbank über einen Tag*

Wenn Sie sich den Verlauf ansehen, erkennen Sie sehr schnell, zu welchen Zeitpunkten gerade besonders viel E/A-Last existiert hat. Leider ist es nicht ganz so einfach herauszufinden, welche Abfragen gerade zu diesen Zeitpunkten ausgeführt wurden. Die dynamische Verwaltungssicht `sys.dm_exec_query_stats` hat aber immerhin eine Spalte `last_execution_time`, in der Sie den Zeitpunkt der letzten Ausführung einer Abfrage finden. Das ist allerdings wenig hilfreich, falls Sie wissen möchten, welche Abfragen in der Zeit zwischen 9:00 Uhr und 11:30 Uhr ausgeführt wurden. Hier liegt letztlich das gleiche Problem vor wie mit allen dynamischen Verwaltungssichten: Es gibt jeweils nur kumulierte Informationen und keinen zeitlichen Verlauf. Falls Sie eine Zeitkomponente benötigen, müssen Sie diese selbst hinzufügen, indem Sie Snapshots anlegen – so wie wir es im Beispiel getan haben. Sie können die in Kapitel 4 verwendete Anweisung zur Abfrage des Plancache ebenfalls in zyklischen Abständen ausführen und das Ergebnis in einer Tabelle speichern. Gehen Sie hierbei genau so vor, wie wir es im Beispiel für die E/A-Operationen getan haben. Die Spalte `last_execution_time` der Tabelle, welche die Snapshots enthält, können

Sie dann benutzen, um zu erfahren, ob die entsprechende Abfrage im protokollierten Zeitraum ausgeführt wurde.

10.1.5 Berichte

Für die SQL Server-Instanz stehen Ihnen vier Berichte zur Verfügung, mit denen Sie Abfragen herausfinden können, die in der Rangliste der E/A-Operationen oder der CPU-Verwendung die vordersten Plätze einnehmen. Die Namen dieser Berichte beginnen alle mit »Leistung«. Die Berichte ähneln sich in der Darstellung und sehen prinzipiell so aus wie in Abbildung 10.5.

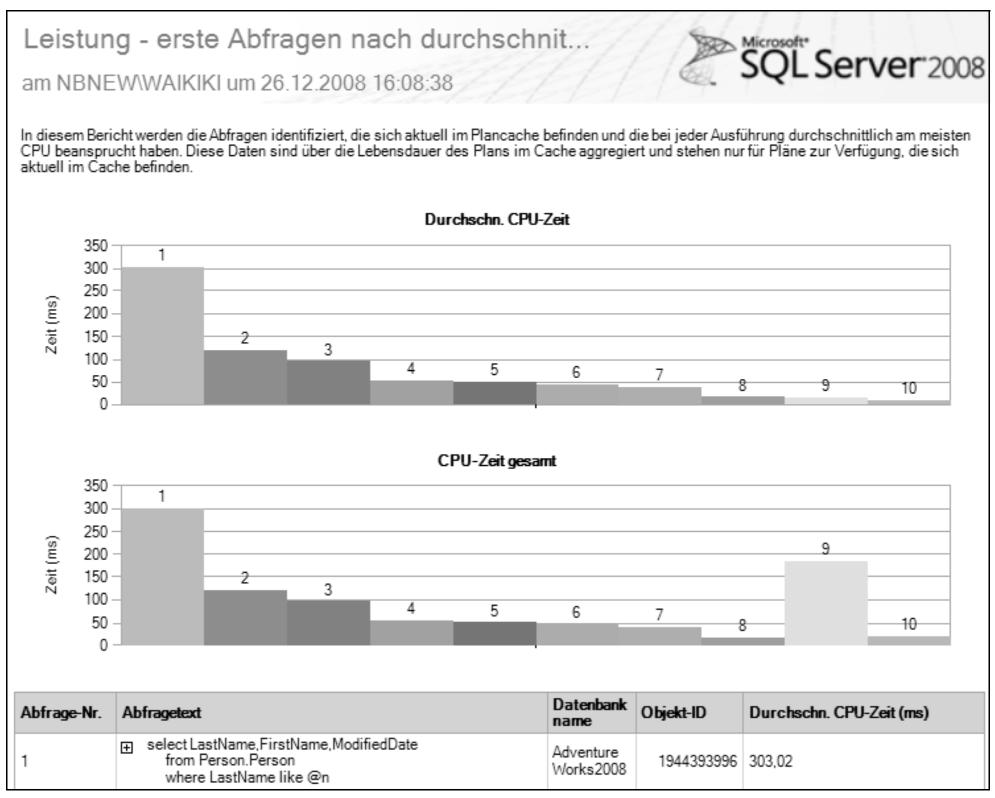


Abbildung 10.5: *Beispiel für einen Bericht zur Abfrageleistung*

Im oberen Bereich finden Sie stets eine grafische Darstellung der Abfragen entsprechend dem ausgewählten Kriterium (CPU oder E/A). Die Tabelle im unteren Bereich enthält dann für jede im Diagramm dargestellte Abfrage die detaillierten Daten und insbesondere die SQL-Anweisung.

Diese Berichte sind ein einfaches und dennoch sehr wirkungsvolles Instrument für das Aufspüren möglicherweise problematischer Abfragen. Denken Sie aber bitte daran, dass alle Berichte auf der Basis des Plancache erstellt werden. Sobald eine Abfrage aus dem

Cache entfernt wird, wird sie in keinem Bericht mehr auftauchen – auch dann nicht, wenn sie eigentlich in der Rangliste weit oben stehen müsste.

Sie können aus den folgenden vier Berichten wählen:

- **Leistung – erste Abfragen nach durchschnittlicher CPU-Zeit**

  Dieser Bericht enthält die Abfragen, die bei jeder Ausführung im Durchschnitt die meiste CPU-Zeit benötigt haben.

- **Leistung – erste Abfragen nach durchschnittlicher EA**

  In diesem Bericht finden Sie Abfragen, die bei jeder Ausführung durchschnittlich die meisten E/A-Operationen verursacht haben.

- **Leistung – erste Abfragen nach CPU gesamt**

  An dieser Stelle stehen Abfragen, die bei allen Ausführungen insgesamt die meiste CPU-Zeit benötigt haben.

- **Leistung – erste Abfragen nach EA gesamt**

  Hier finden Sie Abfragen, die bei allen Ausführungen die meisten E/A-Operationen benötigt haben.

10.2 Arbeiten mit dem Profiler

Eine umfassende Einführung in die Verwendung des SQL Server Profilers haben Sie bereits in Kapitel 4 erhalten. In diesem Abschnitt sollen die Ausführungen aus Kapitel 4 nun angewendet werden. Wir werden eine serverseitige Ablaufverfolgung erstellen und diese Ablaufverfolgung anschließend auswerten, um problematische Abfragen aufzuspüren.

Sie können hierfür eine Ablaufverfolgung unter Verwendung der Vorlage *Tuning* konfigurieren. Fügen Sie bitte noch die Ereignisspalte *Reads* zu den Ereignissen *Stored Procedures/RPC:Completed*, *Stored Procedures/SP:StmtCompleted* und *TSQL/SQL:BatchCompleted* hinzu, denn diese Spalte ist in der Vorlage nicht enthalten. Ebenso sollten Sie die Spalte *RowCounts* zu den Ereignissen hinzufügen. Sie werden etwas weiter unten gleich sehen, warum. Nützlich sind auch die folgenden beiden Ereignisse:

- *Scans/Scan:Stopped* mit der Ereignisspalte *Reads*. Falls in einer Abfrage ein Tabellen- oder Index-Scan ausgeführt wird, wird dieses Ereignis ausgelöst. Die Spalte *Reads* enthält die Information über die erforderlichen logischen Lesevorgänge.
- *Errors and Warnings/Sort Warnings*, zum Beispiel mit der Ereignisspalte *StartTime*. Wenn für eine Sortierung eine *Worktable* benötigt wird, finden Sie dieses Ereignis in Ihrer Ablaufverfolgung. Die Erzeugung dieser temporären Tabelle (der *Worktable*) kann oft durch einen geeigneten Index vermieden werden. Sie sollten daher versuchen, diejenige Abfrage zu finden, welche eine solche temporäre Tabelle verwendet, und eventuell einen entsprechenden Index erstellen.

Lassen Sie sich für Ihre konfigurierte Ablaufverfolgung ein Skript für die Erstellung einer serverseitigen Ablaufverfolgung erzeugen, so wie in Kapitel 4 gezeigt.

Wenn Sie die Ablaufverfolgung anschließend starten (als serverseitige Ablaufverfolgung durch Ausführen des erzeugten Skripts!), werden die Ereignisdaten in der angegebenen Datei gespeichert. Sobald Sie der Meinung sind, dass die in der Datei protokollierten Ereignisse für eine Analyse ausreichend sind, beenden Sie die Ablaufverfolgung. Wenn Sie nicht mehr wissen, wie Sie hierfür vorgehen müssen, lesen Sie bitte noch einmal in Kapitel 4 nach. Bitte bedenken Sie, dass die Ablaufverfolgung selber ebenfalls Ressourcen benötigt, und dass die erzeugte Ablaufverfolgungsdatei sehr groß werden kann. Lassen Sie die Protokollierung also bitte nur so lange laufen wie unbedingt nötig.

Es existiert jetzt also eine Protokolldatei, in der alle relevanten Ereignisse aufgezeichnet wurden. Die Frage ist, wie Sie mit Hilfe dieser Datei problematische Abfragen herausfinden.

Hier kommt die Funktion `fn_trace_gettable()` zum Einsatz. Wie bereits in Kapitel 4 erwähnt, können Sie mit dieser Funktion eine Ablaufverfolgungsdatei in eine Tabelle überführen. Und genau dies soll unser erster Schritt bei der Auswertung der Protokolldaten sein. Die folgende Abfrage speichert die Daten einer oder mehrerer Ablaufverfolgungsdatei(en) in einer Tabelle:

```
use QueryTest;
select *
  into TraceData
  from fn_trace_gettable('c:\SqlTrace\TestDaten.trc', default)
```

Für die Auswertung dieser Tabelle stehen Ihnen anschließend alle Möglichkeiten von T-SQL zur Verfügung. Die folgende Anweisung sortiert zum Beispiel die im Protokoll enthaltenen Abfragen nach ihrer Dauer:

```
select TextData
      ,cast(Duration * 0.001 as decimal(8,1)) as [Duration/ms]
      ,RowCounts
 from TraceData
where isnull(Duration, 0) > 500
order by Duration desc
```

Ein Beispielergebnis der Abfrage sehen Sie in Abbildung 10.6.

TextData	Duration/ms	RowCounts
select OrderDate, OrderQty from Bestellung ...	14180.8	1
select OrderDate from Bestellung where S...	11.0	1
select ShipDate, TotalDue from Bestellung ...	7.0	1946

Abbildung 10.6: *Protokollierte Abfragen, absteigend sortiert nach der Dauer*

Bei der Auswertung der Tabelle mit den Ablaufverfolgungsdaten sind Ihrer Phantasie keine Grenzen gesetzt. Sie können auch Indizes für diese Tabelle erstellen, um entsprechende Abfragen zu beschleunigen. Denken Sie bitte daran, dass in der Ablaufverfolgung eine Spalte `DatabaseID` enthalten ist, über die Sie die betroffene Datenbank herausfinden können. Damit haben Sie eine Verbindung zu den E/A-Operationen je Datenbank (siehe zum Beispiel Abschnitt 10.1.1), da Sie gezielt diejenige Abfrage für eine Datenbank herausfiltern können, die in der Rangliste der E/A-Operationen an vorderster Stelle steht.

Insbesondere sollten Sie nach folgenden Abfragen suchen:

- **Lange Ausführungsdauer**. In OLTP-Systemen sollte die Dauer einer Abfrage normalerweise unterhalb von drei Sekunden liegen. Untersuchen Sie Ihre »Langläufer« und finden Sie heraus, warum eine Abfrage lange benötigt, um ein Resultat zurückzuliefern. Möglicherweise fehlt ein Index, oder es werden einfach zu viele Zeilen zurückgeliefert.
- **Viele zurückgelieferte Zeilen**. Transaktionen in OLTP-Systemen sind kurz und bearbeiten nur wenige Zeilen. Wenn eine Abfrage viele Zeilen zurückliefert, liegt möglicherweise ein Problem mit der Anwendungslogik vor, zum Beispiel weil Verknüpfungen (JOIN-Operationen) erst auf dem Client ausgeführt werden.
- **Viele Lesevorgänge bei nur wenig zurückgelieferten Zeilen**. Eine Abfrage, die 1.000.000 Leseoperationen benötigt, um zwei Zeilen zurückzuliefern, ist mit Sicherheit problematisch. Die Ursachen können vielfältig sein – vom fehlenden Index bis hin zu einem nicht optimalen Datenbankschema.
- **Scan-Operationen**. Falls in Ihrer Ablaufverfolgungsdatei Scan-Operationen enthalten sind, sollten Sie die Ursache herausfinden. Scans haben in OLTP-Anwendungen normalerweise nichts zu suchen. Das Ereignis *Scan:Stopped* finden Sie über die Spalte `EventClass`. Der Wert 52 steht für dieses Ereignis. Leider wird aber der Abfragetext nicht in der Spalte `TextData` angezeigt, sodass es nicht ganz einfach ist, den Bezug zu einer Abfrage herauszufinden.
- **Sort Warnings**. Dieses Ereignis zeigt an, dass Sortiervorgänge nicht im Arbeitsspeicher ausgeführt werden konnten. Die Spalte `EventClass` hat für eine `Sort Warning` den Wert 69. Auch hier wird der Abfragetext nicht in der Spalte `TextData` ausgegeben, was die Zuordnung der *Sort Warning* zu einer Abfrage erschwert.

Leider gibt es keine Ereignisspalten, die eine Gruppierung ähnlicher Abfragen gestatten, so wie dies bei der Abfrage des Plancache durch die Spalten `query_hash` und `query_plan_hash` möglich ist. Um gleiche Abfragen, die sich nur in Literalwerten oder Parametern unterscheiden, aufzufinden müssen Sie die in der Spalte `TextData` enthaltenen Abfragetexte entsprechend bearbeiten oder »umrechnen«. Eine Möglichkeit wäre die Bearbeitung der Ablaufverfolgungstabelle in einem Cursor. Sie können dann die gespeicherte Systemprozedur `sp_get_query_template` für jede Zeile aufrufen, um den in der Spalte `TextData` enthaltenen Abfragetext zu standardisieren beziehungsweise in seine parametrisierte Form zu überführen. Speichern Sie bitte den erhaltenen Wert in einer neuen Spalte für spätere Betrachtungen.

Bequemer ist die Verwendung einer Funktion, welche den in der Spalte `TextData` enthaltenen Abfragetext in eine parametrisierte Form überführt. Eine solche Funktion wurde von Stuart Ozer vom Microsoft SQL Server Customer Advisory Team entwickelt und veröffentlicht. Ich darf Ihnen den Quelltext dieser Funktion aus lizenzrechtlichen Gründen hier leider nicht präsentieren. Sie können die Funktion jedoch relativ leicht im Internet finden. In [1], S. 93, ist der Quelltext ebenfalls abgedruckt.

10.3 Einsatz von Datenauflistungen

Datenauflistungen sind eine neue und sehr bequeme Möglichkeit der Protokollierung und Auswertung von Abfragestatistiken. In Kapitel 4 haben Sie einen ersten Einblick in die Konfiguration eines Verwaltungs-Data Warehouse (VDWH) erhalten und auch gesehen, wie Sie die systemeigenen Datenauflistsätze verwenden können, um E/A-Operationen oder Abfragen zu beobachten. Ich kann Ihnen nur noch einmal empfehlen, sich mit diesem Thema auseinanderzusetzen und Datenauflistungen zum Aufzeichnen und Auswerten einzusetzen.

Für das Auffinden problematischer Abfragen gibt es bereits einen vordefinierten Systemdaten-Auflistsatz, den Sie nach der Konfiguration des VDWH lediglich starten müssen. Dies ist der Auflistsatz *Abfragestatistik*. Sobald dieser Auflistsatz ausgeführt wird, werden im Abstand von zehn Sekunden Abfragestatistiken ermittelt und im lokalen VDWH-Cache-Verzeichnis gespeichert. Das Hochladen dieser zwischengespeicherten Daten erfolgt dann alle 15 Minuten. Dies ist die Standardkonfiguration, die Sie natürlich verändern und an Ihre Erfordernisse anpassen können. Eine entsprechende Anpassung der Zeitpläne ist leider (noch?) nicht mit dem SQL Server Management Studio möglich. Hierzu müssen Sie die gespeicherten Prozeduren zur Verwaltung von Datenauflistungen verwenden. Insgesamt stehen Ihnen zwanzig solcher Prozeduren zur Verfügung, deren Name stets mit `sp_syscollector_` beginnt. Mit diesen Prozeduren können Sie zum Beispiel eigene Datenauflistungen einrichten, Datenauflistungen starten oder anhalten und eben auch Konfigurationsoptionen einstellen. Um zum Beispiel für den Auflistsatz *Abfragestatistik* die Frequenz der Aufzeichnung auf 30 Sekunden einzustellen, verwenden Sie die Prozedur `sp_syscollector_update_collection_item`:

```
use msdb;
exec sp_syscollector_update_collection_item
          @name = N'Abfragestatistik - Abfrageaktivität'
         ,@frequency = 30;
```

Merken Sie sich bitte, dass protokollierte Daten nicht unmittelbar in das VDWH übertragen werden, sondern erst nach einer gewissen Zeitspanne zur Verfügung stehen. So kann es eben vorkommen, dass eine Abfrage, die in der Rangliste an erster Stelle stehen müsste, noch nicht in den entsprechenden Berichten auftaucht, weil die zugehörigen Protokolldaten noch nicht in das VDWH hochgeladen wurden.

Die in Kapitel 4 vorgestellten Berichte stellen die einfachste Möglichkeit dar, die im VDWH gespeicherten Daten auszuwerten. Die Berichte für die systemeigenen Auflistsätze ermöglichen hierbei genau die in den ersten Abschnitten des vorliegenden Kapitels beschriebene Vorgehensweise zum Aufspüren problematischer Abfragen. So können Sie zum Beispiel mit dem Bericht über die Datenträgerverwendung je Datenbank beginnen, um zu sehen, auf welcher Datenbank zu welcher Tageszeit die Anzahl der E/A-Operationen besonders hoch gewesen ist. Anschließend können Sie den Bericht über die Abfragestatistik verwenden und so feststellen, welche Abfragen zur fraglichen Zeit ausgeführt wurden und für die E/A-Operationen verantwortlich waren. Leider ist es nicht möglich, in den Berichten über die Abfragestatistiken nach einer bestimmten Datenbank zu filtern, sodass eine entsprechende Suche unter Umständen etwas mühsam ist. Dennoch bietet die Verwendung des VDWH für die Protokollierung und die Auswertung der Protokoll-

daten durch die zur Verfügung stehenden Berichte eine phantastische Möglichkeit, problematische Abfragen aufzuspüren. Die Berichte liefern vielfältige Möglichkeiten zur Navigation, sodass Sie kritische Abfragen sehr leicht entdecken können. So ist es zum Beispiel möglich, die Rangliste der Abfragen wahlweise nach der CPU-Belastung, der Dauer oder der Anzahl der erforderlichen E/A-Operationen zu sortieren. Schauen Sie sich hierzu bitte noch einmal Abbildung 4.10 an. Für alle in der Rangliste auftauchenden Abfragen werden sogar die Ausführungspläne gespeichert, die Sie selbstverständlich ebenfalls ansehen und auswerten können. Hierzu liefert Ihnen der Bericht entsprechende Navigationsmöglichkeiten.

Bitte bedenken Sie, dass die Informationen für alle im Bericht über die Abfragestatistik dargestellten Abfragen letztlich aus dem Plancache geholt werden. Sofern eine Abfrage zum Zeitpunkt der Protokollierung nicht im Plancache gespeichert war, wird sie auch nicht im Bericht auftauchen.

Dadurch kann es natürlich vorkommen, dass Ihnen ressourcenintensive Abfragen verborgen bleiben. Denken Sie daran, dass nicht alle Abfragen im Plancache gespeichert werden, und dass auch Abfragen aus dem Plancache entfernt werden können. Aus diesem Grund ist das standardmäßige Auflistintervall mit zehn Sekunden entsprechend kurz gewählt.

Die Informationen über die Konfiguration von Auflistsätzen werden in der Systemdatenbank *msdb* gespeichert. Hier finden Sie entsprechende Systemsichten und Systemtabellen, deren Name stets mit `syscollector_` beginnt.

Solange Sie nur die vorkonfigurierten Systemdaten-Auflistsätze mit den zugehörigen Berichten verwenden, werden Sie kaum Probleme mit dem VDWH haben. Das VDWH ist jedoch so mächtig und bietet so vielfältige und hervorragende Möglichkeiten, dass Sie mit Sicherheit auch eigene Auflistsätze konfigurieren werden. Etwas weiter unten finden Sie hierfür ein erstes Beispiel. Im nächsten Kapitel werden wir dann eigene Auflistsätze für die Überwachung der Indexverwendung erstellen.

Wenn Sie eigene Datenauflistungen hinzufügen, ist das Hauptproblem die Auswertung der protokollierten Daten. Die Protokollierung selber ist mit ein paar Aufrufen von gespeicherten Prozeduren schnell konfiguriert. Für die Auswertung der Daten müssen Sie natürlich wissen, wo und in welcher Form die entsprechenden Protokolle abgelegt werden. Dies herauszufinden, ist nicht immer ganz einfach. Insbesondere ist die Dokumentation des VDWH-Schemas bislang nur sehr dürftig. Dies kann natürlich daran liegen, dass das VDWH eine Neuerung in SQL Server 2008 ist, für die eine Dokumentation noch nicht vollständig erstellt und ausgeliefert wurde. Möglicherweise werden die Interna auch ganz bewusst nicht dokumentiert, weil davon ausgegangen wird, dass sich der interne Aufbau mit der Zeit ändern wird. In unseren Beispielen kommen wir hierauf noch einmal zurück.

Schade ist es allemal, denn im VDWH existieren zum Beispiel eine Reihe gespeicherter Prozeduren, die ganz offensichtlich die Daten für Berichte bereitstellen bzw. aufarbeiten. In der Dokumentation suchen Sie allerdings vergeblich nach entsprechenden Erklärungen dieser Prozeduren.

10.3.1 Manuelle Abfragen des VDWH

Die unzureichende Dokumentation ist auch dann hinderlich, wenn Sie die Protokolle der Systemdaten-Auflistsätze abfragen möchten, ohne die Berichte zu verwenden. Möglicherweise wollen Sie eigene Berichte, beispielsweise mit Excel, entwerfen oder einfach die Protokolldaten zu bestimmten Zeitpunkten archivieren. Manchmal ist es auch hinderlich, dass man erst durch einen Bericht »hindurchnavigieren« muss, um an eine entsprechende Information zu gelangen.

In diesen Fällen ist es also erforderlich, direkt auf die entsprechenden Protokolltabellen zuzugreifen. Damit Sie sich hier nicht genauso mühsam durchkämpfen müssen wie ich, möchte ich Ihnen zumindest einige Abfragen präsentieren, welche die Daten der Systemdaten-Auflistsätze zurückliefern.

Bitte bedenken Sie jedoch, dass diese Abfragen auf nicht dokumentierten Informationen basieren. Dadurch ist es nicht sicher, dass etwa nach einem Versionswechsel oder dem Einspielen eines Service Packs alles weiterhin problemlos funktioniert. Eventuell gibt es auch elegantere Möglichkeiten, an die entsprechenden Daten zu kommen, als im Folgenden präsentiert. Diese Möglichkeiten bleiben aber ohne eine entsprechende Dokumentation leider verborgen.

Alle hier präsentierten Abfragen geben auch eine Snapshot-ID und den Zeitpunkt der Protokollierung – also der Erstellung des Snapshots – zurück. Diese beiden Spalten können Sie verwenden, um den zeitlichen Verlauf darzustellen. Hierzu gehen Sie genau so vor, wie in Abschnitt 10.1.4 bei der Erstellung der Sicht `FileIOCourse` demonstriert. Beachten Sie aber bitte, dass die Snapshot-ID nicht fortlaufend nummeriert ist. Es ist durchaus üblich, dass Lücken in der Nummerierung vorkommen.

Abfrage der Abfragestatistik

Die Tabellen `snapshots.notable_query_plan`, `snapshots.notable_query_text` und `snapshots.query_stats` enthalten die Informationen über Abfragestatistiken. Diese Tabellen entsprechen im Prinzip den dynamischen Verwaltungssichten `sys.dm_exec_cached_plans`, `sys.dm_exec_sql_text` und `sys.dm_exec_query_stats`. Wir können deshalb unsere bereits bekannte Abfrage zur Ermittlung der Abfragestatistiken fast unverändert auch für das VDWH verwenden, indem wir einfach die Namen der dynamischen Verwaltungssichten durch die Tabellennamen des VDWH ersetzen. Eine entsprechende Abfrage sieht dann zum Beispiel wie folgt aus:

```
select qs.snapshot_id, qs.collection_time
      ,qt.sql_text as sql_text
      ,replace(replace(substring(qt.sql_text, qs.statement_start_offset/2+1,
               case
                 when qs.statement_end_offset = -1 then
                      len(convert(nvarchar(max), qt.sql_text))
                 else qs.statement_end_offset/2
                     - qs.statement_start_offset/2+1
               end), char(13), ' '), char(10), ' ') as stmt_text
      ,cast(query_plan as xml) as query_plan
      ,qs.creation_time
```

```
      ,qs.last_execution_time
      ,qs.execution_count
      ,qs.total_worker_time
      ,qs.total_physical_reads
      ,qs.total_logical_reads
      ,qs.total_elapsed_time
  from snapshots.notable_query_plan as qp
       inner join snapshots.notable_query_text as qt
               on qt.sql_handle = qt.sql_handle
       inner join snapshots.query_stats as qs
               on qs.sql_handle = qt.sql_handle
              and qs.sql_handle = qp.sql_handle
```

Selbstverständlich können Sie nach bestimmten Zeiträumen filtern und das Ergebnis zum Beispiel nach der Ausführungsdauer oder den E/A-Operationen sortieren und so die Abfragen herausfinden, die in einem bestimmten Zeitraum in Bezug auf die Verwendung von Ressourcen bestimmend waren.

Die Spalten `query_hash` und `query_plan_hash` stehen leider nicht zur Verfügung. Warum dies so ist, kann ich Ihnen nicht sagen, aber wahrscheinlich wurden sie einfach nur vergessen. Das ist natürlich schade, weil dadurch eine Untersuchung nach Problemen mit Parametrisierung erschwert wird.

Seien Sie bitte vorsichtig mit der obigen Abfrage, denn sie gibt auch den Ausführungsplan im XML-Format zurück. Dadurch werden unter Umständen sehr viele Daten übertragen. Besser wäre es, zunächst nur die Abfragen mit der höchsten Ressourcenverwendung zu ermitteln, ohne den Ausführungsplan mit abzufragen. Anschließend können Sie dann für die in der Rangliste an der Spitze stehenden Abfragen die Ausführungspläne gesondert ermitteln. Ansonsten kann es leicht passieren, dass die obige Abfrage selber in der Abfragestatistik an vorderster Stelle auftaucht, wenn Sie beispielsweise nach erforderlichen E/A-Operationen sortieren.

Abfrage der Wartezustände

Die Protokollierung der aufgetretenen Wartezustände ist Bestandteil des Systemdaten-Auflistsatzes zur Serveraktivität. Auch in diesem Fall ist es so, dass für die dynamische Verwaltungssicht `sys.dm_os_wait_stats` ein entsprechendes Pendant im VDWH existiert: `snapshots.os_wait_stats`. Daher ist die Abfrage der aufgetretenen Wartezustände recht einfach:

```
select * from snapshots.os_wait_stats
```

Abfrage der E/A-Operationen

Das Pendent zur dynamischen Verwaltungssicht `sys.dm_io_virtual_file_stats` heißt `snapshots.io_virtual_file_stats`. Daher ist auch die Abfrage der E/A-Operationen je Datenbank sehr einfach:

```
select * from snapshots.io_virtual_file_stats
```

Abfrage der Leistungsindikatoren

Die Abfrage der protokollierten Leistungsindikatoren erfolgt über `snapshots.performance_counters`, das Gegenstück zu `sys.dm_os_performance_counters`:

```
select * from snapshots.performance_counters
```

`snapshots.performance_counters` ist keine Tabelle, sondern eine Sicht.

Sie merken sicherlich bereits, dass das System, welches der Namensgebung der Tabellen und Sichten im VDWH zugrunde liegt, eine recht unkomplizierte Abfrage der protokollierten Daten ermöglicht. Falls Sie sich ein wenig mit dynamischen Verwaltungssichten auskennen, haben Sie sicherlich keine Probleme, die entsprechenden Tabellen bzw. Sichten im VDWH zu finden. Die einzige Herausforderung ist die Darstellung des zeitlichen Verlaufs. Genau hierfür haben Sie in Abschnitt 10.1.4 aber bereits eine mögliche Vorgehensweise kennengelernt.

10.3.2 Erzeugen von Ablaufverfolgungen mit dem Datenauflister

Wie bereits erwähnt, ist es auch möglich, über die Verwendung der systemeigenen Auflistsätze hinaus, eigene Auflistsätze zu erstellen. Ein erstes und einfaches Beispiel möchte ich Ihnen in diesem Abschnitt präsentieren. Im folgenden Kapitel werden wir dann weitere Auflistsätze erzeugen.

Der Datenauflister kennt auch einen sogenannten *SQL-Ablaufverfolgungs-Auflistertyp*. Dadurch ist es möglich, Ablaufverfolgungen im VDWH zu speichern. Eine entsprechende Vorgehensweise möchte ich Ihnen zum Abschluss dieses Kapitels präsentieren.

Prinzipiell werden die benutzerdefinierten Auflistsätze durch den Aufruf gespeicherter Systemprozeduren konfiguriert. Dies sind eben genau diejenigen Prozeduren, die etwas weiter oben bereits erwähnt wurden. Für die Konfiguration einer Ablaufverfolgung, die im VDWH gespeichert wird, können Sie also ein entsprechendes T-SQL-Skript erstellen und ausführen.

Es geht allerdings auch einfacher, wenn Sie den SQL Server Profiler verwenden. Der Profiler bietet nämlich die Möglichkeit, ein entsprechendes Skript für die Konfiguration einer Datenauflistung zu erstellen. Konfigurieren Sie einfach eine Ablaufverfolgung mit den Sie interessierenden Ereignissen und Ereignisspalten. Sobald diese Konfiguration abgeschlossen ist, wählen Sie bitte aus dem Menü des Profilers den Eintrag DATEI • EXPORTIEREN • SKRIPT FÜR ABLAUFVERFOLGUNGSDEFINITION ERSTELLEN • FÜR DEN AUFLISTSATZ DER SQL-ABLAUFVERFOLGUNG(siehe Abbildung 10.7).

Speichern Sie anschließend das Skript und öffnen Sie dann im Management Studio eine Abfrage mit dem Skript.

Sehen Sie sich das Skript an, werden Sie sicherlich erfreut darüber sein, welch enorme Arbeit Ihnen der Profiler hier abnimmt. Die Definition einer entsprechenden Datenauflistung muss in einem speziellen XML-Format erfolgen. Falls Sie diese Definition manuell erstellen, ist der hierfür erforderliche Aufwand sicherlich nicht unerheblich. Im nächsten Kapitel werden wir einen entsprechenden Auflistsatz manuell erstellen. Dann werden Sie verstehen, wovon ich rede.

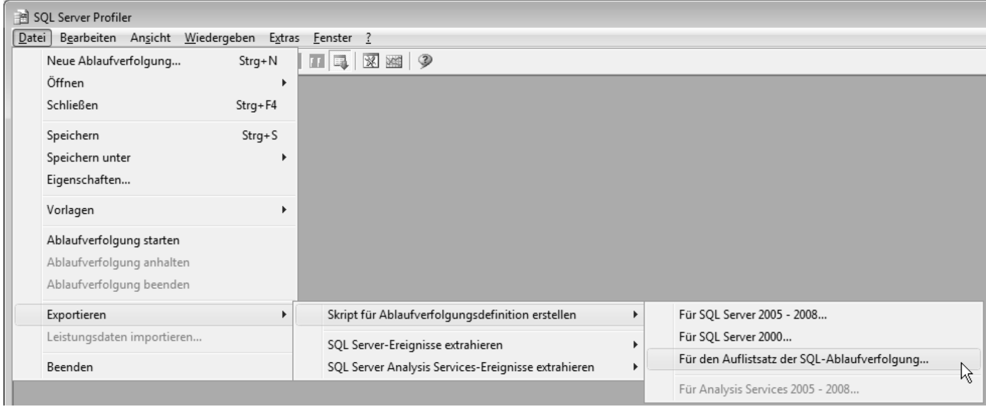


Abbildung 10.7: *Skript für einen Auflistsatz der Ablaufverfolgung erstellen*

Bevor das Skript ausgeführt werden kann, sollten Sie noch die Namen des erzeugten Auflistsatzes und des Auflistelements anpassen. Durchsuchen Sie bitte das Skript nach den Zeichenfolgen »SqlTrace Collection Set Name Here« und »SqlTrace Collection Item Name Here« und ersetzen Sie die Bezeichnungen durch »vernünftige« Namen. Anschließend können Sie das Skript ausführen.

War die Ausführung erfolgreich, sehen Sie Ihren konfigurierten Auflistsatz im Ordner VERWALTUNG/DATENAUFLISTUNG des Objekt-Explorers (siehe Abbildung 10.8). Bei der Ausführung des Skripts werden die IDs des erzeugten Auflistsatzes und des Auflistelements zurückgegeben. Die ID des Auflistelements werden wir etwas weiter unten für die Abfrage der gesammelten Ereignisse noch benötigen.

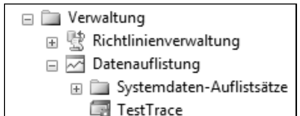


Abbildung 10.8:
Der konfigurierte Auflistsatz für eine SQL-Ablaufverfolgung

Der Auflistsatz wird nach seiner Erzeugung zunächst noch nicht ausgeführt. Sie können den Auflistsatz beispielsweise über das Kontextmenü starten. Sobald der Auflistsatz ausgeführt wird, erfolgt auch die Protokollierung der konfigurierten Ereignisse und Ereignisspalten.

Erkennen Sie die Möglichkeiten, die sich Ihnen dadurch eröffnen? Sie können verschiedene Ablaufverfolgungen konfigurieren und für jede Ablaufverfolgung einen separaten Auflistsatz erzeugen. Dadurch können Sie sich eine Reihe von Auflistsätzen mit Ablaufverfolgungen für ganz unterschiedliche Zwecke erstellen und nach Bedarf starten, um Probleme aufzuspüren. Denken Sie aber bitte auch in diesem Fall daran, dass Ablaufverfolgungen ebenfalls Ressourcen benötigen. Dies gilt natürlich auch dann, wenn Sie eine Ablaufverfolgung als Auflistsatz konfigurieren. Nehmen Sie also bitte nur die wirklich benötigten Ereignisse und Ereignisspalten in die Konfiguration mit auf.

Leider ist die Auswertung der gesammelten Daten in diesem Fall nicht ganz einfach, da keine vordefinierten Berichte existieren. Es bleibt Ihnen also nur die Abfrage der Protokolltabellen. Falls Sie sich mit Reporting Services auskennen, können Sie selbstverständlich auch Ihre eigenen Berichte erstellen und für die Auswertung verwenden.

Die Tabelle `snapshots.trace_data` enthält die Informationen über die protokollierten Ereignisse. Allerdings sind in dieser Tabelle die Daten aller Auflistelemente für Ablaufverfolgungen gespeichert. Falls Sie mehrere solcher Auflistelemente konfiguriert haben, müssen Sie also darauf achten, dass Sie jeweils nur die Ereignisse einer bestimmten Ablaufverfolgung abfragen. Zu diesem Zweck verwenden Sie die ID des Auflistelements, die das Skript zum Anlegen des Elements zurückgegeben hat. Falls Sie diese ID »verloren« haben, können Sie die Sicht `dbo.syscollector_collection_items` der Systemdatenbank *msdb* abfragen, um die ID zu ermitteln:

```
select collection_item_id, name
  from msdb.dbo.syscollector_collection_items
```

Abbildung 10.9 zeigt ein Beispielergebnis der obigen Abfrage.

collection_item_id	name
1	Datenträgerverwendung - Datendateien
2	Datenträgerverwendung - Protokolldateien
3	Serveraktivität - DMV-Snapshots
4	Serveraktivität - Leistungsindikatoren
5	Abfragestatistik - Abfrageaktivität
6	Test Trace Item

Abbildung 10.9:
Existierende Auflistelemente

Den in der Spalte `collection_item_id` zurückgegebenen Wert können Sie dann verwenden, um nur die zu einem bestimmten Auflistelement existierenden Ablaufverfolgungsereignisse abzufragen:

```
select td.*
  from snapshots.trace_data as td
       inner join snapshots.trace_info as ti
               on ti.trace_info_id = td.trace_info_id
 where ti.collection_item_id = 6
```

10.4 Zusammenfassung

Dieses Kapitel hat Ihnen gezeigt, welche Möglichkeiten SQL Server für das Aufspüren problematischer Abfragen zur Verfügung stellt. Sie haben erfahren, wie dynamische Verwaltungssichten eingesetzt werden können, um die E/A-Operationen je Datenbank zu ermitteln und um festzustellen, welche Abfragen für den Großteil der E/A-Operationen verantwortlich zeichnen.

Anschließend sind wir noch einmal darauf eingegangen, wie der Profiler eingesetzt werden kann, um besonders ressourcenintensive Abfragen zu entdecken.

Zum Abschluss dieses Kapitels ist noch einmal die Bedeutung von Datenauflistungen betont worden. Sie haben außerdem gesehen, wie Auflistsätze ausgewertet werden können, und wie Sie Ablaufverfolgungen mit Hilfe von Datenauflistungen erzeugen.

Ganz zum Schluss möchte ich Ihnen noch einmal nahelegen, dass Sie insbesondere die Möglichkeiten des VDWH nutzen sollten, um Ereignisse zur protokollieren und die Protokolle auszuwerten.

11 Optimierung des physischen Datenbankentwurfs

In diesem Kapitel beschäftigen wir uns noch einmal mit physischen Aspekten des Datenbankentwurfs. Wie so oft, steht dabei wieder das Ziel der Minimierung physischer E/A-Operationen im Mittelpunkt.

Die wesentlichen Ansatzpunkte für eine Minimierung physischer E/A-Operationen sind Ihnen bereits bekannt. Zunächst einmal sollten Abfragen so entworfen werden, dass sie nicht mehr Daten bzw. Zeilen als nötig verarbeiten. Ausreichend verfügbarer Hauptspeicher, damit vor allem Leseoperationen möglichst häufig allein durch eine Abfrage des Datencaches erledigt werden können, ist ein weiteres wichtiges Kriterium. Schließlich haben wir in den vorangegangenen Kapiteln immer wieder ein drittes und sehr wesentliches Instrument zur Minimierung von E/A-Operationen untersucht: Indizes.

Wir kommen in diesem Kapitel noch einmal auf Indizes zurück und betrachten weiterführende Konzepte für die Überwachung der Indexverwendung und die Optimierung von Datenzugriffen mit Indizes.

11.1 Indexüberwachung mit Datenauflistungen

Fehlende Indizes können E/A-Operationen um ein Vielfaches in die Höhe treiben. Glücklicherweise protokolliert die SQL Server-Abfrage-Engine Hinweise über fehlende Indizes, und es ist dadurch relativ einfach, solche Indizes aufzuspüren. Sie können hierfür zum Beispiel die `sys.dm_db_missing_index...`-Systemsichten verwenden, so wie in Kapitel 6 gezeigt. Allerdings sind die Informationen über fehlende Indizes flüchtig und gehen spätestens bei einem Neustart der SQL Server-Instanz verloren.

Auch für überflüssige Indizes gilt, dass die in den dynamischen Verwaltungssichten verfügbare Information nicht permanent gespeichert wird. Überflüssige Indizes haben eine negative Auswirkung auf die Abfrageleistung, da sie bei Datenänderungen ebenfalls aktualisiert werden müssen, aber für Such- oder Scanoperationen nie verwendet werden. Die Auswirkung eines überflüssigen Index auf die Performance ist normalerweise nicht so drastisch. Dafür ist ein überflüssiger Index in der Regel schwer zu finden, da es erforderlich ist, das System über einen hinreichend langen Zeitraum zu beobachten. Gerade diese Beobachtung über einen langen Zeitraum macht es unbedingt nötig, die Informationen über überflüssige Indizes dauerhaft zu speichern.

Ein Ansatz zur permanenten Speicherung dieser Informationen wurde bereits in Kapitel 6 angedeutet. An dieser Stelle wollen wir nun ein neues Feature von SQL Server 2008 verwenden, um die Informationen über fehlende und überflüssige Indizes permanent zu speichern: Datenauflistungen. Hierbei werden Sie auch einige neue Informationen über das Arbeiten mit Datenauflistungen im Allgemeinen erhalten.

11.1.1 Ein Auflistsatz für fehlende und überflüssige Indizes

Wenn Sie die Konfiguration für Datenauflistungen durchgeführt haben, können Sie einen Auflistsatz konfigurieren, der dafür sorgt, dass Informationen über die Indexverwendung im VDWH gespeichert werden. Da wir die Daten für unsere Protokollierung durch SQL-Abfragen ermitteln, können wir hierzu den T-SQL-Abfrageauflistertyp verwenden. Dieser Auflistertyp steht Ihnen durch die Konfiguration der Datenauflistungen automatisch zur Verfügung. Für das Erzeugen unseres T-SQL-Auflistsatzes benötigen wir als Erstes die Daten des Auflistertyps. Die folgende Abfrage listet die Daten für die existierenden Auflistertypen auf:

```
select collector_type_uid, name
  from msdb.dbo.syscollector_collector_types
```

Das Ergebnis der Abfrage sehen Sie in Abbildung 11.1. Für das Anlegen unserer Auflistsätze vom Typ T-SQL verwenden wir später den Wert der Spalte `collector_type_uid`.

	collector_type_uid	name
1	302E93D1-3424-4BE7-AA8E-84813ECF2419	Generic T-SQL Query Collector Type
2	0E218CF8-ECB5-417B-B533-D851C0251271	Generic SQL Trace Collector Type
3	14AF3C12-38E6-4155-BD29-F33E7966BA23	Query Activity Collector Type
4	294605DD-21DE-40B2-B20F-F3E170EA1EC3	Performance Counters Collector Type

Abbildung 11.1:
Vorhandene Auflistertypen

Jetzt benötigen wir noch einen Zeitplan, nach dem die Informationen dauerhaft gespeichert werden. Auch hier können wir einen bereits vorhandenen Zeitplan verwenden. Die existierenden Zeitpläne gibt die folgende Abfrage zurück:

```
select schedule_uid, name
  from msdb.dbo.sysschedules_localserver_view
```

Das Ergebnis ist in Abbildung 11.2 zu sehen.

	schedule_uid	name
1	3A750608-DF77-4FB0-AED8-CBC9B9E5E3F7	RunAsSQLAgentServiceStartSchedule
2	57A381B7-4CF3-426D-8BBB-B8DD15D23C0C	CollectorSchedule_Every_5min
3	3853C052-9621-4141-B856-3677FAB4A161	CollectorSchedule_Every_10min
4	C7022AF3-51B8-4011-B159-64C47C88FF70	CollectorSchedule_Every_15min
5	4E0A31B7-F20F-44E3-9000-C299B7551ACB	CollectorSchedule_Every_30min
6	FE3C6B0D-4772-468A-89E7-BD2831605BEB	CollectorSchedule_Every_60min
7	1E4C63F0-5697-4E60-9A48-9B1255D4D1D2	CollectorSchedule_Every_6h
8	92FE2FC2-B3D2-4A64-B161-81548A01F310	syspolicy_purge_history_schedule
9	557AAC79-B6B7-47F3-A9FD-F95E46A565B9	mdw_purge_data_schedule

Abbildung 11.2:
Vorhandene Zeitpläne für Datenauflistungen

Unsere Informationen sollen alle 60 Minuten aufgelistet und in das VDWH hochgeladen werden. Daher können wir den existierenden Zeitplan `CollectorSchedule_Every_60min` verwenden. Wir benötigen die `schedule_uid` für diesen Zeitplan. Natürlich wäre es auch möglich, einen neuen Zeitplan zu erstellen und diesen zu verwenden. Das ist jedoch in unserem Fall nicht erforderlich, weil in den standardmäßig vorhandenen Zeitplänen bereits ein geeigneter Plan existiert.

Nachdem wir über die Informationen für den Auflistertyp und den Zeitplan verfügen, können wir unseren Auflistsatz anlegen. Hierfür gibt es die gespeicherte Prozedur `sp_syscollector_create_collection_set`. Diese Prozedur richtet einen Auflistsatz ein, dem wir dann später noch entsprechende Auflistelemente für die Ermittlung der eigentlichen Daten zuordnen müssen. Der Auflistsatz selber enthält die Informationen zur Frequenz, mit der die Daten aufgelistet und hochgeladen werden sollen. Außerdem geben Sie dort auch an, wie lange gesammelte Daten im VDWH aufgehoben werden sollen. Nach Ablauf der entsprechenden Zeitspanne werden ältere Daten einfach mit neuen Protokollen überschrieben. Anderenfalls würde das VDWH beständig anwachsen und irgendwann die Festplattenkapazität überschreiten. Sie können die Prozedur wie folgt aufrufen, um einen Auflistsatz für fehlende Indizes zu erstellen:

```
use msdb;
declare @collection_set_id int;
exec dbo.sp_syscollector_create_collection_set
         @name = N'Index-Statistiken'
         -- Dies ist die GUID für alle 60 Minuten
        ,@schedule_uid = 'FE3C6B0D-4772-468A-89E7-BD2831605BEB'
        ,@collection_mode = 1  -- Keine Zwischenspeicherung.
                               -- Gleicher Zeitplan für Auflistung und Upload
        ,@days_until_expiration = 180   -- 180 Tage aufheben
        ,@description = N'Speichert Informationen über fehlende
                          und nicht verwendete Indizes.'
        ,@collection_set_id = @collection_set_id output;
```

Die Prozedur bekommt die ID für den zu verwendenden Zeitplan als Parameter. Außerdem geben wir an, dass die Auflistung – also das Speichern der zu protokollierenden Daten im lokalen Cache des VDWH – und das eigentliche Hochladen in das VDWH nicht nach separaten Zeitplänen erfolgen sollen. In unserem Fall sollen die Daten nach demselben Zeitplan, nämlich alle 60 Minuten, aufgelistet und dann sofort hochgeladen werden.

Der so erzeugte Auflistungssatz bekommt eine ID, die wir später für das Anlegen der eigentlichen Auflistungen benötigen. `sp_syscollector_create_collection_set` gibt die ID als Ausgabeparameter zurück. Sie können die IDs der existierenden Auflistsätze auch jederzeit so abfragen:

```
select collection_set_id, description
  from msdb.dbo.syscollector_collection_sets
```

11.1.2 Ein Auflistelement für fehlende Indizes

Wir können zum existierenden Auflistungssatz nun Auflistelemente hinzufügen. Dies erledigen wir durch den Aufruf der gespeicherten Prozedur `sp_syscollector_create_collection_item`. Diese Prozedur erhält als Parameter natürlich die ID des Auflistsatzes. Außerdem erwartet die Prozedur auch die ID des Auflistertyps – in unserem Fall die ID des T-SQL-Abfrageauflistertyps, die wir etwas weiter oben ermittelt haben. Der wichtigste Parameter ist sicherlich die XML-Beschreibung des Auflistelements.

Unser erstes Auflistelement soll Informationen über fehlende Indizes sammeln. Dieses Auflistelement kann durch den folgenden Aufruf der Prozedur `sp_syscollector_create_collection_item` erzeugt werden:

```
declare @definition xml;
declare @collection_item_id int;

select @definition = cast(
      N'<ns:TSQLQueryCollector xmlns:ns="DataCollectorType">
         <Query>
           <Value>select db_name(d.database_id) as db_name
                        ,s.login_time as instance_start_time
                        ,current_timestamp as snapshot_time
                        ,d.statement
                        ,d.equality_columns, d.inequality_columns
                        ,d.included_columns
                        ,gs.avg_total_user_cost
                        ,gs.avg_user_impact
                        ,gs.user_seeks, gs.user_scans
                        ,gs.last_user_seek, gs.last_user_scan
                    from sys.dm_db_missing_index_groups as g
                         inner join sys.dm_db_missing_index_group_stats as gs
                                on gs.group_handle = g.index_group_handle
                         inner join sys.dm_db_missing_index_details as d
                                on g.index_handle = d.index_handle
                         inner join sys.dm_exec_sessions as s
                                on s.session_id = 1
                   where d.database_id > 4
           </Value>
           <OutputTable>MissingIndexes</OutputTable>
         </Query>
       </ns:TSQLQueryCollector>' as xml);

exec dbo.sp_syscollector_create_collection_item
         -- Die Id des Auflistsatzes aus syscollector_collection_sets
         @collection_set_id = 11
         -- Dies ist die GUID für den T-SQL-Auflistungstyp
        ,@collector_type_uid = N'302E93D1-3424-4BE7-AA8E-84813ECF2419'
        ,@name = 'Fehlende Indizes'
        ,@frequency = 5 -- Wird im Snapshot Mode ignoriert
        ,@parameters = @definition
        ,@collection_item_id = @collection_item_id output;
```

Bitte achten Sie darauf, für den Parameter `@collection_set_id` den korrekten Wert anzugeben, also den aus der Abfrage der Tabelle `syscollector_collection_sets` erhaltenen Wert.

Der Zeitpunkt der Snapshot-Erstellung wird in der Spalte `snapshot_time` im `DATETIME`-Format gespeichert. Dieser Zeitpunkt wird auch – wie Sie etwas weiter unten noch sehen werden – in einer automatisch hinzugefügten Spalte gespeichert. Für diese automatisch erstellte Spalte ist der Datentyp jedoch `DATETIMEOFFSET(7)` und das Rechnen mit diesem

Datentyp ist etwas umständlich. Deshalb speichern wir den Zeitpunkt hier in einer eigenen Spalte.

Der interessante Teil des Skripts ist natürlich die XML-Definition des Auflistelements. In dieser Definition wird die Abfrage, die aufgerufen wird, spezifiziert, um die zu protokollierenden Daten zu ermitteln. Im Element `<OutputTable>` geben Sie den Namen der Tabelle an, die im VDWH erzeugt wird. In dieser Tabelle werden die Daten der Abfrage gespeichert. Die Tabelle wird erzeugt, sobald der Auflistsatz gestartet wird. Wir speichern auch den Zeitpunkt des SQL Server-Starts in jedem Snapshot. Dieser Zeitpunkt wird später für die Auswertung der Daten benötigt. Darauf kommen wir in Abschnitt 11.1.4 noch einmal zurück.

11.1.3 Ein Auflistelement für die Indexverwendung

Als zweite Auflistung wollen wir nun die Informationen über die Indexverwendung hinzufügen. Dies erledigt das folgende Skript:

```
declare @definition xml;
declare @collection_item_id int;

select @definition = cast(
      N'<ns:TSQLQueryCollector xmlns:ns="DataCollectorType">
         <Query>
           <Value>select object_name(i.object_id) as table_name
                        ,s.login_time as instance_start_time
                        ,current_timestamp as snapshot_time
                        ,i.type_desc, i.name
                        ,us.user_seeks, us.user_scans
                        ,us.user_lookups,us.user_updates
                        ,us.last_user_scan, us.last_user_update
                   from sys.indexes as i
                        inner join sys.dm_exec_sessions as s
                                on s.session_id = 1
                        left outer join sys.dm_db_index_usage_stats as us
                                     on i.index_id=us.index_id
                                    and i.object_id=us.object_id
                                    and us.database_id = db_id()
                  where objectproperty(i.object_id, ''IsUserTable'') = 1
           </Value>
           <OutputTable>IndexUsage</OutputTable>
         </Query>
         <Databases UseSystemDatabases="false" UseUserDatabases="true" />
       </ns:TSQLQueryCollector>' as xml);

exec dbo.sp_syscollector_create_collection_item
         -- Die Id des Auflistsatzes aus syscollector_collection_sets
         @collection_set_id = 11
         -- Dies ist die GUID für den T-SQL-Auflistungstyp
        ,@collector_type_uid = N'302E93D1-3424-4BE7-AA8E-84813ECF2419'
```

```
    ,@name = 'Indexverwendung'
    ,@frequency = 5 -- Wird im Snapshot Mode ignoriert
    ,@parameters = @definition
    ,@collection_item_id = @collection_item_id output;
```

Achten Sie bitte auch hier wieder darauf, dass die ID des Auflistsatzes korrekt angegeben werden muss.

Die SQL-Anweisung zur Ermittlung der Protokolldaten bestimmt alle Indizes und deren Verwendung. Diese Daten werden nur für Benutzertabellen abgefragt. Dafür sorgt die folgende Filterbedingung:

```
where objectproperty(i.object_id, 'IsUserTable') = 1
```

In der XML-Definition des Auflistelements wird durch das Element

```
<Databases UseSystemDatabases="false" UseUserDatabases="true" />
```

festgelegt, dass die spezifizierte Abfrage für alle Benutzerdatenbanken, aber nicht für Systemdatenbanken ausgeführt wird. An dieser Stelle könnten Sie auch nur die Namen bestimmter Datenbanken angeben. Das ist wirklich eine sehr bequeme Möglichkeit, eine Datensammlung so zu konfigurieren, dass identische Informationen für mehrere Datenbanken protokolliert werden.

Den Startzeitpunkt der SQL Server-Instanz benötigen wir wiederum für die Auswertung der protokollierten Daten. Darum geht es nun im folgenden Abschnitt.

11.1.4 Daten sammeln und auswerten

Wenn Sie die obigen Prozeduraufrufe zum Anlegen des Auflistsatzes und der beiden Auflistelemente ausgeführt haben, dann existiert der Auflistsatz *Index-Statistiken*, so wie in Abbildung 11.3 zu sehen.

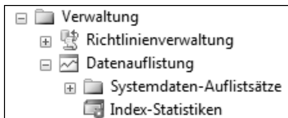


Abbildung 11.3:
Der Auflistsatz für Index-Statistiken

Sie können diesen Auflistsatz einfach aus dem Kontextmenü starten. Achten Sie bitte darauf, dass hierzu der SQL Server Agent laufen muss.

Die Auflistung und das Hochladen in das VDWH erfolgt nun alle 60 Minuten. Sie können das Hochladen der Daten auch jederzeit manuell aus dem Kontextmenü veranlassen.

Nach dem Start des Auflistsatzes finden Sie im VDWH die beiden Tabellen `custom_snapshots.IndexUsage` und `custom_snapshots.MissingIndexes`. Die Tabellen werden in dem speziellen Schema `custom_snapshots` erzeugt. Die Namen der Tabellen entsprechen dem in der XML-Definition des Auflistelements angegebenen Namen. Da wir zwei Auflistelemente mit je einer Tabelle konfiguriert haben, erhalten wir in unserem Fall also für jedes Auflistelement eine Tabelle.

Die Spalten für diese Tabellen werden automatisch aus der SQL-Anweisung abgeleitet, wobei noch zwei Spalten ergänzt werden, die den Snapshot selber – also den Zeitpunkt der Protokollierung – kennzeichnen (siehe Abbildung 11.4).

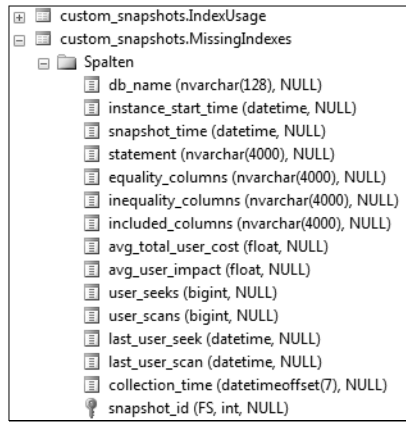


Abbildung 11.4:
Tabelle für Informationen über fehlende Indizes

Für die Auswertung der protokollierten Daten existieren keine vordefinierten Berichte. Sie sind also darauf angewiesen, diese Auswertung selber durchzuführen. Hierfür stehen Ihnen alle Möglichkeiten von T-SQL zur Verfügung. Die folgende Abfrage erzeugt beispielsweise die `CREATE INDEX`-Anweisungen zum Anlegen der protokollierten fehlenden Indizes:

```
with Ixs as
(
select top 40
       statement
      ,avg_user_impact * user_seeks * avg_total_user_cost as Impact
      ,user_seeks
      ,user_scans
      ,last_user_seek
      ,avg_total_user_cost
      ,avg_user_impact
      ,equality_columns
      ,inequality_columns
      ,included_columns
      ,'-- Info ------------------------------' + char(13) +
       '-- Impact Total  : ' + cast(cast(avg_user_impact * user_seeks
             * avg_total_user_cost as bigint) as varchar(80)) + char(13) +
       '-- User Seeks    : ' + cast(user_seeks as varchar(80)) + char(13) +
       '-- Avg Cost      : ' + cast(avg_total_user_cost as varchar(80))
                  + char(13) +
       '-- Avg Impact    : ' + cast(avg_user_impact as varchar(80))
                 + char(13) +
       '-- Last User Seek: ' + convert(varchar(20),last_user_seek,113)
                                              as Info
  from VDWH.custom_snapshots.MissingIndexes
 order by Impact desc
```

```
)
,IxCreate as
(
select distinct Info,statement + char(13) + '        '
          + replace(isnull('('+ equality_columns
             ,'(') + isnull(inequality_columns+')',')'),'][','],[')
      + isnull(char(13)+'      include (' + included_columns + ')','')
                       as Cols
  from Ixs
)
select Info + char(13) +
       ' create index _autoIx_'
              + replace(cast(newid() as varchar(80)),'-','')
              + char(13) + '    on ' + Cols + char(13)
 from IxCreate
```

Wenn Sie diese Abfrage ausführen, sollten Sie zuvor einstellen, dass das Abfrageergebnis im Textformat zurückgegeben wird und nicht in der sonst üblichen tabellarischen Form. Legen Sie in den Abfrageoptionen außerdem die maximal zulässige Spaltenbreite auf 8.192 Zeichen fest. (Dies ist der Maximalwert.) Sobald Sie dann die Abfrage starten, erhalten Sie für jeden fehlenden Index einen Kommentar mit einer Bewertung sowie eine `CREATE INDEX`-Anweisung zum Erzeugen des fehlenden Index.

Hier sehen Sie ein Beispiel für das Abfrageergebnis:

```
-- Info ------------------------------
-- Impact Total  : 2050
-- User Seeks    : 20
-- Avg Cost      : 1.15682
-- Avg Impact    : 88.72
-- Last User Seek: 24 Dez 2008 22:02:13
 create index _autoIx_B4DFC6C402B34828823996A7743D50EA
    on [AdventureWorks2008].[Sales].[SalesOrderDetail]
        ([CarrierTrackingNumber])
       include ([OrderQty], [ProductID])
```

Ich möchte Sie an dieser Stelle noch einmal daran erinnern, dass Sie die Hinweise über fehlende Indizes nicht bedenkenlos anwenden, sondern zuvor überprüfen sollten. Denken Sie bitte daran, dass für derartige Hinweise eine Reihe von Einschränkungen existieren, die Sie unbedingt beachten sollten. Sie finden diese Einschränkungen in der Online-Dokumentation. In den Kapiteln 6 und 10 folgen ebenfalls einige Erklärungen hierzu.

Da die Snapshots auf der Basis dynamischer Verwaltungssichten erstellt werden, enthalten sie stets kumulierte Daten. Wenn Sie zum Beispiel wissen möchten, welche Indizes in einem bestimmten Zeitraum nicht verwendet wurden, dann müssen Sie die Daten mehrerer Snapshots auswerten und die Ergebnisse voneinander subtrahieren. Eine entsprechende Verfahrensweise haben Sie bereits in Kapitel 10 kennengelernt. Die dort präsentierte Vorgehensweise hat allerdings den Nachteil, dass sie nach einem Neustart der SQL Server-Instanz keine korrekten Werte mehr liefert. In einem solchen Fall beginnen die Summierungen alle wieder mit dem Wert 0. Falls zwischen zwei aufgezeichneten Snapshots ein Neustart der

SQL Server-Instanz erfolgt, müssen die Werte dieser beiden Snapshots nicht voneinander abgezogen werden. Dieser Spezialfall soll natürlich ebenfalls berücksichtigt werden.

Zu diesem Zweck eignet sich am besten eine Tabellenwertfunktion, der Sie einen Datumsbereich übergeben können. Das folgende Skript zeigt, wie Sie eine solche Funktion in der Systemdatenbank *msdb* erstellen:

```
use msdb;
if (object_id('fn_GetIndexUsageInfo', 'IF') is not null)
  drop function dbo.fn_GetIndexUsageInfo
go
create function dbo.fn_GetIndexUsageInfo(@from datetime, @to datetime)
 returns table as
  return
   with Params(start_time, end_time) as
    (
     select min(snapshot_time)
           ,max(snapshot_time)
       from VDWH.custom_snapshots.IndexUsage
      where snapshot_time
          between isnull(@from,'19000101') and isnull(@to,'22000101')
    )
    ,CompleteValues(snapshot_time, instance_start_time
                   ,database_name, table_name
                   ,type_desc, name
                   ,user_seeks, user_scans, user_lookups, user_updates) as
    (
     select distinct
           '19000101', '19000101'
           ,database_name, table_name
           ,type_desc, name
           ,0, 0, 0, 0
       from VDWH.custom_snapshots.IndexUsage
      union all
     select snapshot_time, instance_start_time
           ,database_name, table_name
           ,type_desc, name
           ,isnull(user_seeks, 0), isnull(user_scans, 0)
           ,isnull(user_lookups, 0), isnull(user_updates, 0)
       from VDWH.custom_snapshots.IndexUsage
            inner join Params as p
                    on snapshot_time between p.start_time and p.end_time
    )
    ,NumberedValues(Record
                   ,snapshot_time, instance_start_time
                   ,database_name, table_name, type_desc, name
                   ,user_seeks, user_scans, user_lookups, user_updates) as
    (
     select row_number()
          over(partition by database_name, table_name, name
               order by snapshot_time)
```

```
          ,snapshot_time, instance_start_time
          ,database_name, table_name, type_desc, name
          ,user_seeks, user_scans, user_lookups, user_updates
      from CompleteValues
   )
  ,UsageCounts(database_name, table_name, name
              ,instance_start_time, from_time, to_time
              ,user_seeks, user_scans, user_lookups, user_updates) as
   (
    select nv1.database_name, nv1.table_name, nv1.name
          ,nv1.instance_start_time, nv2.snapshot_time, nv1.snapshot_time
          ,case
             when nv1.instance_start_time between nv2.snapshot_time
                                               and nv1.snapshot_time
                then nv1.user_seeks
             else nv1.user_seeks-nv2.user_seeks
           end
          ,case
             when nv1.instance_start_time between nv2.snapshot_time
                                               and nv1.snapshot_time
                then nv1.user_scans
             else nv1.user_scans-nv2.user_scans
           end
          ,case
             when nv1.instance_start_time between nv2.snapshot_time
                                               and nv1.snapshot_time
                then nv1.user_lookups
             else nv1.user_lookups-nv2.user_lookups
           end
          ,case
             when nv1.instance_start_time between nv2.snapshot_time
                                               and nv1.snapshot_time
                then nv1.user_updates
             else nv1.user_updates-nv2.user_updates
           end
      from NumberedValues as nv1
           inner join NumberedValues as nv2
                   on nv2.name = nv1.name
                  and nv2.database_name = nv1.database_name
                  and nv2.table_name = nv1.table_name
                  and nv2.Record = nv1.Record - 1
   )
   select database_name, table_name, name as index_name
         ,from_time, to_time
         ,user_seeks
         ,user_scans
         ,user_lookups
         ,user_updates
     from UsageCounts
go
```

Diese Funktion können Sie dann zum Beispiel verwenden, um herauszufinden, ob Indizes existieren, die in einem bestimmten Zeitraum ausschließlich für Aktualisierungsoperationen verwendet wurden:

```
select *
  from msdb.dbo.fn_GetIndexUsageInfo('20081001', '20091231')
 where user_seeks = 0
   and user_lookups = 0
   and user_scans = 0
   and user_updates > 0
```

Möglich ist auch eine Summierung nach Jahr und Monat:

```
select year(to_time) as Jahr
      ,month(to_time) as Monat
      ,database_name, table_name, index_name
      ,sum(user_seeks) as user_seeks, sum(user_scans) as user_scans
      ,sum(user_lookups) as user_lookups, sum(user_updates) as user_updates
 from msdb.dbo.fn_GetIndexUsageInfo(null, null)
group by grouping sets((year(to_time), month(to_time)), database_name)
        ,table_name, index_name
```

Über `GROUPING SETS` werden hier auch noch Gesamtsummen je Datenbank und Tabelle hinzugefügt.

Weitere Möglichkeiten ergeben sich, wenn Sie auf der Basis dieser Funktion eine Sicht definieren. Diese Sicht können Sie dann recht einfach in Excel verwenden, um die Daten zu importieren. Auf jeden Fall lohnt es sich auch, dass Sie sich einmal die Reporting Services ansehen und eventuell einen Bericht erstellen, der die von der Funktion zurückgelieferten Daten entsprechend aufbereitet. Diesen Bericht können Sie in das Management Studio integrieren. Dadurch ist der Bericht dann genauso aus dem Management Studio heraus aufrufbar wie die standardmäßig ausgelieferten Berichte.

11.2 Partitionierung mit Indizes

In diesem Abschnitt kommen wir noch einmal auf das Thema Partitionierung zurück. Zur Erinnerung: Bei einer Partitionierung werden Tabellen- und/oder Indexdaten auf unterschiedliche physische Speicherorte verteilt. Das Ziel ist die Minimierung bzw. Parallelisierung von E/A-Operationen. Wir werden an dieser Stelle spezielle Techniken mit Indizes verwenden, um eine Partitionierung zu bewerkstelligen. Darüber hinaus finden Sie hier auch einige Ergänzungen zum Thema Indexoptimierung.

Für die Experimente und Untersuchungen verwenden wir dabei die in Kapitel 7 erzeugte Tabelle `Kunde`. Führen Sie bitte das zu Beginn von Kapitel 7 gezeigte Skript, welches die Tabelle erstellt und anschließend 30.000 Zeilen einfügt, noch einmal aus.

Für die Abfragen in den Versuchen ist die Option `SET STATISTICS IO ON` zur Messung der E/A-Operationen gesetzt.

11.2.1 Horizontale Partitionierung

Die folgende Abfrage gibt alle Kunden zurück, deren letzte Bestellung nicht weiter zurückliegt als in das Jahr 2007:

```
select Nr, Nachname, LetzteBestellung
  from Kunde
 where LetzteBestellung >= '20080101'
 order by LetzteBestellung desc
```

Der zugehörige Ausführungsplan ist in Abbildung 11.5 zu sehen.

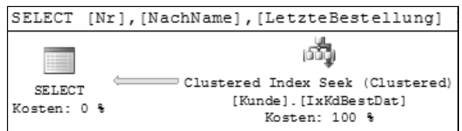


Abbildung 11.5:
Clustered Index Seek für eine Suche nach dem letzten Bestelldatum

Wie zu erwarten, wird eine Suche im gruppierten Index durchgeführt. Hierfür sind insgesamt 357 logische Lesevorgänge erforderlich.

Die Suche im gruppierten Index ist für die obige Abfrage der optimale Weg. Dieser Index wird allerdings nur deshalb verwendet, weil kein besserer Index existiert. Ein nichtgruppierter, abdeckender Index kann die Anzahl der erforderlichen Lesevorgänge mit Sicherheit beträchtlich reduzieren. Die recht hohe Anzahl logischer Lesevorgänge kommt in diesem Fall dadurch zustande, dass über den gruppierten Index nicht nur die tatsächlich benötigten Spalten, sondern alle Spalten gelesen werden müssen.

Wir können folgenden abdeckenden Index hinzufügen, um dies zu vermeiden:

```
create nonclustered index IxKdBestDat_0
   on Kunde(LetzteBestellung) include(Nr,Nachname)
```

Nun werden für die gleiche Abfrage wesentlich weniger logische Lesevorgänge benötigt, nämlich nur noch 27. Aus dem Abfrageplan ist ersichtlich, dass unser Index auch verwendet wird (siehe Abbildung 11.6). Bitte beachten Sie, dass der nichtgruppierte Index vom Optimierer nicht als fehlender Index bemängelt wurde (siehe Abbildung 11.5), obwohl er die Anzahl der Lesevorgänge und auch die geschätzten Kosten der Abfrage um mehr als den Faktor 13 reduziert!

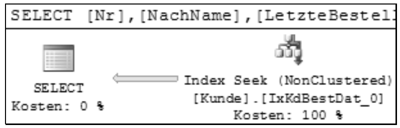


Abbildung 11.6:
Index Seek für eine Suche nach dem letzten Bestelldatum

Wenn wir diese Idee weiterverfolgen und hierbei die neue Möglichkeit von gefilterten Indizes in Betracht ziehen, könnten wir auch statt eines einzigen abdeckenden Index mehrere dieser Indizes erzeugen, zum Beispiel so:

```
create nonclustered index IxKdBestDat_1
   on Kunde(LetzteBestellung) include(Nr,Nachname)
    where (LetzteBestellung < '20061231')
go
create nonclustered index IxKdBestDat_2
   on Kunde(LetzteBestellung) include(Nr,Nachname)
    where (LetzteBestellung >= '20061231'
      and LetzteBestellung < '20080101')
go
create nonclustered index IxKdBestDat_3
   on Kunde(LetzteBestellung) include(Nr,Nachname)
    where (LetzteBestellung >= '20080101')
```

Selbstverständlich können wir die einzelnen Indizes auch in unterschiedlichen Dateigruppen anlegen, sodass letztlich auch eine physische Trennung der Daten entsteht. Auf diese Weise haben wir hier auch irgendwie eine horizontale Partitionierung erzielt. Die Idee dabei ist, dass der Indexfilter dem Kriterium der Abfrage entspricht und dass dadurch nunmehr nicht ein großer (also tiefer) Indexbaum durchsucht werden muss. Für unsere Abfrage würde es jetzt ausreichen, über den Index `IxKdBestDat_3` zu suchen; möglicherweise sind hierfür ja weniger Suchschritte erforderlich.

Dies ist jedoch nicht unbedingt der Fall. Wenn Sie die drei gefilterten Indizes erzeugen und die Abfrage nochmals ausführen, werden Sie feststellen, dass der passende gefilterte Index nicht verwendet wird. Sie erhalten denselben Ausführungsplan wie in Abbildung 11.6.

Warum ist das so? Nun, alle abdeckenden Indizes haben dieselbe Tiefe. Der Optimierer kann sich also genauso gut für den nicht gefilterten Index entscheiden, was er auch tut.

Die Situation ändert sich, sobald die Anzahl der Zeilen in den beiden Indizes so stark variiert, dass der gefilterte Index weniger Stufen besitzt. Das folgende Skript fügt 500.000 Zeilen in die Tabelle ein, wobei die Spalte `LetzteBestellung` stets auf einen Wert vor dem Datum 01.01.2008 gesetzt wird.

```
insert Kunde (Nr, VorName, NachName, LetzteBestellung)
 select n+100000,newid(),newid()
       ,dateadd(d, -(abs(checksum(newid())) % 5000), '20080101')
   from Numbers where n <= 500000
```

Das Skript wird den Index `IxKdBestDat_3` also nicht verändern, wohl aber den Index `IxKdBestDat_0`. Nachdem das Skript gelaufen ist, hat der Indexbaum für den Index `IxKdBestDat_3` eine geringere Tiefe als der Baum für den Index `IxKdBestDat_0`, der ja über alle Zeilen der Tabelle indiziert.

Wenn wir die Abfrage nun nochmals ausführen, entscheidet sich der Optimierer für einen Index-Scan des gefilterten Index, da diese Möglichkeit nun günstiger ist. Eine Suche im Indexbaum, also ein Index Seek ist nicht erforderlich, weil alle Daten des Index gelesen werden. Sehr viel macht dies allerdings nicht aus. Mit meinen Testdaten betrug der Unterschied gerade einmal drei logische Lesevorgänge, was ungefähr 11 Prozent entspricht.

Sie können den Unterschied sofort erkennen, wenn Sie die Abfrage mit einem Indexhinweis ausführen:

```
select Nr, NachName, LetzteBestellung
  from Kunde with(index=IxKdBestDat_0)
 where LetzteBestellung >= '20080101'
 order by LetzteBestellung desc

select Nr, NachName, LetzteBestellung
  from Kunde
 where LetzteBestellung >= '20080101'
 order by LetzteBestellung desc
```

Abdeckende, gefilterte Indizes bieten also auch eine Möglichkeit zur horizontalen Partitionierung. Aus meiner Sicht ist die Administration solcher Indizes einfacher als die ständige Überwachung und Veränderung von »echten« horizontalen Partitionen. Wenn sich Ihre Daten verändern, müssen Sie lediglich die Indizes mit geänderten Parametern neu aufbauen. Da Sie Ihre Indizes sowieso regelmäßig defragmentieren sollten (siehe Kapitel 6), können Sie geänderte Bedingungen für gefilterte Indizes auch gleich bei der Reorganisation mit erledigen.

Demgegenüber schlagen natürlich der größere Speicherplatzbedarf und die Beeinflussung von Aktualisierungsoperationen als Nachteile der »Index-Partitionen« zu Buche. Diese Nachteile können in OLTP-Anwendungen durchaus ins Gewicht fallen. In OLAP-Datenbanken kann es jedoch durchaus sinnvoll sein, eine horizontale Partitionierung mit Indizes durchzuführen. Oftmals werden in OLAP-Datenbanken Indizes vor dem Laden von Daten deaktiviert und nach dem Ladevorgang erneut aufgebaut, um die Aktualisierung zu beschleunigen. In diesem Fall können Sie natürlich »partitionierte Indizes« verwenden, da Sie die Aktualisierung nur geringfügig beeinflussen. Wie so oft, ist es also auch in diesem Fall nicht möglich, ein allgemein gültiges »Rezept« anzugeben. Die Entscheidung hängt größtenteils von den beiden folgenden Faktoren ab:

1. Wie groß sind die Auswirkungen der zusätzlichen Indizes auf Ihre Aktualisierungsoperationen? Hier eine Voraussage zu treffen, ist ziemlich schwierig. Normalerweise müssen Sie diese Auswirkungen experimentell ermitteln. Wenn Sie eine Performance-Richtlinie haben, die Sie überprüfen können – zum Beispiel mit T-SQL-Skripten –, dann sollte eine solche Überprüfung nicht allzu schwierig sein. Generell sollten Sie die Verfahrensweise in OLTP-Datenbanken nicht oder nur sehr vorsichtig anwenden, wenn dadurch zusätzliche Indizes erforderlich sind. Falls es Ihnen gelingt, einen Index durch mehrere gefilterte Indizes zu ersetzen, sieht dies natürlich etwas anders aus. In diesem Fall fällt der zusätzliche Verwaltungsaufwand für die Indizes nicht so sehr ins Gewicht. Möglicherweise ist eine Aktualisierungsoperation dann sogar günstiger, weil nur ein kleinerer Index anstelle eines großen Index aktualisiert werden muss.
2. Wie groß ist der Vorteil für Leseoperationen? Sie haben in unseren Experimenten gesehen, dass sich der Aufwand nur für große Datenmengen, also Tabellen mit vielen Zeilen, lohnt. Eine Partitionierung mit Indizes ist also genauso wie die »normale« Partitionierung erst ab einem gewissen Datenvolumen sinnvoll.

11.2.2 Vertikale Partitionierung

Das Beispiel aus dem vorherigen Abschnitt hat auch etwas anderes demonstriert: Wir haben einen nichtgruppierten, abdeckenden Index eingeführt, um den relativ teuren Clustered Index Scan zu vermeiden. Dieser abdeckende Index hat letztlich deshalb deutlich weniger Lesevorgänge benötigt, weil er nur die tatsächlich benötigten Spalten enthält.

In gewisser Weise ist dies auch eine vertikale Partitionierung. Sie können abdeckende Indizes erstellen, die im Hinblick auf bestimmte Abfragen optimiert wurden. Dadurch erreichen Sie eine Minimierung der Lesevorgänge für diese Abfragen. Dies haben wir in vielen Experimenten in den vorangegangenen Kapiteln oftmals getan. Natürlich ist es möglich, diverse abdeckende Indizes für eine Tabelle zu erstellen, um unterschiedliche Abfragen zu unterstützen. Die dadurch entstehende vertikale Partitionierung basiert allerdings darauf, dass Tabellendaten in Kopie existieren und dass aus diesen Kopien gelesen wird.

Der Vorteil einer solchen vertikalen Partitionierung ist ihre einfache Handhabbarkeit. Wenn sich Ihre Abfragen ändern, können Sie jederzeit die entsprechenden abdeckenden Indizes modifizieren. Dies ist sehr viel einfacher, als eine vertikale Partitionierung zu konfigurieren und zu überwachen, bei der die Spalten einer Tabelle auf mehrere Tabellen verteilt werden (siehe das Beispiel in Kapitel 7). Der Preis, den Sie hierfür zahlen, ist der höhere Speicherplatzbedarf und Verwaltungsaufwand.

Gleichzeitig liegt auch eine Gefahr darin, dass Sie abdeckende Indizes anpassen müssen, wenn sich Abfragen verändern. Denken Sie noch einmal an das in Kapitel 9 zum Thema »Auffinden geeigneter Indizes« Gesagte: Ein abdeckender Index ist für eine bestimmte Abfrage maßgeschneidert. Wenn sich die Abfrage ändert, kann es passieren, dass ein abdeckender Index nicht mehr seinen eigentlichen Zweck erfüllt, nämlich alle für eine Abfrage benötigten Daten zu liefern. Der höhere Verwaltungs- und Speicherplatzbedarf für den abdeckenden Index ist dann nicht mehr gerechtfertigt. Trotzdem kann es durchaus möglich sein, dass der Index nach wie vor für Index Seek-Operationen verwendet wird. Und eben genau deshalb ist es sehr schwierig, solche Indizes durch eine Überwachung der Indexverwendung aufzuspüren.

11.3 Arbeiten mit dem Datenbankoptimierungsratgeber

Der Datenbankoptimierungsratgeber (englisch: Database Tuning Advisor, DTA) ist ein Werkzeug zur Optimierung der physischen Datenbankstruktur. Der Ratgeber analysiert eine sogenannte Arbeitsauslastung, die in unterschiedlichen Formaten zur Verfügung gestellt werden kann:

- **Als T-SQL Skript**. Dieses Skript kann in einer Datei gespeichert sein, die durch den DTA geladen wird. Möglich ist auch die Übergabe einer im Management Studio geöffneten Abfrage an den DTA.
- **Als Ablaufverfolgung des SQL Server Profilers**. Eine Ablaufverfolgung kann wahlweise in einer Ablaufverfolgungsdatei, einer Datenbanktabelle oder auch in einer XML-Ablaufverfolgungsdatei gespeichert werden. Der DTA kann all diese Formate als

Eingabe verarbeiten. Hierzu müssen allerdings bestimmte Ereignisse und Ereignisspalten in die Ablaufverfolgung eingeschlossen werden, andernfalls ist eine Analyse einer durch den Profiler erzeugten Arbeitsauslastung mit dem DTA nicht möglich. Die minimal erforderlichen Ereignisse sind in der Vorlage *Tuning* bereits enthalten. Verwenden Sie also einfach diese Vorlage, falls Sie eine Arbeitsauslastung für den DTA erstellen möchten.

Der DTA analysiert die Arbeitsauslastung und erstellt als Ergebnis der Analyse Vorschläge für Änderungen der physischen Datenbankstruktur. Diese Vorschläge umfassen die folgenden Bereiche:

- **Indizes**. Wenn die in der Arbeitsauslastung enthaltenen T-SQL-Anweisungen von zusätzlichen Indizes profitieren können, dann sind diese Indizes in der Liste der Vorschläge enthalten. Die Analyse im Hinblick auf fehlende Indizes ist um einiges präziser, als sie der Optimierer bei der Erstellung eines Ausführungsplanes vornimmt. Sie können daher in Ausführungsplänen bemängelte fehlende Indizes nochmals durch den DTA überprüfen lassen, sofern Sie sich nicht sicher sind, ob ein im Ausführungsplan aufgeführter fehlender Index tatsächlich die angegebene Verbesserung bewirkt. Hierzu verwenden Sie einfach die entsprechende Abfrage als Eingabe für den DTA.
- **Horizontale Partitionierungen**. Der DTA unterbreitet auch Vorschläge für horizontale Partitionierungen, also die Aufteilung der Zeilen einer Tabelle auf mehrere Tabellen.
- **Fehlende Statistiken**. Zusätzliche Statistiken, die den Optimierer bei der Erstellung eines optimalen Ausführungsplans unterstützen würden, sind in der Liste der Vorschläge ebenfalls enthalten. Normalerweise werden solche Statistiken vor der Erstellung des Ausführungsplans automatisch erstellt, wenn Sie für eine Datenbank die Option `AUTO_CREATE_STATISTICS` eingeschaltet haben. Diese automatisch erstellten Statistiken sind allerdings stets einspaltig. Der DTA unterbreitet auch Vorschläge für mehrspaltige Statistiken.
- **Indizierte Sichten**. Falls die in der Arbeitsauslastung enthaltenen T-SQL-Anweisungen von indizierten Sichten profitieren würden, kann der DTA auch hierfür entsprechende Hinweise erstellen.

Nach Abschluss der Analyse erhalten Sie nicht nur Vorschläge für Veränderungen bzw. Verbesserungen der physischen Datenbankstruktur, sondern auch diverse Berichte, die nähere Informationen zur Analyse enthalten. Über diese Berichte können Sie zum Beispiel abfragen, wie häufig ein bestehender Index verwendet wurde, wie oft eine bestimmte Abfrage in der Arbeitsauslastung enthalten ist oder auch, wie oft auf bestimmte Tabellen zugegriffen wurde. Diese Berichte sind ein wichtiges Hilfsmittel bei der Auswertung der unterbreiteten Vorschläge. In der Regel ist die Liste der Vorschläge so lang, dass Hilfsmittel bei der Evaluierung dieser Vorschläge hilfreich sind. Der DTA prognostiziert auch eine Verbesserung der Abfrageleistung; allerdings bezieht sich der Wert für die erwartete Verbesserung stets auf die Gesamtliste der Änderungen. Falls in den Änderungsvorschlägen beispielsweise dreizehn zusätzliche Indizes enthalten sind, die insgesamt eine prognostizierte Verbesserung um 88 Prozent bewirken, kann es durchaus sein, dass bereits das Hinzufügen von nur zwei dieser Indizes eine Verbesserung um 80 Prozent bewirkt, sodass die restlichen elf Indizes dann lediglich noch einmal eine Verbesserung um acht Prozent hervorrufen.

Normalerweise werden Sie also nicht alle Elemente der Vorschlagsliste ohne Überlegungen anwenden. Die Kunst besteht darin, aus der Liste der Vorschläge diejenigen mit der größten Auswirkung auf die Verbesserung der Abfrageleistung herauszupicken. Außer den zur Verfügung stehenden Berichten hilft hierbei die Möglichkeit der Evaluierung der angebotenen Verbesserungsvorschläge. Sie können aus den Vorschlägen gezielt nur spezielle Empfehlungen auswählen und dann mit dieser Auswahl eine neue Analyse starten. Dadurch können Sie letztlich die Vorschläge mit dem größten Performance-Gewinn herausfinden und beispielsweise messen, inwiefern sich das Hinzufügen eines bestimmten Index auf die Abfrageleistung auswirkt.

Sie können den DTA auf drei Arten starten:

1. Über das Startmenü. In der Programmgruppe MICROSOFT SQL SERVER 2008 • LEISTUNGSTOOLS finden Sie den Eintrag DATENBANKOPTIMIERUNGSRATGEBER.
2. Aus dem Management Studio über den Menüeintrag EXTRAS • DATENBANKOPTIMIERUNGSRATGEBER.
3. Aus dem Management Studio über den Menüeintrag ABFRAGE • ABFRAGE MIT DEM DATENBANKMODUL-OPTIMIERUNGSRATGEBER OPTIMIEREN. Diese Möglichkeit haben Sie, wenn das aktive Fenster ein Abfragefenster ist. Das in diesem Fenster enthaltene T-SQL-Skript wird dann als Eingabe für den DTA verwendet. Interessanterweise wird hier noch der alte Name »Datenbank*modul*-Optimierungsratgeber« verwendet. Dieser Name wurde mit der Version 2008 auf »Datenbankoptimierungsratgeber« geändert – offensichtlich aber noch nicht durchgängig.

Ich möchte Ihnen für die dritte Möglichkeit ein Beispiel präsentieren, bei dem noch einmal die Tabelle `Kunde` aus Abschnitt 11.2.1 verwendet wird.

Wir entfernen zunächst alle nichtgruppierten Indizes auf der Tabelle `Kunde`. Anschließend öffnen wir ein neues Abfragefenster mit der folgenden Abfrage:

```
use QueryTest;
select Nr, NachName, LetzteBestellung
  from Kunde
 where LetzteBestellung >= '20080101'
 order by LetzteBestellung desc
```

Dies ist die Abfrage aus Abschnitt 11.2.1, deren Ausführungsplan Sie in Abbildung 11.5 sehen.

Über das Menü ABFRAGE • ABFRAGE MIT DEM DATENBANKMODUL-OPTIMIERUNGSRATGEBER OPTIMIEREN kann nun eine Analyse der Abfrage mit dem DTA vorgenommen werden.

Nach dem Start des DTA wird sofort eine neue Sitzung geöffnet. Diese Sitzung erhält einen automatisch vergebenen Namen, der sich aus dem Namen des angemeldeten Windows-Benutzers und der aktuellen Zeit zusammensetzt. Sie können den Namen nach Belieben ändern, etwa in *Performance-Test-Kunde*. In Abbildung 11.7 sehen Sie den Datenbankoptimierungsratgeber in Aktion.

Beim Start des DTA auf die oben beschriebene Art haben Sie keine Möglichkeit, eine Arbeitsauslastung anzugeben. Als Arbeitsauslastung wird automatisch das übergebene T-SQL-Skript aus dem Abfragefenster verwendet. Ansonsten müssten Sie in einem ers-

ten Schritt nun die Quelle für die Arbeitsauslastung spezifizieren. In unserem Fall ist hier bereits die Option *Abfrage* ausgewählt; es gibt keine Möglichkeit, dies zu verändern.

Unter DATENBANK FÜR ARBEITSAUSLASTUNGSANALYSE können Sie die Datenbank auswählen, mit der die Analyse startet. Falls Sie in Ihrem Skript den Datenbankkontext durch das Kommando USE <db> ändern, wird dies natürlich berücksichtigt.

Falls Sie in Ihr Skript kein USE <db> aufnehmen und hier eine falsche Datenbank angeben, wird die Analyse nicht funktionieren. Sofern möglich, legen Sie den Datenbankkontext immer in Ihrem Skript fest, um so etwas zu vermeiden.

In der unteren Liste der Datenbanken können Sie für jede aufgeführte Datenbank Tabellen und Sichten auswählen, die in der Analyse berücksichtigt werden sollen.

Abbildung 11.7: *Der Datenbankoptimierungsratgeber in Aktion*

Nachdem Sie die erforderlichen Einstellungen vorgenommen haben, können Sie auf der Seite OPTIMIERUNGSOPTIONEN detailliert festlegen, welche physischen Strukturen bei der Analyse berücksichtigt werden sollen. Abbildung 11.8 zeigt die vorhandenen Möglichkeiten. Sie können dort zum Beispiel bestimmen, ob bestehende Indexstrukturen ebenfalls in die Analyse einfließen sollen. Das ist wichtig, falls die Arbeitsauslastung nicht vollständig repräsentativ für Ihre Anwendungen ist. Es ist durchaus möglich, dass Ihre Arbeitsauslastung bestimmte Indizes nicht benötigt. Der DTA würde dann möglicherweise empfehlen, solche Indizes zu löschen, da diese ja niemals für eine Suche verwendet werden. Um so etwas zu verhindern, wählen Sie die Option ALLE VORHANDENEN PHYSISCHEN ENTWURFSSTRUKTUREN BEIBEHALTEN, so wie in Abbildung 11.8 gezeigt.

Die Analyse einer größeren Arbeitsauslastung kann sehr umfangreich werden. Möglicherweise ist es erforderlich, dass Sie die Analysezeit begrenzen, um in einer überschaubaren Zeit ein Ergebnis zu erhalten. Dieses Ergebnis ist dann natürlich nicht ganz exakt, da nicht alle Ereignisse bzw. T-SQL-Anweisungen in die Analyse eingegangen sind.

Abbildung 11.8: *Optimierungsoptionen des DTA*

Sobald alle Optionen feststehen, können Sie über die Schaltfläche ANALYSE STARTEN aus der Menüleiste mit der Analyse beginnen. Wie bereits gesagt, sollten Sie bei einer größeren Arbeitsauslastung hierfür etwas Zeit einplanen. Beachten Sie bitte auch, dass für die Analyse Serverressourcen benötigt werden. Es ist also sinnvoll, eine Analyse zu einem Zeitpunkt geringer sonstiger Aktivitäten durchzuführen. Für unsere Arbeitsauslastung mit nur einer Abfrage gilt dies natürlich nicht. Nach dem Start der Analyse erhalten wir bereits nach sehr kurzer Zeit eine Liste mit dem Analyseergebnis, so wie in Abbildung 11.9 gezeigt. Durch die ganz links dargestellte Option können Sie aus dieser Liste bestimmte Vorschläge für weitere Aktionen auswählen. In unserem Fall enthält die Liste lediglich einen Eintrag.

Abbildung 11.9: *Ergebnis der Analyse*

Der DTA empfiehlt das Hinzufügen eines Index und prognostiziert hierfür eine Verbesserung um 91 Prozent. Eine solche Empfehlung kann relativ bedenkenlos ohne weitere Überlegungen übernommen werden. Hierzu wählen Sie zum Beispiel aus dem Menü den Eintrag AKTIONEN • EMPFEHLUNGEN ANWENDEN aus. In diesem Fall werden die in der Vorschlagsliste ausgewählten Einträge direkt in der entsprechenden Datenbank angelegt.

Der DTA vergibt automatisch Namen für alle in der Liste der Vorschläge enthaltenen Objekte, also zum Beispiel für Indizes. Normalerweise werden Sie diese Namen nicht übernehmen, sondern an Ihre Vorgaben anpassen. Daher ist die direkte Anwendung der ausgewählten Empfehlungen in der Regel nicht praktikabel. In den meisten Fällen werden Sie ein T-SQL-Skript erstellen und die Erzeugung der Objekte dementsprechend anpassen. Dabei können Sie nicht nur die Namen (zum Beispiel der Indizes) ändern, sondern auch physische Optionen (wie etwa Dateigruppen) an Ihre Erfordernisse anpassen. Um ein entsprechendes Skript zu erzeugen, haben Sie zwei Möglichkeiten. Zum einen können Sie für jedes Objekt ein Skript erstellen. Klicken Sie hierzu einfach auf den in der Spalte DEFINITION angegebenen Link. Möglich ist auch die Erstellung eines T-SQL-Skripts für alle ausgewählten Elemente. Dies erledigen Sie über den Menüpunkt AKTIONEN • EMPFEHLUNGEN SPEICHERN... oder einfach mittels der Tastenkombination [STRG] + [S].

Das Skript für den vorgeschlagenen Index sieht auf meiner Maschine wie folgt aus:

```
CREATE NONCLUSTERED INDEX [_dta_index_Kunde_10_4195065__K4_1_3]
          ON [dbo].[Kunde]
([LetzteBestellung] ASC) INCLUDE ([Nr],[NachName])
    WITH (SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY = OFF
        , DROP_EXISTING = OFF, ONLINE = OFF) ON [PRIMARY]
```

Sicher ist Ihnen aufgefallen, dass die durch den DTA vorgenommene Analyse auf jeden Fall detaillierter ist als die Untersuchung auf fehlende Indizes hin, die der Optimierer bei der Erstellung eines Ausführungsplans vornimmt. Betrachten Sie noch einmal Abbildung 11.5, so finden Sie dort keinen Hinweis auf einen fehlenden Index. Der DTA kundschaftet allerdings sehr wohl einen nützlichen Index aus, der sogar eine Verbesserung um 91 Prozent bewirkt. Wenn Sie einmal genau hinsehen, werden Sie feststellen, dass dies genau der Index ist, den wir in Abschnitt 11.2.1 manuell ermittelt haben.

Denken Sie also bitte daran, wenn Sie die in Abfrageplänen oder in den `sys.dm_db_missing_index...`-Verwaltungssichten enthaltenen Informationen auswerten: Die dort verfügbaren Informationen basieren nicht auf einer vollständigen Analyse und sind dementsprechend auch nicht in jedem Fall ausreichend für eine Aussage über fehlende Indizes.

11.3.1 Tipps zur Verwendung des Datenbankoptimierungsratgebers

Der DTA ist ein sehr nützliches Werkzeug, vor allem zur Ermittlung fehlender Indizes. Sie sollten jedoch einige Hinweise beim Umgang mit diesem Werkzeug beachten.

- **Halten Sie die Arbeitsauslastung so kurz wie möglich**. Die durch den DTA durchgeführte Analyse ist sehr detailliert und dauert für große Arbeitsauslastungen entsprechend lange. Versuchen Sie also nach Möglichkeit, nur eine Arbeitsauslastung für einen interessierenden Zeitraum zu verwenden. Alternativ können Sie die Analysezeit beschränken, aber dann ist das Ergebnis eben nicht vollständig.
- **Wählen Sie die Option** ALLE VORHANDENEN PHYSISCHEN STRUKTUREN BEIBEHALTEN **aus**. Dies ergibt sich unmittelbar aus der Forderung nach einer möglichst kurzen Arbeitsauslastung. Eine kurze Arbeitsauslastung wird sehr wahrscheinlich nicht repräsentativ genug sein, um zum Beispiel auch überflüssige Indizes zu ermitteln.
- **Empfehlungen für zusätzliche Statistiken**. Der DTA schlägt auch eine Reihe zusätzlicher Statistiken vor. Wenn Sie damit beginnen, die Analyseergebnisse auszuwerten, so lassen Sie beachten bitte diese Statistikempfehlungen zunächst nicht. Untersuchen Sie zu Beginn lediglich die ausgesprochenen Indexempfehlungen.
- **Empfehlungen für Indizes**. Der DTA wird in vielen Fällen eine lange Liste fehlender Indizes erstellen. Nicht in jedem Fall werden Sie tatsächlich all diese Indizes benötigen. Es gibt durchaus Situationen, in denen bereits ein bis zwei Indizes 80 Prozent der prognostizierten Verbesserung bewirken. Sofern dies der Fall ist, können Sie auf viele der vorgeschlagenen Indizes verzichten. Es kann durchaus mühsam sein, die wirklich wichtigen Indizes aus der Liste der Empfehlungen herauszufinden. Und dies bringt uns zum nächsten Tipp:
- **Experimentieren Sie mit den Empfehlungen**. Sie können aus den gemachten Empfehlungen gezielt nur einige auswählen und dann mit dieser Auswahl eine erneute Analyse starten. Rufen Sie hierzu den Eintrag AKTIONEN • EMPFEHLUNGEN BEWERTEN auf. Dadurch wird eine neue Sitzung gestartet, die eine Analyse mit den ausgewählten Empfehlungen wiederholt – also eine Art »Was wäre wenn?«-Szenario. Wie bereits gesagt, sollten Sie hierbei zunächst die empfohlenen Statistiken weglassen und ausschließlich mit Indizes beginnen. Um hier systematisch vorzugehen, ist allerdings einige Erfahrung erforderlich. Die empfohlenen Indizes müssen auch in beliebigen Kombinationen überprüft werden. Dabei sollten Sie natürlich mit denjenigen Kombinationen beginnen, die am vielversprechendsten für eine Verbesserung sind – und hierfür ist neben einer Kenntnis der entsprechenden Abfragen auch eine gewisse Vertrautheit mit der Arbeitsweise des Optimierers unerlässlich. Ich hoffe natürlich, dass Ihnen dieses Buch eine Hilfe dabei ist, bei der Evaluierung mit den aussichtsreichsten Indizes zu beginnen. Bedenken Sie bitte auch, dass Ihnen im Ergebnis der Analyse verstärkt abdeckende Indizes begegnen werden. Diese Indizes sind speziell auf die in der Arbeitsauslastung enthaltenen Abfragen angepasst und bei Änderungen in den Abfragen daher eventuell nicht mehr optimal. Außerdem kann es vorkommen, dass Sie dadurch gleichwertige Indizes im Ergebnis der Analyse vorfinden. So ist es zum Beispiel möglich, dass zwei nahezu gleiche Abfragen in der Arbeitsauslastung enthalten sind, die auf derselben Tabelle mit identischen `WHERE`-Klauseln aber

unterschiedlichen Spalten in der SELECT-Liste existieren. Der DTA wird in einem solchen Fall möglicherweise zwei Indizes auf derselben Tabelle und derselben Spalte, aber mit unterschiedlichen Spalten in der INCLUDE-Liste empfehlen. Sie werden wahrscheinlich nicht beide Indizes benötigen, und es liegt an Ihnen, einen von beiden nicht zu erstellen.

- **Schauen Sie sich die erzeugten Berichte an**. Nach der Analyse stehen Ihnen auf der Seite BERICHTE eine Reihe von Berichten zur Verfügung, über die Sie nähere Informationen zur Analyse erhalten können (siehe Abbildung 11.9). Schauen Sie sich diese Berichte an, um herauszufinden, welche empfohlenen Indizes möglicherweise die geeignetsten sind.
- **Verwenden Sie eventuell das Kommandozeilenprogramm DTA.EXE**. Wenn Sie den Datenbankoptimierungsratgeber häufig verwenden, kann es nützlich sein, dass Sie die Kommandozeilen-Version verwenden. Dadurch lassen sich beispielsweise Analysen automatisieren.
- **Verstehen Sie den DTA als ein Hilfsmittel**. Der Datenbankoptimierungsratgeber ist letztlich ein Hilfsmittel, das Sie vor allem bei der Analyse fehlender Indizes unterstützt. Auch wenn die Analyse umfangreich und detailliert ist, können Sie das Analyseergebnis in den meisten Fällen nicht ohne eine manuelle Überprüfung anwenden. Betrachten Sie zum Beispiel gefilterte Indizes: In unserer Analyse wurde keine Empfehlung für einen gefilterten Index ausgesprochen, obwohl dies eine geringfügige Verbesserung bewirkt hätte. Unabhängig davon, was die Ursache hierfür ist (wahrscheinlich ist in unserem Fall die Arbeitsauslastung nicht repräsentativ genug), so zeigt dies anschaulich, dass Sie stets über das Ergebnis Ihrer Analyse nachdenken sollten.

11.4 Zusammenfassung

In diesem Kapitel wurden erweiterte Konzepte der Indexoptimierung untersucht. Dabei haben Sie erfahren, wie Sie die Indexverwendung mit Datenauflistungen protokollieren und auswerten können.

Außerdem wurde erklärt, wie Sie gefilterte und abdeckende Indizes für Partitionierungen verwenden können. Sie sollten bedenken, diese Techniken nur mit Bedacht einzusetzen, da zusätzliche Indizes in aller Regel einen erhöhten Verwaltungs- und Speicherplatzbedarf zur Folge haben.

Sie wissen nun auch, wie Sie den Datenbankoptimierungsratgeber verwenden können, um zum Beispiel fehlende Indizes aufzuspüren, und welche Punkte Sie beim Umgang mit dem DTA beachten sollten.

12 Kontrollieren von Ressourcen

Wir haben in den vorangegangenen Kapiteln häufig von OLTP- und OLAP-Anwendungen gesprochen und in vielen Fällen unterschiedliche Verhaltensweisen dieser beiden Anwendungstypen herausgestellt. Auch für die Optimierung müssen Sie normalerweise beide Anwendungstypen getrennt voneinander betrachten, da sich die Optimierungsstrategien ebenfalls unterscheiden. Im Idealfall können Sie Ihre Optimierungsstrategie also auf einen bestimmten Anwendungstyp ausrichten.

Leider sieht es in der Praxis allerdings etwas anders aus. Die meisten Datenbanken müssen sowohl OLTP- als auch OLAP-Anforderungen erfüllen. Es ist absolut üblich, dass in OLTP-Datenbanken nicht nur geschäftliche Transaktionen verarbeitet werden, sondern auch Berichte auf der Basis der in diesen Datenbanken stets aktuell vorhandenen Daten erstellt werden. Die Erstellung von Berichten kann eindeutig als eine OLAP-Aktivität eingestuft werden. Damit wird eine Unterscheidung nach OLTP- und OLAP-Datenbanken schwierig.

Abfragen zur Berichtserstellung sind häufig sehr »ressourcenhungrig« und dauern entsprechend lange. Im schlimmsten Fall kann eine Berichtserstellung dazu führen, dass das Tagesgeschäft beeinträchtigt wird, weil die Verarbeitungszeiten für OLTP-Transaktionen stark anwachsen.

Denken Sie beispielsweise an den letzten Adventssamstag vor Weihnachten. Der Verkauf läuft in allen 400 Filialen eines Unternehmens auf Hochtouren, es haben sich bereits längere Schlangen an den Kassen gebildet. Ebenso ist auch der Jahresabschluss bereits in Vorbereitung. Der neue Assistent der Geschäftsleitung geht deshalb an diesem Samstag nicht einkaufen und arbeitet stattdessen. Er erstellt ausgefeilte Berichte auf der Basis der aktuellen Verkaufsdaten, die er am Montagmorgen sofort dem Geschäftsführer vorlegen kann. Diese Berichte enthalten zum Beispiel die Verkaufsdaten des gesamten Jahres und dauern daher dementsprechend lange. Das Warenwirtschaftssystem »geht in die Knie«, der Verkauf gerät ins Stocken, die Schlangen an den Kassen werden immer länger und die ersten Kunden verlassen genervt die Läden, ohne etwas zu kaufen.

Ein Ansatz für eine Optimierung kann in diesem natürlich Fall sein zu verbieten, dass zu Hauptverkaufszeiten andere ressourcenintensive Vorgänge – wie etwa das Erstellen von Berichten – laufen. Eine absolute Sicherheit erreichen Sie dadurch allerdings nicht. Wesentlich eleganter wäre eine Lösung, die dafür sorgt, dass Abfragen unterschiedlicher Anwendungen oder von Benutzern unterschiedlicher Priorität verarbeitet werden. Auf diese Weise könnten Sie dafür sorgen, dass der Verkauf stets Vorrang vor der Berichtserstellung erhält. Der Assistent muss dann ein wenig länger auf seine Berichte warten, aber der Verkauf wird nicht beeinträchtigt.

Hier kommt die Ressourcenkontrolle von SQL Server ins Spiel.

12.1 Funktionsweise der Ressourcenkontrolle

Mit Hilfe der Ressourcenkontrolle lassen sich eingehende Verbindungen klassifizieren. Je nach Verbindungseigenschaften findet dann für jede Verbindung eine Zuteilung der maximal möglichen Ressourcennutzung in Bezug auf CPU und Hauptspeicher statt. Möglich ist auch eine Priorisierung eingehender Verbindungen in drei Stufen (hoch, mittel und niedrig).

Für unser Beispiel aus der Einleitung müssten wir also zwischen Verbindungen unterscheiden, die Berichte erstellen, und solchen, die mit der OLTP-Anwendung arbeiten. Jeder dieser Verbindungstypen sollte dann einem eigenen Ressourcenpool zugeordnet werden. Über diesen Pool wird letztlich die maximal erlaubte CPU-Last und Hauptspeicherverwendung für alle dem Pool zugeordneten Verbindungen konfiguriert.

In Abbildung 12.1 sehen Sie eine vereinfachte Darstellung dieser Verfahrensweise.

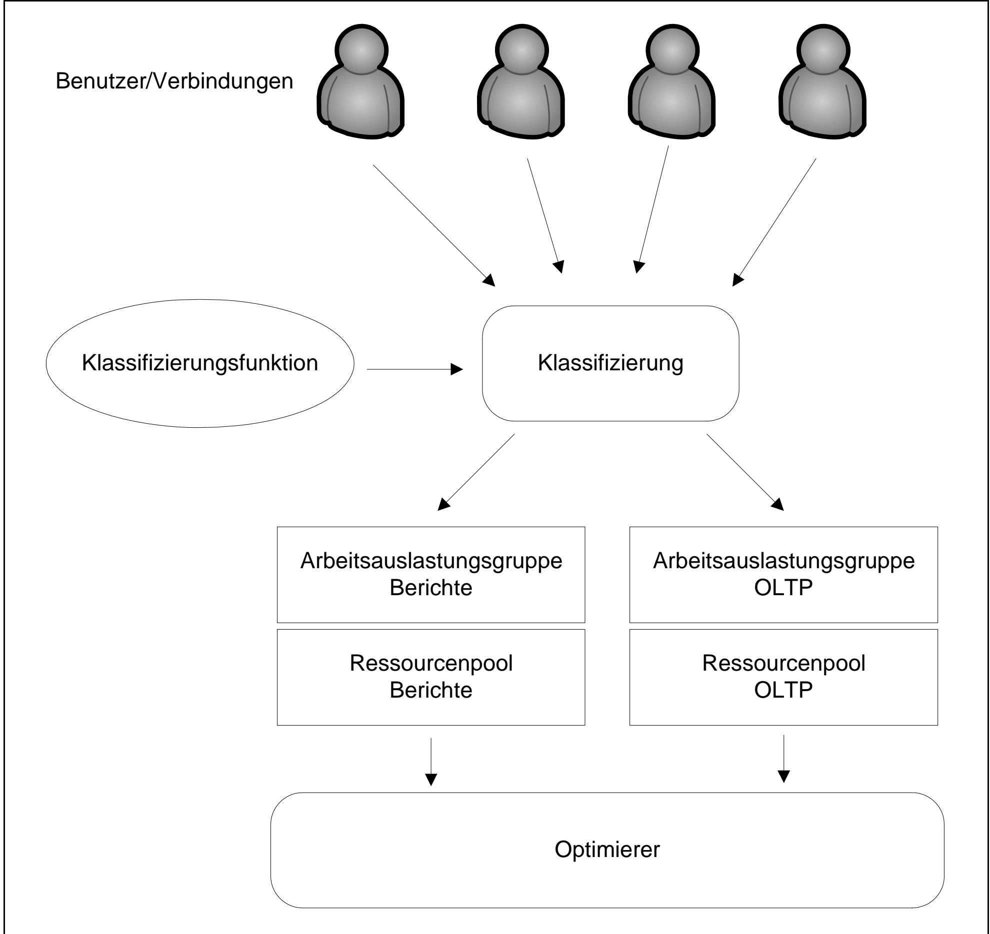


Abbildung 12.1: *Funktionsweise der Ressourcenkontrolle*

Eingehende Verbindungen werden zunächst klassifiziert und einer Arbeitsauslastungsgruppe zugeordnet. Diese Arbeitsauslastungsgruppe ist letztlich ein Container, in dem die zugeordneten Verbindungen verwaltet werden. Für die Arbeitsauslastungsgruppe können Sie beispielsweise festlegen, wie viel CPU-Zeit eine dieser Gruppe zugeordnete Abfrage maximal verwenden darf, oder wie viele gleichzeitige Anforderungen alle Verbindungen dieser Gruppe gemeinsam absetzen dürfen. Hier geben Sie auch eine Priorität an, mit der Anforderungen der zugeordneten Verbindungen abgearbeitet werden sollen.

Mit jeder Arbeitsauslastungsgruppe ist ein Ressourcenpool verknüpft. Dieser Ressourcenpool bestimmt die Maximalwerte für die CPU- und die Hauptspeicher-Verwendung für die der Arbeitsauslastungsgruppe zugeordneten Verbindungen.

12.2 Einrichten der Ressourcenkontrolle

Nach der Vorstellung der prinzipiellen Arbeitsweise der Ressourcenkontrolle wollen wir nun für unser einleitendes Beispiel die erforderlichen Objekte einrichten und die Ressourcenkontrolle aktivieren. Wir verwenden für die Einrichtung T-SQL-Skripte. Sie könnten hierfür ebenso gut die grafische Oberfläche des Management Studios hierfür benutzen.

12.2.1 Erstellen von Ressourcenpools

Im ersten Schritt erstellen wir die beiden Ressourcenpools für die OLTP-Anwendung und die Berichtsanwendung. Dies geschieht über den Befehl `CREATE RESOURCE POOL`. Wie bereits erwähnt, werden hierbei die Werte für die maximal zulässige Verwendung von CPU und Hauptspeicher festgelegt. Für diese Festlegungen stehen die folgenden Parameter zur Verfügung:

- **`MAX_CPU_PERCENT`**. Geben Sie hier den Maximalwert für die CPU-Last in Prozent an, die bei Konflikten für diesen Ressourcenpool zur Verfügung gestellt wird.
- **`MIN_CPU_PERCENT`**. Hier spezifizieren Sie, wie viel CPU garantiert für diesen Ressourcenpool zur Verfügung gestellt wird.
- **`MAX_MEMORY_PERCENT`**. Dies ist der maximal zur Verfügung stehende Arbeitsspeicher, den Abfragen dieses Ressourcenpools verwenden dürfen.
- **`MIN_MEMORY_PERCENT`**. Dieser Wert gibt den garantiert zur Verfügung stehenden Hauptspeicher für den Ressourcenpool an.

Bitte beachten Sie, dass die über den Ressourcenpool konfigurierten Werte nur dann als Begrenzung wirken, sofern Konflikte auftreten. Wenn Sie beispielsweise eine maximale CPU-Last von 50 Prozent konfigurieren, ist es durchaus möglich, dass auch 90 Prozent CPU-Last auftreten können, solange kein »Streit« um die Ressource CPU vorliegt. Die Ressourcenkontrolle wirkt also erst dann, wenn tatsächlich Ressourcenkonflikte auftreten.

Ressourcenpool für die OLTP-Anwendung

Die folgende Anweisung richtet den Ressourcenpool für die OLTP-Anwendung ein:

```
create resource pool rpOLTP with
 (
  max_cpu_percent = 100
 ,min_cpu_percent = 90
 ,max_memory_percent = 100
 ,min_memory_percent = 90
 )
```

Da unser SQL Server in der Hauptsache für die OLTP-Anwendung installiert wurde, soll diese Anwendung auch entsprechend hohe Belastungen für CPU und Arbeitsspeicher erzeugen dürfen.

Ressourcenpool für Benutzer, die Berichte erstellen

Die Berichtsanwendungen müssen mit deutlich weniger Ressourcen auskommen:

```
create resource pool rpReporting with
 (
  max_cpu_percent = 25
 ,min_cpu_percent = 0
 ,max_memory_percent = 25
 ,min_memory_percent = 0
 )
```

Hier sollen sowohl die Arbeitsspeicherverwendung als auch die CPU-Belastung nicht über 25 Prozent der Maximallast ansteigen dürfen, sofern diese Ressourcen knapp werden, also Konflikte auftreten.

12.2.2 Einrichten von Arbeitsauslastungsgruppen

Nach dem Anlegen des Ressourcenpools können wir nun die Arbeitsauslastungsgruppen erstellen. Hierzu existiert das Kommando `CREATE WORKLOAD GROUP`.

Wir benötigen zwei Arbeitsauslastungsgruppen, so wie in Abbildung 12.1 dargestellt: Eine für unsere OLTP-Anwendung und eine weitere für die Berichtsanwendungen.

Arbeitsauslastungsgruppe für OLTP-Anwender

Die Arbeitsauslastungsgruppe für die OLTP-Anwendung kann mit dem folgenden Befehl erstellt werden:

```
create workload group wlgOLTP
  with (importance = high)
  using rpOLTP
```

Wir setzen hier die Priorität hoch und verknüpfen die Arbeitsauslastungsgruppe mit dem zuvor erstellten Ressourcenpool.

Arbeitsauslastungsgruppe für die Berichtserstellung

Für Berichtsanwendungen sieht die Erstellung der Arbeitsauslastungsgruppe so aus:

```
create workload group wlgReporting
  with (importance = low)
  using rpReporting
```

Auch hier erfolgt die Verknüpfung mit dem entsprechenden Ressourcenpool. Außerdem legen wir fest, dass Anforderungen, die über diese Arbeitsauslastungsgruppe abgesetzt werden, mit einer niedrigeren Priorität bearbeitet werden.

12.2.3 Entwerfen einer Klassifizierungsfunktion

Was nun noch fehlt, ist eine Klassifizierungsfunktion, welche die Eigenschaften eingehender Verbindungen überprüft und eine Zuordnung der Verbindungen zu den einzelnen Arbeitsauslastungsgruppen vornimmt. Eine solche Funktion zu entwerfen ist einfacher, als Sie vielleicht denken. Die Funktion muss den Namen einer Arbeitsauslastungsgruppe als Zeichenkette zurückliefern. Innerhalb der Funktion müssen Sie also entscheiden, welche Arbeitsauslastungsgruppe verwendet werden soll. Hierzu können Sie auf diverse Systemfunktionen zurückgreifen, zum Beispiel: `APP_NAME()`, `HOST_NAME()`, `SUSER_NAME()` oder `IS_SRVROLEMEMBER()`.

Mit diesen Funktionen können Sie Verbindungseigenschaften abfragen und anhand dieser Eigenschaften festlegen, welcher Arbeitsauslastungsgruppe eine Verbindung zugeordnet werden soll. Unsere Klassifizierungsfunktion kann zum Beispiel wie folgt entworfen werden:

```
use master
go
create function dbo.rgClassification()
   returns sysname
   with schemabinding as
  begin
    declare @wlgName      sysname
           ,@current_time time
    set @current_time = cast(current_timestamp as time)
    if (@current_time between '09:00' and '20:00')
      begin
        if (app_name() like '%Reporting%')
           or (app_name() like '%Bericht%')
          set @wlgName = 'wlgReporting'
        else
          set @wlgName = 'wlgOLTP'
      end
    return @wlgName
  end
```

In der Funktion wird zunächst überprüft, ob der aktuelle Zeitpunkt innerhalb der Ladenöffnungszeit liegt. Ist dies nicht der Fall, wird die Funktion sofort verlassen. Dadurch sind Berichtsanwendungen in der Lage, Berichte zum Beispiel in der Nacht mit höherer Priorität zu erstellen.

Über die Funktion `APP_NAME()`wird entschieden, welcher Arbeitsauslastungsgruppe eine Verbindung zugeordnet wird.

Sicher fragen Sie sich nun, was passiert, wenn die Funktion außerhalb der Ladenöffnungszeiten aufgerufen wird. In diesem Fall wird ja der Wert `NULL` zurückgeliefert, was die Frage aufwirft, welcher Ressourcenpool in diesem Fall verwendet wird. SQL Server besitzt für diesen Zweck einen Systemressourcenpool namens *default* mit einer zugehörigen Arbeitsauslastungsgruppe gleichen Namens. Wann immer die Klassifizierungsfunktion eine unbekannte Arbeitsauslastungsgruppe zurückgibt, erfolgt die Zuordnung zu der Arbeitsauslastungsgruppe *default*. Sowohl der Ressourcenpool *default* als auch die Arbeitsauslastungsgruppe *default* können nicht gelöscht werden. Es ist aber möglich, die Ressourcenverwendung für diese Gruppe oder diesen Pool anzupassen.

Bedenken Sie bitte, dass die innerhalb der Klassifizierungsfunktion abgefragten Verbindungseigenschaften nicht unveränderlich sind. So kann zum Beispiel eine Anwendung beim Aufbau einer Verbindung einfach in den Verbindungseigenschaften den Parameter für den Anwendungsnamen setzen. Falls Ihre Klassifizierungsfunktion den Namen der Anwendung als Entscheidungsgrundlage verwendet, wird die Klassifizierung nicht mehr funktionieren, wenn die Anwendung die Verbindungszeichenfolge so ändert, dass ein anderer Anwendungsname angegeben wird. In diesem Fall erfolgt dann möglicherweise eine Zuordnung zur Arbeitsauslastungsgruppe *default*. Beachten Sie dies bitte beim Entwurf Ihrer Klassifizierungsfunktion.

Einen weiteren Hinweis sollten Sie ebenfalls beachten:

SQL Server erlaubt nur eine einzige Klassifizierungsfunktion. Falls Sie also viele Arbeitsauslastungsgruppen verwenden möchten oder müssen, kann eine Klassifizierungsfunktion entsprechend umfangreich und unübersichtlich werden.

12.2.4 Aktivieren der Ressourcenkontrolle

Nachdem nun alle erforderlichen Elemente erzeugt wurden, kann die Ressourcenkontrolle aktiviert werden. Hierzu existiert das Kommando `ALTER RESOURCE GOVERNOR`. Für die Aktivierung muss der Ressourcenkontrolle zunächst die Klassifizierungsfunktion mitgeteilt werden. Anschließend kann die Einstellung durch `RECONFIGURE` übernommen werden:

```
use master;
alter resource governor
  with (classifier_function = dbo.rgClassification)
go
alter resource governor reconfigure;
```

Die Ressourcenkontrolle ist nun aktiv und der Verkauf kann wieder normal laufen. Der Assistent der Geschäftsleitung kann seine Berichte erzeugen, ohne dass dies das Tagesgeschäft beeinträchtigt. Er wird eben nur etwas länger warten müssen. Falls er so lange warten muss, dass es nach 20:00 Uhr wird, oder er generell vor 09:00 Uhr oder nach 20:00 Uhr arbeitet, dann laufen seine Berichte zügig durch, denn zu diesen Zeiten werden Berichtsanwendungen der Arbeitsauslastungsgruppe *default* zugeordnet.

Im Objekt-Explorer des Management Studios können Sie die konfigurierte Ressourcenkontrolle sehen. Öffnen Sie hierzu den Ordner *Verwaltung/Ressourcenkontrolle* (siehe Abbildung 12.2).

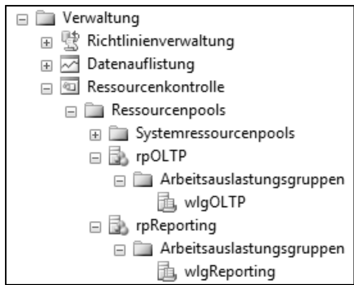


Abbildung 12.2: *Ansicht der Ressourcenkontrolle im Objekt-Explorer des SSMS*

Für alle dargestellten Elemente ist die Anzeige der Eigenschaften möglich. Änderungen der Konfiguration, wie zum Beispiel die Anpassung der Ressourcenverwendung für die existierenden Pools, können Sie ebenfalls vornehmen.

12.3 Zusammenfassung

Ressourcenkontrolle ist ein spezieller Mechanismus zur Einordnung von Verbindungen in unterschiedliche Prioritätsklassen. In diesem Kapitel haben Sie gelernt, wie Sie die Ressourcenkontrolle einsetzen können, um Verbindungen nach ihrer Wichtigkeit einzustufen, und wie Sie Ihr System vor unvorhersehbaren Anforderungen oder Belastungen schützen können.

13 Testen und Optimieren des E/A-Systems

Der erste Ansatz, wenn ein System Performance-Probleme aufweist, ist für viele Administratoren die Aufrüstung der Hardware. Ich möchte Sie an dieser Stelle noch einmal darauf hinweisen, dass diese Vorgehensweise alles andere als optimal ist. Sie würden sich ja auch nicht in Ihr neues Auto einen stärkeren Motor einbauen lassen, nur weil Sie ständig mit angezogener Handbremse fahren, oder? Natürlich hat eine Aufrüstung der Hardware den Vorteil, dass sie in vielen Fällen relativ schnell und kostengünstig erledigt werden kann. Wenn Sie jedoch nicht den Ursachen für Performance-Probleme auf den Grund gehen, wird eine Hardware-Erweiterung wahrscheinlich nur eine kurzfristige Abhilfe schaffen. Üblicherweise wachsen Performance-Probleme exponentiell mit der zu bewältigenden Datenmenge, sodass diese Probleme mit der Zeit immer größer werden; wenn wir annehmen, dass mit der Zeit auch die zu verarbeitenden Daten anwachsen. Die Hardware-Aufrüstung allein kann hier auf Dauer also keine Abhilfe schaffen.

Auf der anderen Seite ist eine leistungsfähige und ordnungsgemäß konfigurierte Hardware natürlich ein wesentlicher Faktor für ein funktionierendes Gesamtsystem. Hierzu müssen zunächst einmal CPUs und Hauptspeicher ausreichend bemessen sein. Darüber hinaus ist vor allem die Konfiguration des E/A-Systems entscheidend für eine zufriedenstellende Systemleistung. Erinnern Sie sich bitte noch einmal daran, dass vor allem physische E/A-Operationen die Systemleistung entscheidend beeinflussen. In diesem Kapitel werden wir uns daher darauf konzentrieren, wie Sie Ihr E/A-System einrichten sollten, damit physische E/A-Operationen beschleunigt werden.

Physische E/A-Operationen können durchaus verringert werden, zum Beispiel durch geeignete Indizes in Kombination mit ausreichend Hauptspeicher bzw. Datencache. Hiermit haben wir uns in den vorangegangenen Kapiteln eingehend auseinandergesetzt. Letztlich liegen die Daten Ihrer Datenbanken aber auf der Festplatte, und dies bedeutet nun einmal, dass physische E/A-Operationen nicht gänzlich vermieden werden können.

Die Konsequenz aus dieser Erkenntnis ist recht einfach und klar: Sofern physische E/A-Operationen erforderlich sind, sollten sie möglichst schnell sein. Sie werden in diesem Kapitel erfahren, wie Sie Ihr E/A-System im Hinblick auf eine optimale Verwendung durch SQL Server konfigurieren sollten. Außerdem erhalten Sie einige Ratschläge in Bezug auf die Messung der E/A-Performance.

13.1 Physikalisches Datenbanklayout

Es existieren einige generelle Empfehlungen zur Organisation des E/A-Systems eines Servers, der als Gastbetriebssystem für SQL Server konfiguriert werden soll. Diese Aussage bringt uns sofort auf einen sehr wichtigen Punkt: Wann immer möglich, sollten Sie einem SQL Server für ein Produktivsystem eine dedizierte Maschine »spendieren«. Sorgen Sie also dafür, dass außer einer SQL Server-Instanz keine weiteren Prozesse auf dem entsprechenden Server ausgeführt werden.

In Abbildung 13.1 sehen Sie ein Beispiel für die Verteilung der Dateien von SQL Server und des Betriebssystems.

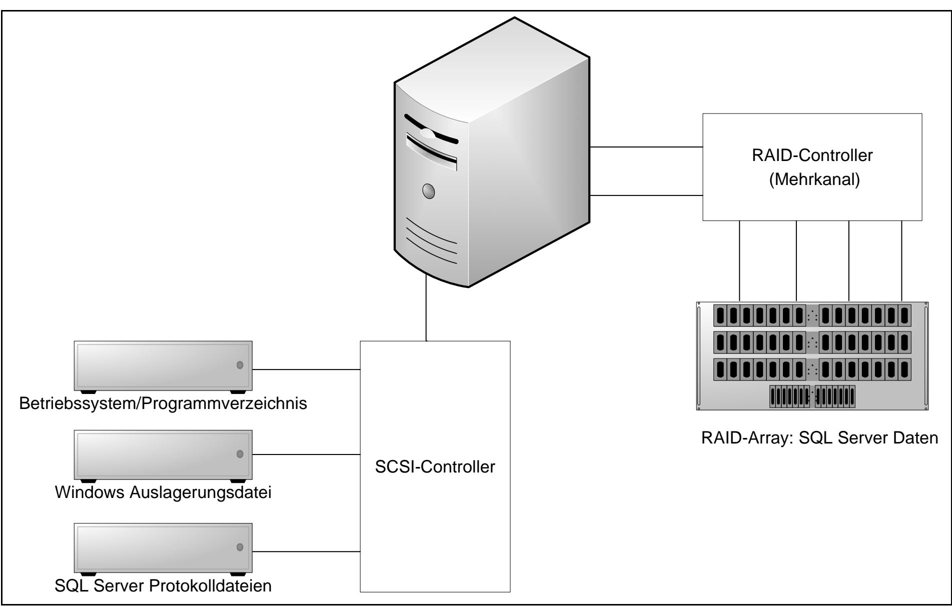


Abbildung 13.1: *Beispiel für ein physikalisches Datenbanklayout*

Die Abbildung macht deutlich, dass die Dateien des Betriebssystems nach Möglichkeit von den Datenbankdateien getrennt gespeichert werden sollten. Darüber hinaus sollten Sie bei der Konfiguration des E/A-Systems die folgenden Punkte bedenken:

- **Speichern Sie auf jeden Fall SQL Server-Protokoll- und Datendateien auf physikalisch unterschiedlichen Laufwerken**. Dies habe ich bereits mehrfach betont. Erinnern Sie sich daran, dass Protokolldateien vor allem sequenziell geschrieben werden, während aus Datendateien in erster Linie zufällig gelesen wird. Auf Grund dieser unterschiedlichen Zugriffsmodi sollten Sie beide Dateiarten niemals »vermischen«, also auf demselben Laufwerk speichern. Eine derartige Vorgehensweise verbietet sich bereits allein aus Gründen der Datensicherheit, ist aber auch aus Gründen der E/A-Performance sehr wesentlich.
- **Je mehr Spindeln, desto besser**. Verwenden Sie lieber viele kleine Festplatten als wenige große. Dadurch steigt der Grad der Parallelität von E/A-Operationen, und der E/A-Durchsatz erhöht sich.

- **Wenn Sie können, speichern Sie Daten- und Indexdateien auf unterschiedlichen Laufwerken**. Das erhöht die Performance von Lookup-Operationen.
- **Legen Sie unterschiedliche Dateigruppen auf unterschiedlichen Datenträgern an**. Bei großen Datenbanken lohnt sich eventuell eine manuelle Aufteilung von Tabellen- und Indexdaten auf unterschiedliche Laufwerke. Hierfür benötigen Sie im SQL Server Dateigruppen. Denken Sie zum Beispiel an einen Join, der zwei Tabellen miteinander verknüpft. Wenn diese beiden Tabellen auf unterschiedlichen Festplatten gespeichert sind, können die Tabellendaten parallel gelesen werden.

  Im Durchschnitt sollte jede Dateigruppe pro CPU-Kern zwischen einem Viertel und einer Datei enthalten. Wenn Ihr Server also insgesamt über acht CPU-Kerne verfügt, dann sollten Sie in einer Dateigruppe zwei bis acht Dateien unterbringen.
- **Speichern Sie eventuell die Datenbank *tempdb* auf einem separaten Laufwerk**. Wie im vorherigen Kapitel bereits einmal erwähnt, kann die Systemdatenbank *tempdb* zum Flaschenhals werden, da alle Verbindungen diese Datenbank gemeinsam verwenden. Die *tempdb* sollte in einer Dateigruppe gespeichert werden, die je CPU-Kern eine Datei enthält. Falls die Datenbank *tempdb* in Ihrer Rangliste der E/A-Operationen weit oben steht, kann es sich auch lohnen, für diese Datenbank eine eigene LUN zu verwenden.
- **Verwenden Sie 64 kByte als Blockgröße für die Formatierung Ihrer NTFS-Festplatten.** Wenn Sie SQL Server eine dedizierte Maschine zur Verfügung stellen, sollten Sie die Festplatten mit dieser Blockgröße formatieren. Eine Blockgröße von 64 kByte ist für SQL Server optimal.
- **Verwenden Sie DISKPART.EXE zur Ausrichtung von Blöcken und Sektoren**. Falls Sie ein Betriebssystem vor Windows Server 2008 verwenden (also zum Beispiel Windows Server 2003), sollten Sie bedenken, dass durch ein Problem in diesen Betriebssystemen Block- und Sektorgrenzen nicht zusammenfallen. Dadurch muss ein Lesevorgang, der einen Block liest, unter Umständen auf zwei Sektoren der Festplatte zugreifen. Dies kann die Performance erheblich beeinträchtigen. Im Internet finden Sie Anleitungen, wie Sie das Systemprogramm DISKPART.EXE verwenden, um eine solche Situation zu vermeiden. Ab Windows Server 2008 (oder auch Windows Vista) existiert das Problem nicht mehr.
- **Legen Sie Ihre Datenbank- und Protokolldateien in der erforderlichen Größe an**. Vertrauen Sie bitte nicht auf die automatische Vergrößerung. Diese funktioniert zwar, führt jedoch auf Dauer zu einer Fragmentierung. Außerdem kostet eine automatische Vergrößerung natürlich ebenfalls Zeit und verringert damit die Abfrageleistung.
- **Planen Sie Ihre Festplatten nach dem erforderlichen E/A-Durchsatz**. Denken Sie daran, dass die E/A-Leistung mit der Anzahl der Festplatten steigt. Festplatten sind in der heutigen Zeit durchaus erschwinglich. Fangen Sie also nicht an, Ihr E/A-System nach Kapazität zu planen. Dieser Ansatz ist überholt! Im nächsten Abschnitt beschäftigen wir uns mit der Messung des E/A-Durchsatzes.
- **Verwenden Sie nach Möglichkeit RAID 1+0 für Ihr E/A-System**. Dadurch erreichen Sie die beste Performance. Setzen Sie kein Software-RAID ein, sondern verwenden Sie in jedem Fall ein Hardware-RAID. Insbesondere sollten Sie RAID 1+0 für das Speichern des Protokolls verwenden.

13.2 Testen des E/A-Systems

Es wird empfohlen, dass Sie das E/A-System eines Servers überprüfen, bevor Sie SQL Server installieren. Für diese Überprüfung gibt es kostenlose Programme, die Sie aus dem Internet beziehen können. SQL Server-E/A-Operationen verwenden bestimmte Muster, die Ihr E/A-System vor spezielle Herausforderungen stellt. Eine Untersuchung des E/A-Systems ist daher ein sehr wichtiger Punkt bei der Evaluierung neuer oder auch bestehender Hardware. Diese Untersuchung sollte zwei Bereiche abdecken:

- **Korrektheit**. Zunächst einmal muss sichergestellt sein, dass Ihr E/A-System die speziellen SQL Server-E/A-Muster überhaupt fehlerfrei verarbeiten kann.
- **Performance**. Mit dieser Überprüfung stellen Sie sicher, dass Ihr E/A-System den Anforderungen hinsichtlich des erforderlichen E/A-Durchsatzes gerecht wird.

13.2.1 Testen auf Korrektheit von E/A-Operationen mit SQLIOSIM

Bevor Sie die Performance eines E/A-Systems überprüfen, sollte zunächst sichergestellt sein, dass das E/A-System überhaupt den Anforderungen von SQL Server genügt, das heißt, dass E/A-Anforderungen also generell korrekt abgearbeitet werden.

Ein auf SQL Server abgestimmtes E/A-System muss in der Lage sein, ganz bestimmte E/A-Muster zu verarbeiten. Dies ist gar nicht so selbstverständlich, wie es den Anschein haben mag. Wenn Sie einmal das Internet nach Fehlerbeschreibungen durchsuchen, die sich auf E/A-Systeme im Zusammenhang mit SQL Server beziehen, werden Sie sich über die Vielfalt der existierenden Fehlerbeschreibungen wundern: eine veraltete Controller-Firmware oder ein fehlerhaft arbeitender Controller-Cache. – Die Ursachen für E/A-Fehler können sehr vielfältig sein.

Microsoft stellt daher ein Programm zur Verfügung, mit dem Sie ein E/A-System auf seine Zuverlässigkeit hin überprüfen können: SQLIOSIM.EXE. Wie der Name bereits vermuten lässt, handelt es sich bei diesem Programm um eine Simulationssoftware. Wird das Programm ausgeführt, simuliert es E/A-Operationen nach dem Muster von SQL Server.

Sie können SQLIOSIM.EXE kostenlos aus dem Internet herunterladen. Bei mir war es sogar Bestandteil der SQL Server-Installation. Ich habe SQLIOSIM.EXE nebst Konfigurationsdateien im Verzeichnis *MSSQL/Binn* meiner SQL Server-Instanz gefunden.

SQLIOSIM ist recht einfach zu handhaben. Über eine Konfigurationsdatei namens *sqliosim.cfg.ini* können Sie festlegen, wie viele Dateien SQLIOSIM insgesamt für die Simulation benutzen soll. Dort legen Sie auch fest, ob eine Datei als SQL Server-Daten- oder -Protokolldatei angesehen werden soll. Beide Dateitypen weisen natürlich unterschiedliche E/A-Muster auf, und dies wird von SQLIOSIM bei der Simulation von E/A-Operationen berücksichtigt. Beim Start von SQLIOSIM werden die in der Konfigurationsdatei festgelegten Parameter angezeigt. Sie haben dort auch die Möglichkeit, diese Parameter in einem entsprechenden Dialog zu überschreiben. Sie können also Daten- und Protokolldateien in den Verzeichnissen und in der Größe erstellen, wie Sie dies für eine spätere SQL Server-Installation planen.

Nach dem Start der Simulation wird direkt ein Protokoll angezeigt (siehe Abbildung 13.2). Dieses Protokoll wird auch automatisch im XML-Format in der Datei *sqliosim.log.xml* gespeichert, sodass Sie die Protokolldaten jederzeit – auch nach Beendigung von SQLIOSIM – einsehen können.

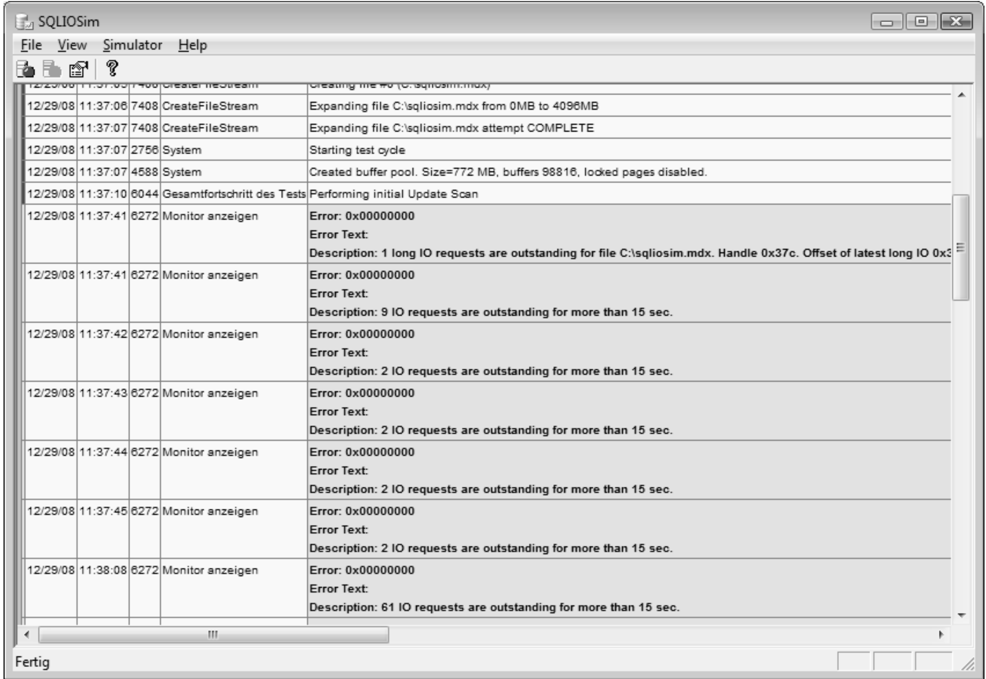


Abbildung 13.2: *Beispielausgabe von SQLIOSIM*

Für das erstellte Protokoll existiert auch ein XSL-Dokument mit Namen *ErrorLog.xslt*, welches das Protokoll in eine tabellarische Darstellung überführt. Wenn Sie das XML-Protokoll und das XSL-Stylesheet in dasselbe Verzeichnis kopieren, lässt sich die XML-Protokolldatei einfach durch einen Doppelklick im Internet Explorer öffnen.

Das Ergebnis in Abbildung 13.2 zeigt deutlich, dass meine kleine Notebook-Festplatte den Anforderungen nicht gewachsen ist. Es wird eine Reihe von Timeout-Fehlern erzeugt, die letztlich besagen, dass mein Notebook nicht für eine SQL Server-Produktivumgebung geeignet ist. SQLIOSIM prüft also auch zeitliche Aspekte des E/A-Systems. Trotzdem sollten Sie SQLIOSIM nicht als ein Werkzeug zur Performance-Analyse betrachten.

SQLIOSIM ist ein Programm zum Testen der Korrektheit Ihres E/A-Systems. Auch wenn gewisse zeitliche Aspekte in die Beurteilung der Korrektheit einfließen, bedeutet dies *nicht*, dass SQLIOSIM auch einen Stresstest in Bezug auf den E/A-Durchsatz vornimmt.

Für das Messen der E/A-Performance gibt es ein separates Programm, das Sie nun im folgenden Abschnitt kennenlernen: SQLIO.EXE.

13.2.2 Messen des E/A-Durchsatzes mit SQLIO

Das Programm SQLIO.EXE gibt es ebenfalls kostenlos im Internet. Mit diesem Programm können Sie SQL Server-ähnliche E/A-Operationen simulieren, wobei Sie Ihr E/A-System einem Stresstest unterziehen. SQLIO ist ein Kommandozeilenprogramm. Über entsprechende Parameter können Sie Zugriffsmuster einstellen und damit Lese- sowie Schreiboperationen in SQL Server-Protokoll- und -Datendateien simulieren. Hierbei haben Sie die Wahl zwischen einer Vielzahl von Zugriffsmustern, sodass Sie E/A-Operationen sowohl für OLTP- als auch für OLAP-Systeme nachstellen können.

SQLIO erlaubt die Ausführung von Lese- und Schreiboperationen jeweils sequenziell oder zufällig (random). Für einen SQLIO Aufruf kann allerdings nur jeweils ein Zugriffsmuster festgelegt werden. SQLIO führt also entweder nur Lese- oder nur Schreiboperationen aus. Ebenso werden diese Operationen entweder sequenziell oder zufällig ausgeführt. Die Konsequenz ist, dass Sie in jedem Fall mehrere Aufrufe von SQLIO mit unterschiedlichen Zugriffsmustern verwenden müssen, um Ihr E/A-System in den Sie interessierenden Kategorien zu messen. Welche Zugriffsmuster für welches System passend sind, erfahren Sie etwas weiter unten in diesem Abschnitt. Im Normalfall werden Sie eine Stapelverarbeitungsdatei erstellen, in welche Sie die unterschiedlichen SQLIO-Aufrufe eintragen.

Die Konfiguration der Zugriffsmuster erfolgt über entsprechende Kommandozeilenparameter. Die wichtigsten Parameter sehen Sie in Abbildung 13.2.

Parameter	Erklärung
-o	Pro Thread ausstehende E/A-Vorgänge in der Warteschlange. Die Anzahl der Threads wird in der Konfigurationsdatei festgelegt.
-LS	Wird dieser Parameter angegeben, misst SQLIO auch die Latenzzeit der Festplatte(n). Sie sollten diesen Parameter angeben, um die Latenzzeit zu messen, wenn Sie ein System testen.
-k	Gibt den Modus der E/A-Operation an. *R* steht für Lesen (Read) und *W* für Schreiben (Write).
-s	Gibt die Dauer des Tests in Sekunden an.
-b	Über diesen Parameter legen Sie die Blockgröße der E/A-Operationen in kByte fest.
-f	Hier wird die Zugriffsart angegeben. Der Wert kann *sequential* für sequenzielle E/A-Operationen oder *random* für zufällige E/A-Operationen sein.
-F	Hier geben Sie den Namen der Konfigurationsdatei an (siehe unten).

Tabelle 13.1: *Kommandozeilenparameter für SQLIO*

In der Konfigurationsdatei legen Sie fest, welche Dateien SQLIO für den Test verwenden soll. Jede Zeile in der Konfigurationsdatei steht hierbei für eine zu verwendende Datei.

Die Dateibeschreibung selber besteht aus den folgenden vier Spalten:

- **Pfad**. Hier steht der vollständige Name der Testdatei, inklusive der Pfadangabe. SQLIO verwendet diese Datei, sofern sie bereits existiert. Falls die Datei noch nicht vorhanden ist, wird sie vor dem Beginn des Tests in der konfigurierten Größe (siehe unten) angelegt. Je nach Dateigröße dauert das Anlegen der Datei eine Weile, wodurch die Ausführung von SQLIO entsprechend verlängert wird.
- **Anzahl der zu verwendenden Threads**. Denken Sie bitte daran, dass die wartenden E/A-Vorgänge, die Sie über den Parameter *–o* konfigurieren, per Thread sind. Dadurch hat der hier angegebene Wert auch Einfluss auf die wartenden E/A-Vorgänge. Sie können zum Beispiel einen Thread mit vielen wartenden E/A-Operationen verwenden oder aber mehrere Threads mit nur jeweils einer wartenden E/A-Operation konfigurieren. Diese beiden Konfigurationsparameter sind also voneinander abhängig. Normalerweise sollten Sie gleiche Messwerte erhalten, wenn das Produkt aus der Anzahl der Threads und den wartenden E/A-Vorgängen ebenfalls gleich ist. Die Anzahl der verfügbaren Prozessorkerne ist ein geeigneter Wert für diesen Parameter.
- **Maske für E/A-Operationen**. Die Maske sollte immer *0x0* sein.
- **Größe der Datei**. Geben Sie hier die Größe der zu verwendenden Testdatei in MByte an. Wählen Sie die Datei nach Möglichkeit so groß, wie die tatsächliche Datenbankdatei später einmal sein wird. Auf jeden Fall sollte dieser Wert deutlich größer (Faktor zwei bis vier) als der Cache des E/A-Systems sein. Schließlich wollen Sie nicht die Leistung des Controller-Cache messen, sondern die Performance der E/A-Operationen.

Eine Beispielkonfigurationsdatei mit Namen *param.txt* wird durch die Installation mitgeliefert. Diese Datei können Sie als Muster für Ihre Konfiguration verwenden.

Eine Zeile in der Konfigurationsdatei sieht zum Beispiel so aus:

```
D:\SqlData\testdata.sqlio 4 0x0 20480
```

Über diese Zeile wird festgelegt, dass eine Testdatei des angegebenen Namens mit einer Größe von 20 GByte verwendet wird. Die gegen diese Datei ausgeführten Tests verwenden das Muster *0x0* für E/A-Operationen sowie vier Threads.

Damit wissen Sie nun also, wie SQLIO generell verwendet wird. Welche Zugriffsmuster Sie für den Test Ihres E/A-Systems verwenden sollten, erfahren Sie im verbleibenden Teil dieses Abschnitts. Außerdem erhalten Sie Tipps für eine generelle Vorgehensweise bei Messungen mit SQLIO.

Testen von E/A-Operationen gegen SQL Server-Datendateien

Die verwendeten (und somit für den Test relevanten) Zugriffsmuster variieren je nach Anwendungstyp. Es ist leider nicht möglich, eine allgemein gültige Aussage zu treffen.

Für OLTP-Datenbanken, die per Definition viele kleine Transaktionen verarbeiten, sollten Sie sich auf das Messen zufälliger Lese- und Schreiboperationen mit einer Blockgröße von 8 kByte konzentrieren. Hierbei ist nicht so sehr der eigentliche Datendurchsatz, gemessen in MByte pro Sekunde, entscheidend. Wesentlich interessanter ist die Anzahl der E/A-Operationen pro Sekunde, auf die Sie bei einer solchen Messung Ihr Augenmerk richten sollten.

Allerdings sind in OLTP-Datenbanken Scan-Operationen nicht ausgeschlossen, wie Sie aus den vorangegangenen Kapiteln wissen. Bei großen Scan-Operationen werden Read Aheads mit einer Blockgröße von bis zu 1.024 kByte in der Enterprise Edition von SQL Server verwendet. Diese Read Ahead-Lesevorgänge sind sequenzieller Natur. Wenn Ihre Anwendungen viele solcher Read Ahead-Operationen auslösen, sollten Sie folglich auch sequenzielle Leseoperationen in verschiedenen Blockgrößen testen.

Normalerweise sind Scan-Operationen aber untypisch für gut entworfene OLTP-Anwendungen. Konzentrieren Sie sich daher bei den Tests auf die Messung der Anzahl zufälliger Leseoperationen pro Sekunde mit einer Blockgröße von 8 kByte.

Erinnern Sie sich bitte noch einmal daran, dass das Schreiben geänderter Datenseiten größtenteils asynchron erfolgt. Eine Messung des Schreibdurchsatzes ist deshalb nicht ganz so wichtig. Allerdings sollten natürlich beim Auslösen eines `CHECKPOINT` die erforderlichen Datenseiten möglichst schnell aus dem Cache auf die Platte übertragen werden, weil ansonsten Ihre System-Performance leidet. Daher können Sie auch die Anzahl der Schreiboperationen pro Sekunde mit einer Blockgröße von 8 kByte in Ihre Messung mit einbeziehen. Diese Schreiboperationen sind ebenfalls zufälliger Natur.

In OLAP-Datenbanken sieht es etwas anders aus. Dort haben Sie es größtenteils mit sequenziellen Lese- und Schreiboperationen zu tun. Für Leseoperationen beträgt die Blockgröße zwischen 8 und 256 kByte; in der Enterprise Edition auch bis zu 1.024 kByte. Auch Schreiboperationen (BULK LOAD) werden größtenteils sequenziell ausgeführt. Hier ist die maximale Blockgröße auf 128 kByte begrenzt.

Bei diesen sequenziellen Lese- und Schreiboperationen kommt es auf den Datendurchsatz, gemessen in MByte pro Sekunde an. Konzentrieren Sie sich also auf diesen Messwert; die Anzahl der E/A-Operationen pro Sekunde ist in diesem Fall nicht interessant.

Testen von protokolltypischen E/A-Operationen

E/A-Operationen gegen SQL Server-Protokolldateien sind stets sequenziell. Erinnern Sie sich bitte noch einmal an unser Experiment aus Kapitel 2 und an das dort Gesagte: Bevor eine Transaktion als abgeschlossen betrachtet werden kann, müssen zunächst die erforderlichen Einträge in der Protokolldatei erstellt werden. Genau genommen werden diese Einträge an das Protokoll angefügt, und zwar synchron. Das wichtigste Szenario für das Messen der protokollspezifischen E/A-Operationen sind daher sequenzielle Schreibvorgänge. Je nach Transaktionsumfang haben diese Schreiboperationen eine Blockgröße von bis zu 60 kByte.

Aus dem Transaktionsprotokoll werden auch Daten gelesen. Dies ist zum Beispiel der Fall, wenn Sie eine Sicherung des Protokolls erstellen oder etwa eine Spiegelung bzw. transaktionale Replikation einsetzen. Auch die Leseoperationen erfolgen sequenziell, wobei die Blockgröße maximal etwa 120 kByte beträgt.

Konzentrieren Sie sich bei den Messungen für die E/A-Performance auf sequenzielle Schreiboperationen in Blockgrößen von 4 bis zu 64 kByte und betrachten Sie vor allem den Datendurchsatz, also die übertragenen MByte pro Sekunde. Bei vielen kleinen Transaktionen (also in OLTP-Systemen) ist die Latenzzeit der Festplatte(n) ebenfalls ein wesentlicher Faktor, den Sie einer Beurteilung unterziehen sollten. Es gilt, die Protokolleinträge so schnell wie nur irgend möglich auf die Festplatte zu schreiben. Geringe

Latenzzeiten sind deshalb sehr wichtig. Erinnern Sie sich noch einmal an das Beispiel aus Kapitel 2, in dem einzig und allein das Schreiben des Transaktionsprotokolls für die Ausführungsdauer bestimmend war.

Generelle Vorgehensweise für Messungen mit SQLIO

Beginnen Sie zunächst mit kleineren Ausführungszeiten für Ihre Messungen, um einen Gesamtüberblick zu bekommen. Anhand dieser Gesamtübersicht können Sie dann Szenarien konfigurieren und schließlich die Messungen über einen längeren Zeitraum durchführen. Es ist sehr wichtig, dass Sie einen ausreichend großen Zeitraum beobachten, weil viele Controller intelligente Caches besitzen, die erst nach einer gewissen Anlernphase richtig »auf Touren kommen«.

Was Sie für die Tests unbedingt benötigen, ist Zeit. Für die Messungen in den unterschiedlichen Konfigurationen bzw. Szenarien ist eine Reihe von Experimenten erforderlich. Diese Experimente müssen natürlich geplant und durchgeführt werden. Es wird nicht möglich sein, dass Sie diese Aufgabe nebenbei bewältigen; planen Sie für die Untersuchungen also besser eine Woche ein. Insbesondere müssen Sie mit einer unterschiedlichen Anzahl wartender E/A-Vorgänge in unterschiedlichen Blockgrößen experimentieren, um Ihr System beurteilen zu können.

Wir kommen etwas weiter unten noch einmal auf Blockgrößen und Warteschlangen zurück.

SQLIO gibt die Messergebnisse auf dem Bildschirm aus. Wie bei Kommandozeilenprogrammen gewohnt, haben Sie aber auch die Möglichkeit, die Ausgabe in eine Datei umzuleiten.

In Abbildung 13.3 sehen Sie ein Beispiel für eine Ausgabe von SQLIO.

```
Administrator: C:\Windows\system32\cmd.exe

C:\Program Files\SQLIO>sqlio -kR -s60 -frandom -b8 -o2 -LS -Fparam.txt
sqlio v1.5.SG
using system counter for latency timings, 14318180 counts per second
parameter file used: param.txt
        file c:\testfile.dat with 1 thread (0) using mask 0x0 (0)
1 thread reading for 60 secs from file c:\testfile.dat
        using 8KB random IOs
        enabling multiple I/Os per thread with 2 outstanding
using specified size: 1000 MB for file: c:\testfile.dat
initialization done
CUMULATIVE DATA:
throughput metrics:
IOs/sec:    76.39
MBs/sec:     0.59
latency metrics:
Min_Latency(ms): 3
Avg_Latency(ms): 25
Max_Latency(ms): 238
histogram:
ms: 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24+
%:  0  0  0  0  0  0  0  0  0  0  1  1  1  2  1  2  3  3  4  4  4  5  5  5 58
```

Abbildung 13.3: *Von SQLIO ausgegebene Messergebnisse*

Bitte lassen Sie sich von den mehr als miserablen Messwerten nicht verwirren. Das Experiment wurde mit der Festplatte meines etwas betagten Notebooks durchgeführt, das ja bereits von SQLIOSIM als für SQL Server unzureichend eingestuft wurde (siehe nochmals Abbildung 13.2). Letztlich erhalten Sie eine Reihe von Messwerten, die Auskunft über das Verhalten Ihres E/A-Systems geben. Wie diese Messwerte interpretiert werden, erfahren Sie etwas weiter unten.

Leider kann SQLIO nur das in Abbildung 13.3 dargestellte Protokoll erzeugen. Eine Ausgabe in einem anderen Format, etwa grafisch oder tabellarisch, ist nicht möglich. Das Übertragen der Messergebnisse in eine Excel-Tabelle müssen Sie also manuell vornehmen, was etwas mühselig ist. Hier hilft der Systemmonitor weiter. Wenn Sie die folgenden Indikatoren des Objekts *Logischer Datenträger (Logical Disk)* für eine bestimmte LUN hinzufügen, erhalten Sie wahlweise eine grafische Darstellung oder einen Bericht. Lesen Sie noch einmal in Kapitel 4 nach, wenn Sie nicht mehr wissen, wie Sie den Systemmonitor hierfür einsetzen.

- **Durchschnittl. Warteschlangenlänge des Datenträgers (Avg. Disk Queue Length)**. Der Indikator zeigt an, wie viele E/A-Operationen in der Warteschlange auf Abarbeitung warten. Sie können die Länge der Warteschlange bei einem SQLIO-Durchlauf über den Parameter *–o* und die Anzahl der verwendeten Threads bestimmen.
- **Mittlere Sek./Schreibvorgänge (Avg. Disk sec/Write)**. Verwenden Sie diesen Indikator beim Test von Schreiboperationen. Der Indikator gibt die durchschnittliche Zeitdauer für einen Schreibvorgang in Sekunden an und sollte einen möglichst geringen Wert aufweisen.
- **Bytes geschrieben/s (Disk Byte Write/sec)**. Diesen Indikator sollten Sie ebenfalls beim Test von Schreiboperationen hinzufügen. Er gibt den eigentlichen Datendurchsatz für einen Schreibvorgang an.
- **Mittlere Sek./Lesevorgänge (Avg. Disk sec/Read)**. Für Tests von Leseoperationen verwenden Sie diesen Indikator. Er gibt die durchschnittliche Zeitdauer für eine Leseoperation in Sekunden an.
- **Bytes gelesen/s (Disk Byte Read/sec)**. Dieser Wert enthält den Datendurchsatz für Leseoperationen, sollte also für den Test von Leseoperationen hinzugefügt werden.

Wie gesagt, sollten Sie sich zu Beginn zunächst eine Übersicht verschaffen. Hierzu können Sie die folgenden SQLIO-Aufrufe für unterschiedliche Kategorien von E/A-Mustern verwenden:

Messen der Anzahl zufälliger Lese- und Schreiboperationen in einem OLTP-System Wie etwas weiter oben erläutert, sollte die Blockgröße in diesem Fall 8 kByte betragen. Die folgenden beiden SQLIO-Aufrufe messen den Lese- und Schreibdurchsatz:

```
sqlio -kR -s300 -frandom -o8 -b8 -LS -param.txt
timeout /t60
sqlio -kW -s300 -frandom -o8 -b8 -LS -param.txt
```

Der *Timeout* ist erforderlich, damit das E/A-System sich nach jedem Test erst einmal »beruhigen« kann und die anstehenden E/A-Operationen beendet werden, bevor ein neuer Test beginnt.

Sequenzielles Lesen und Schreiben in einem OLAP-System In diesem Fall müssen Sie unterschiedliche Blockgrößen sowie sequenzielle Lese- und Schreiboperationen verwenden:

```
sqlio -kR -s300 -sequential -o8 -b8 -LS -param.txt
sqlio -kR -s300 -sequential -o8 -b64 -LS -param.txt
sqlio -kR -s300 -sequential -o8 -b128 -LS -param.txt
sqlio -kR -s300 -sequential -o8 -b256 -LS -param.txt
sqlio -kR -s300 -sequential -o8 -b1024 -LS -param.txt
sqlio -kW -s300 -sequential -o8 -b64 -LS -param.txt
sqlio -kW -s300 -sequential -o8 -b128 -LS -param.txt
sqlio -kW -s300 -sequential -o8 -b256 -LS -param.txt
```

Falls Sie nicht die Enterprise Edition von SQL Server einsetzen, können Sie den Aufruf mit der Blockgröße von 1.024 kByte auslassen.

Die *Timeout*-Aufrufe wurden hier aus Gründen der Übersichtlichkeit weggelassen. Sie sollten aber daran denken, jeweils zwischen zwei SQLIO-Aufrufen eine Pause einzulegen.

Sequenzielles Schreiben des Protokolls Die Länge der Warteschlange kann für diesen Test etwas geringer sein. Erinnern Sie sich bitte an die Kapitel 2 und 4: Falls die Warteschlange für das Schreiben des Protokolls zu groß wird, benötigen Sie ein schnelleres Laufwerk oder ein verringertes Transaktionsvolumen. Daher testen wir mit einer Warteschlangenlänge von vier.

Die folgenden Aufrufe simulieren das Schreiben in das Protokoll in unterschiedlichen Blockgrößen:

```
sqlio -kW -s300 -sequential -o4 -b4 -LS -param.txt
sqlio -kW -s300 -sequential -o4 -b8 -LS -param.txt
sqlio -kW -s300 -sequential -o4 -b16 -LS -param.txt
sqlio -kW -s300 -sequential -o4 -b32 -LS -param.txt
sqlio -kW -s300 -sequential -o4 -b64 -LS -param.txt
```

Auch hier wurden wiederum die *Timeout*-Aufrufe weggelassen.

Noch einmal zur Erinnerung: In einem OLTP-System ist eine geringe Latenzzeit besonders wichtig.

Worauf Sie bei allen Messungen achten müssen, ist, Ihr E/A-System unterhalb der Sättigung zu testen. Wie Sie hierfür vorgehen, erfahren Sie im folgenden Abschnitt.

Ermitteln der Sättigung des E/A-Systems Die Sättigung des E/A-Systems gibt den maximal erreichbaren Datendurchsatz an, den Sie in einer bestimmten Konfiguration erreichen können. Sie können die Sättigung je Blockgröße, Zugriffsart (Lesen oder Schreiben) und Zugriffsmodus (sequenziell oder zufällig) experimentell ermitteln. Hierzu müssen Sie in nacheinander folgenden Aufrufen von SQLIO für eine Zugriffsart, einen Zugriffsmodus und eine Blockgröße die Anzahl der ausstehenden E/A-Vorgänge über den Parameter *–o* sukzessive erhöhen. Für eine Blockgröße von 64 kByte könnten die entsprechenden SQLIO-Aufrufe so aussehen:

```
sqlio -kW -s120 -fsequential -o1 -b64 -LS -Fparam.txt
sqlio -kW -s120 -fsequential -o2 -b64 -LS -Fparam.txt
sqlio -kW -s120 -fsequential -o3 -b64 -LS -Fparam.txt
sqlio -kW -s120 -fsequential -o4 -b64 -LS -Fparam.txt
sqlio -kW -s120 -fsequential -o5 -b64 -LS -Fparam.txt
sqlio -kW -s120 -fsequential -o6 -b64 -LS -Fparam.txt
sqlio -kW -s120 -fsequential -o7 -b64 -LS -Fparam.txt
sqlio -kW -s120 -fsequential -o8 -b64 -LS -Fparam.txt
```

In der Konfigurationsdatei geben Sie für diese Messung eine Datei sowie nur einen Thread an, also etwa so:

```
D:\SqlData\testdata.sqlio 1 0x0 20480
```

Beobachten Sie bitte beim Ablauf des Tests die etwas weiter oben aufgelisteten Indikatoren des Systemmonitors. Sobald der Datendurchsatz bei einer Erhöhung der Warteschlange nicht mehr anwächst, ist die Sättigung erreicht. Sie erkennen diesen Zustand sehr schön in der grafischen Ansicht des Systemmonitors. An der Stelle, an der nur noch die Warteschlange und die für eine Lese- oder Schreiboperation benötigte Zeit anwachsen, ohne dass sich auch die gemessene Übertragungsrate erhöht, ist die Sättigung erreicht. Sie haben dann den maximalen Datendurchsatz gefunden – wohlgemerkt: für *eine* bestimmte Blockgröße, *eine* Zugriffsart, *einen* Zugriffsmodus und *eine* LUN. Sie müssen die entsprechenden SQLIO-Aufrufe also für unterschiedliche Zugriffsmodi, Blockgrößen, Zugriffsarten und logische Laufwerke wiederholen, um den maximal erreichbaren Datendurchsatz je Einstellung zu ermitteln. Ein gutes Argument dafür, dass Sie für SQLIO-Experimente ausreichend Zeit einplanen sollten, nicht wahr?

Testen mehrerer Pfade In der Konfigurationsdatei können Sie mehrere Dateien angeben, die dann bei einem Test gleichzeitig gelesen bzw. geschrieben werden. Beginnen sollten Sie zunächst mit nur einer Datei. Sobald Sie gesicherte Erkenntnisse beim Experimentieren mit einer Datei gewonnen haben, können Sie Ihre Versuche auf mehrere Dateien ausdehnen.

Interpretieren der SQLIO-Ergebnisse

Für die Auswertung der Messergebnisse sollten Sie die folgenden Punkte beachten:

- Stellen Sie sicher, dass außer Ihnen niemand sonst das E/A-System verwendet. Andernfalls sind Ihre Testergebnisse wertlos.
- Wenn Sie die Blockgröße erhöhen, wird normalerweise auch die Latenzzeit anwachsen.
- Denken Sie daran, dass Sie für zufällige E/A-Operationen die Anzahl der Operationen pro Sekunde betrachten. Für sequenzielle E/A-Operationen ist dagegen der Datendurchsatz in MByte pro Sekunde entscheidend.
- Wenn Sie beginnen mehrere Dateien zu verwenden, sollte der Datendurchsatz anwachsen.
- Heben Sie Ihre Messergebnisse auf. Sie sind damit in der Lage, Ihren Hardwarelieferanten zu konsultieren, wenn Sie alleine nicht in der Lage sind, die Ergebnisse zu interpretieren.

13.3 Zusammenfassung

Ein optimal konfiguriertes E/A-System ist ein wesentlicher Faktor für eine zufriedenstellende Performance eines Systems. In diesem Kapitel haben Sie wichtige Hinweise erhalten, die Sie bei der Einrichtung eines E/A-Systems beachten sollten, um es speziell an die Erfordernisse von SQL Server anzupassen.

Darüber hinaus haben Sie erfahren, wie Sie ein bestehendes E/A-System unter Verwendung von SQLIOSIM und SQLIO im Hinblick auf Funktionalität und Datendurchsatz überprüfen können. Denken Sie bitte daran, dass diese Überprüfungen zur Validierung des Systems dienen und daher nach Möglichkeit vor der Installation von SQL Server durchgeführt werden sollten.

Teil 4

Anhänge

A Häufige Fehler und Irrtümer 335
B Literatur 345

A Häufige Fehler und Irrtümer

In diesem Anhang habe ich für Sie Fehler und Irrtümer zusammengefasst, die mir immer wieder begegnen. Teilweise handelt es sich hierbei um Wiederholungen, also Themen, die Ihnen aus den einzelnen Kapiteln dieses Buches bereits bekannt sein sollten. In einigen Fällen werden diese bekannten Themen noch einmal vertieft. Darüber hinaus erfahren Sie auch einige Neuigkeiten, die in diesem Anhang Platz gefunden haben, weil eine Zuordnung zu einem der Kapitel nicht eindeutig möglich war.

Ich hoffe, dass Ihnen dieser Anhang als Nachschlagewerk gute Dienste leistet und Sie die hier aufgezählten Ratschläge in Erinnerung behalten. Ziel ist es auch, Sie zu ermuntern, über Probleme einmal anders nachzudenken und nicht irgendwelche im Internet präsentierten Erkenntnisse ohne Überlegung als Richtlinien herzunehmen. Die vorgestellten Hinweise werden dabei mehr oder weniger willkürlich aufgelistet und sind keinesfalls nach ihrer Wichtigkeit, Wirksamkeit oder Bedeutung angeordnet.

A.1 Vertrauen auf RAID 5

Ein RAID 5 Array ist für Schreiboperationen ganz klar nicht die optimale Lösung. Vor allem das Protokoll sollten Sie nicht auf einem RAID 5 Array speichern. Hier ist ein RAID 1+0 eindeutig vorzuziehen. Vor allem ist immer noch häufig eine Konstellation anzutreffen, bei der sowohl Daten- als auch Protokolldateien auf einer LUN in einem RAID 5 Array gespeichert werden. Eine solche Konfiguration ist nicht optimal.

A.2 Planung des E/A-Systems nach Kapazität

Diese Vorgehensweise sollte der Vergangenheit angehören. Natürlich sollten Sie abschätzen können, welches Datenvolumen zu erwarten ist. Eine Planung des E/A-Systems nach der Größe der Datenbanken ist jedoch überholt. Planen Sie Ihr E/A-System nach dem erforderlichen E/A-Durchsatz und nicht nach der erforderlichen Festplattenkapazität.

A.3 Gruppierter Index für den Primärschlüssel

SQL Server erstellt standardmäßig für den Primärschlüssel einer Tabelle einen gruppierten Index, sofern Sie nichts anderes festlegen. Es gibt genügend Beispiele dafür, wann ein gruppierter Primärschlüssel alles andere als die optimale Wahl ist. Sie können nur einen gruppierten Index je Tabelle auswählen. Denken Sie also bitte sorgfältig darüber nach, welche Spalten Sie in den gruppierten Index aufnehmen.

Lesen Sie noch einmal in Kapitel 9 nach, welche Kriterien für die Auswahl des gruppierten Index ausschlaggebend sind.

A.4 Verwenden von GUIDs als Primärschlüssel

In der objektorientierten Anwendungsentwicklung benötigt jedes Objekt eine eindeutige Identität. Anwendungsentwickler verwenden mit Vorliebe GUIDs für die Festlegung dieser Identität, denn in diesem Fall hat ein Objekt stets eine eindeutige und universell gültige ID. Als Konsequenz finden Sie in Datenbanktabellen sehr oft künstliche Primärschlüssel vom Typ GUID. In SQL Server ist dies der Datentyp `UNIQUEIDENTIFIER`. Der Primärschlüssel identifiziert damit nicht mehr nur eine Zeile innerhalb einer Tabelle, sondern ermöglicht eine eindeutige Identifikation dieser Zeile im gesamten »Universum«. Sie sollten sehr gut überlegen, ob Sie wirklich diese universell gültige Identifikation benötigen. Spalten vom Datentyp `UNIQUEIDENTIFIER` benötigen 16 Byte Speicherplatz, wohingegen der Datentyp `INTEGER`, der für eine Identifikation innerhalb der Tabelle in den meisten Fällen ausreicht, nur 4 Byte erfordert.

Zunächst einmal klingt dieser Unterschied von acht Byte nicht sehr dramatisch – und eigentlich ist er es auch nicht. In einer Tabelle mit 1.000.000 Zeilen sind das nicht einmal 8 MByte. Sie sollten jedoch bei dieser Kalkulation nicht vergessen, dass der Primärschlüssel einer Tabelle für Fremdschlüsselbeziehungen auch in die »zu n-Seite« einer 1:n-Beziehung dupliziert werden muss, und somit haben Sie einen erhöhten Speicherplatzbedarf auch in diesen Tabellen. Hinzu kommt, dass sowohl für den Primärschlüssel als auch für Fremdschlüsselbeziehungen Indizes existieren, für die ebenfalls ein erhöhter Speicherbedarf zu Buche schlägt. Sie sollten nicht unterschätzen, welche gravierenden Auswirkungen ein solches Design auf die erforderlichen E/A-Operationen und den größeren Arbeitsspeicherbedarf haben kann!

Dieses Problem kann im Übrigen in Kombination mit dem vorherigen auftreten, wenn Sie für die `UNIQUEIDENTIFIER`-Primärschlüssel auch noch den gruppierten Index verwenden. In diesem Fall verlangsamen sich Ihre `INSERT`-Operationen ganz erheblich.

A.5 Verwenden von Autogrow

SQL Server erlaubt auf einfache Weise das Anlegen einer Datenbank durch das folgende Kommando:

```
create database DB1;
```

Damit wird die Datenbank in der Standardgröße (der Größe der Datenbank *model*) angelegt. Sobald der Speicherlatz bei Datenänderungen nicht mehr ausreicht, wird diese Datenbank automatisch vergrößert. Diese automatische Vergrößerung (englisch: Autogrow) kostet zum einen Zeit, und Ihre Transaktionen werden entsprechend langsamer, falls sie mit einem *Autogrow*-Ereignis kollidieren. Viel schlimmer ist aber die Tatsache,

dass durch die automatische Vergrößerung auch die Fragmentierung Ihrer Datenbank nach und nach erhöht wird – und das kann die Leistung ganz erheblich beeinträchtigen.

Planen Sie also bitte Ihre Datenbankgrößen sorgfältig und legen Sie die Datenbanken in der erforderlichen Größe an, um ein *Autogrow* nach Möglichkeit zu vermeiden.

A.6 Verwenden von SHRINK DATABASE

In vielen Wartungsplänen wird auch eine Verkleinerung von Datenbanken durchgeführt. So etwas sollten Sie allerdings unbedingt vermeiden. Bei einer Verkleinerung einer Datenbank werden Lücken in den Datendateien geschlossen, sodass am Ende der Datei Speicher frei wird und die Datei einfach abgeschnitten werden kann. Dabei werden Datenseiten vom Ende der Datei in eine weiter vorn existierende Lücke verschoben. Dies ist per Definition eine Fragmentierung. Eine Verkleinerung einer Datenbank oder von Datenbankdateien wird also die Fragmentierung der Datenbank erhöhen. Allein aus diesem Grund sollten Sie eine Verkleinerung von Datenbankdateien unbedingt unterlassen.

A.7 Aktualisieren der Statistiken nach dem Re-Index

Gerade beim Einsatz von Wartungsplänen habe ich oft beobachtet, dass eine Aktualisierung der Statistiken nach der Neuorganisation der Indizes durchgeführt wird. Denken Sie bitte daran, dass bei einem Indexaufbau auch die zum Index gehörende Statistik neu erstellt wird – und zwar auf der Basis aller Tabellenzeilen. Eine bessere Statistik können Sie nicht bekommen. Wenn Sie nach dem Indexaufbau `sp_updatestats` aufrufen, überschreiben Sie die zuvor erstellte, genaue Statistik durch eine Statistik, die lediglich auf einer zufälligen Stichprobe basiert.

A.8 Optimierung = leistungsfähigere Hardware anschaffen

Optimierung hat viele Ansatzpunkte – und die Optimierung der Hardware ist nur einer davon. Sie sollten zunächst herausfinden, wo das Problem liegt, bevor Sie neue Hardware anschaffen. In den meisten Fällen ist eine Optimierung des SQL-Code oder der Indizes vielversprechender als eine sofortige Aufrüstung der Hardware, wenn Performance-Probleme auftreten.

A.9 Scans sind generell schlecht

Wenn Sie Scans in Abfrageplänen beobachten, dann ist dies zunächst noch kein Grund, in Panik zu verfallen. In OLAP-Anwendungen sind Scans sogar an der Tagesordnung. Es gibt durchaus Abfragen, bei denen ein Scan einer Tabelle oder eines Index die optimale Möglichkeit darstellt. In OLTP-Anwendungen sollten Sie dagegen die Ursache für einen Scan herausfinden. Wenn eine Scan-Operation viele Zeilen verarbeitet, aber nur wenige Zeilen zurückgibt, fehlt möglicherweise ein Index. Falls eine Abfrage in einer OLTP-Anwendung viele Zeilen zurückgibt, kann ein Scan der optimale Operator sein. In diesem Fall sollten Sie jedoch untersuchen, warum es erforderlich ist, dass eine Abfrage eine große Anzahl Zeilen zurückliefert.

A.10 Dynamisches SQL ist »ungesund«

Generell sollten Sie dynamisches SQL vermeiden. Die damit verbundenen Risiken, etwa SQL Injection oder das »Überfluten« des Plancache, sind einfach zu hoch. Es gibt jedoch auch Ausnahmen, in denen der Einsatz von dynamischem SQL Performance-Vorteile hervorbringt. Denken Sie an Probleme mit Parametrisierung oder Parameter-Sniffing, die durch dynamisches SQL gelöst werden können. Lesen Sie noch einmal in Kapitel 9 nach, falls Sie sich nicht mehr erinnern.

Verstehen Sie dies bitte nicht als Aufforderung, dynamisches SQL bedenkenlos zu verwenden. Dazu rate ich Ihnen ganz gewiss nicht. Sie sollten dynamisches SQL aber nicht grundsätzlich »verteufeln«, sondern lediglich nachdenken, ob ein existierendes Problem mit dynamischem SQL gelöst werden kann. Falls Sie dynamisches SQL einsetzen, treffen Sie bitte unbedingt Maßnahmen, um SQL Injection zu vermeiden. In der Online-Dokumentation sowie im Internet finden Sie hierzu entsprechende Anleitungen und Techniken.

A.11 Verwenden automatisch erstellter UNIQUE-Indizes

Denken Sie daran, dass für eine `UNIQUE`-Einschränkung automatisch ein Index erstellt wird. Dieser Index enthält keine eingeschlossenen Spalten, ist also für die meisten Abfragen nicht abdeckend. So kann es Ihnen zum Beispiel passieren, dass der Datenbankoptimierungsratgeber das Anlegen des Index noch einmal empfiehlt, wobei eine Reihe von Spalten in den Index eingeschlossen werden soll. Sie sollten also überlegen, ob der automatisch erstellte Index für die `UNIQUE`-Einschränkung Ihrer Abfragen geeignet ist, oder ob Sie den Index doch lieber manuell erstellen und diverse Spalten in den Index einschließen. Zwei Indizes auf derselben Spalte, einmal ohne und einmal mit eingeschlossenen Spalten, sind meist nicht erforderlich beziehungsweise optimal.

A.12 Cursors sind in jedem Fall zu vermeiden

Generell sollten Sie Cursors nur dann einsetzen, wenn Ihnen für ein Problem keine mengenbasierte Lösung einfällt. In den meisten Fällen ist eine Cursor-basierte Lösung langsamer als eine vergleichbare mengenorientierte Lösung. Cursors sind jedoch nicht grundsätzlich schlecht. Es gibt durchaus Fälle, in denen ein Cursor deutlich schneller zum Ziel führt als ein mengenbasiertes Äquivalent.

Schauen Sie sich bitte das folgende Beispiel an, in dem eine laufende Summe berechnet werden soll. Tabelle A.1 veranschaulicht diese Berechnung.

ID	Wert	Summe
1	10	10
2	12	22
3	10	32
4	20	52
5	25	77

Tabelle A.1: *Berechnung einer laufenden Summe*

In der dritten Spalte wird jeweils die laufende Summe aller bislang in der mittleren Spalte ausgegebenen Werte dargestellt.

Wir legen zunächst eine Testtabelle an und fügen 200.000 Zeilen ein:

```
use QueryTest;
-- Tabelle anlegen und befüllen
if (object_id('T1', 'U') is not null)
  drop table T1
go
create table T1(id int identity(1,1) not null primary key
               ,x decimal(8,2) not null default 0
               ,Spalten char(100) not null default '#')
go

insert T1(x)
  select 0.01 * abs(checksum(newid()) % 200000)
   from Numbers
  where n <= 200000
```

Um nun die laufende Summe für die Spalte `x` zu berechnen, können verschiedene Ansätze verfolgt werden. Eine Möglichkeit ist ein SELF JOIN:

```
select T1.id,sum(T2.x) as rt
  from T1
       inner join T1 as T2 on T2.id <= T1.id
 group by T1.Id
```

Diese Abfrage dauert auf meinem PC 60 Sekunden.

Eine weitere Möglichkeit ist die Verwendung einer Unterabfrage:

```
select T1.id
      ,(select sum(T2.x) from T1 as T2 where T2.id <= T1.Id) as rt
  from T1
```

Für die Ausführung dieser Abfrage waren bereits 117 Sekunden erforderlich.

Schließlich führt hier auch ein geeigneter Cursor zum Ziel:

```
create table #t(id int not null primary key,s decimal(16,2) not null)
declare @id int
       ,@x decimal(8,2)
       ,@s decimal(16,2)
set @s = 0
declare #c cursor fast_forward for
 select id,x from T1 order by id
open #c
 while (1=1)
  begin
    fetch next from #c into @id, @x
    if (@@fetch_status != 0) break
    set @s = @s + @x
    if @@trancount = 0
      begin tran
    insert #t values(@id,@s)
    if (@id % 10000) = 0
      commit
  end
if @@trancount > 0
  commit
close #c
deallocate #c
select * from #t order by id
drop table #t
```

Innerhalb der Cursor-Schleife werden die Summen in einer temporären Tabelle gespeichert, deren Inhalt am Ende ausgegeben wird. Diese Abfrage benötigt insgesamt nur sieben Sekunden und ist damit fast neun Mal schneller als die schnellere der beiden mengenbasierten Lösungen.

Mit Cursors verhält es sich also genau so wie mit anderen Features auch: Mit Überlegung und Bedacht eingesetzt, sind sie durchaus nützlich.

A.13 Mehr Einschränkungen in der WHERE-Klausel senken die Abfragekosten

Zum Abschluss möchte ich Ihnen noch ein Beispiel präsentieren, das einen verbreiteten Trugschluss aufklärt. Für dieses Beispiel legen wir zunächst eine Tabelle an:

```
if (object_id('Produkt') is not null)
  drop table Produkt
go
create table Produkt
 (
   ProduktId integer identity(1,1) primary key not null
  ,Nummer nvarchar(20) not null
  ,Status smallint not null default 0
  ,Spalten nchar(200) null
 )
go
insert Produkt(Nummer, Status)
  select char(65 + ((abs(checksum(newid())-1)) % 26))
          + '-' + right('0000000000' + cast(abs(checksum(newid())) % 100000 as
nvarchar(12)),8)
        ,abs(checksum(newid())) % 3
    from Numbers
   where n <= 30000
go
create index Ix_ProdNr on Produkt(Nummer)
```

Wir verwenden hier wieder eine Tabelle `Produkt`, für die ein nichtgruppierter Index auf der Spalte `Nummer` erstellt wird.

Wenn wir nun die `ProduktId` für eine bestimmte `Nummer` suchen, führt die folgende Abfrage zum Ziel:

```
select ProduktId
  from Produkt
 where Nummer = 'Q-00020428'
```

Wie zu erwarten, wird der nichtgruppierte Index auf der Spalte `Nummer` verwendet. Abbildung A.1 zeigt den Ausführungsplan. Die Abfragekosten betragen 0,003.

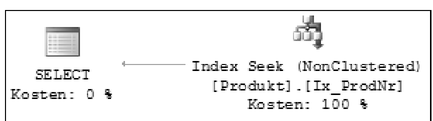


Abbildung A.1:
Ausführungsplan für die Suche einer ProduktId nach der Nummer

Wenn zur WHERE-Klausel nun eine weitere Bedingung unter Verwendung des AND-Operators hinzugefügt wird, ist die Erwartung sicherlich, dass die Abfragekosten sinken. Die Filterbedingung ist ja selektiver; und dadurch sollten die Kosten geringer werden. Das Gegenteil ist jedoch der Fall! Wenn wir die folgende Abfrage

```
select ProduktId
  from Produkt
 where Nummer = 'Q-00020428'
   and Status = 1
```

ausführen, stellen wir fest, dass sich die Abfragekosten erhöht haben. In Abbildung A.2 sehen Sie den Ausführungsplan.

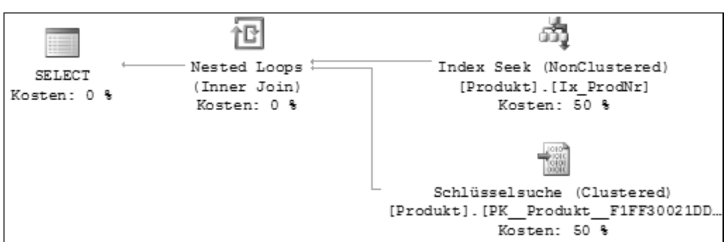


Abbildung A.2: *Ausführungsplan für die Suche nach Nummer und Status*

Die Abfragekosten sind mit 0,006 doppelt so hoch wie zuvor. Die Ursache hierfür ist klar: Die Suche wird über den Index auf der Spalte Nummer durchgeführt. Aus diesem Index kann der Wert für die Spalte Status aber nicht ermittelt werden. Hierfür ist eine zusätzliche Schlüsselsuche im gruppierten Index erforderlich: Dadurch wird die Abfrage teurer. Eine Lösung wäre, die Spalte Status in den Index mit aufzunehmen – dann würde die Schlüsselsuche eingespart. Bleibt noch zu sagen, dass der Optimierer hierfür keine Empfehlung ausspricht, es wird also kein fehlender Index bemängelt. Sofern Sie die Abfrage mit dem Datenbankoptimierungsratgeber analysieren, ist ein entsprechender Index in der Liste der Vorschläge enthalten.

Denken Sie bitte also daran, dass mehr Bedingungen in der WHERE-Klausel nicht zwangsläufig zu einer Verminderung der Abfragekosten führen.

A.14 Unzureichende Einschränkungen

Was Sie beim logischen Datenbankentwurf beachten sollten, damit Ihre Abfragen nicht unter einem unpassenden Datenbankdesign leiden müssen, wurde in diesem Buch nur am Rande erwähnt. Falls Sie sich hierfür interessieren, sollten Sie unbedingt einmal in [3] nachlesen. Sicher werden Sie sofort einsehen, dass es problematisch ist, wenn Ihr Datenbankdesign so ausgelegt ist, dass 70 Prozent der Abfragen einer OLTP-Anwendung zehn oder mehr Tabellen benötigen. Dies ist aber nur ein Aspekt des Datenbankdesigns. In kleinerem Maßstab gedacht, ist es auch sehr wichtig, die Definitionen von Tabellen und Einschränkungen so zu gestalten, dass der Optimierer bei der Erstellung von Ausführungsplänen hiervon profitieren kann.

Schauen Sie sich als Beispiel einmal die folgende Abfrage an:

```
select * from AdventureWorks2008.Person.Person
 where LastName is null
```

Die Abfrage soll alle Personen zurückgeben, deren Nachname nicht festgelegt wurde, also den Wert `NULL` aufweist. Abbildung A.3 zeigt den Ausführungsplan.

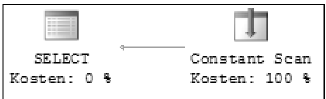


Abbildung A.3:
Ausführungsplan für eine Suche nach »LastName is NULL«

Die Abfragekosten sind nahezu 0, und zwar unabhängig davon, wie viele Zeilen die Tabelle `Person.Person` tatsächlich enthält. Die Ursache ist eine Einschränkung, mit der festgelegt wird, dass die Spalte `LastName` niemals `NULL` sein darf. Dies erkennt der Optimierer, sodass keinerlei Leseoperationen erforderlich sind.

Dies ist nur ein Beispiel. Auch andere Einschränkungen, wie etwa Fremdschlüssel, werden bei der Erstellung des Ausführungsplans hinzugezogen, um die Abfragekosten zu minimieren. Denken Sie also bitte daran, dass Sie Ihre Einschränkungen möglichst genau bzw. möglichst restriktiv festlegen sollten. Dies erleichtert dem Optimierer die Arbeit und senkt möglicherweise die Abfragekosten. Auch wenn das eigentliche Datenbankdesign bereits abgeschlossen ist, können Sie eventuell im Nachhinein noch weitere Einschränkungen hinzufügen.

B Literatur

[1] *Ben-Gan, Itztik, Kollar, Lubor, Sarka, Dejan:* Inside Microsoft SQL Server 2005 T-SQL Querying. 1. Auflage. Microsoft Press, 2006.

[2] *Schmeling, Holger:* Datenbankentwicklung mit dem Microsoft SQL Server 2005. 1. Auflage. Carl Hanser Verlag, 2007.

[3] *Davidson, Louis, Kline, Kevin, Windisch, Kurt:* Pro SQL Server 2005 Database Design and Optimization. 1. Auflage. Apress, 2006.

[4] *Celko, Joe:* SQL FOR SMARTIES: Advanced SQL Programming. Second Edition, Morgan Kaufmann Publishers, 2000.

Stichwortverzeichnis

A

Abfrage
- kompilieren 36
- logische Ausführungsreihenfolge 31
- parametrisierte 185, 212
- Parametrisierung 186
- problematische 266

Abfragekosten 35, 341
Abfragestatistik 282
Ablaufverfolgung 53
- Ereignisse 59
- Ereignisse filtern 61
- Ereignisspalten 59
- erstellen 55
- lange Ausführungsdauer 279
- Scan-Operationen 279
- serverseitige 63, 277
- Sort Warnings 279
- speichern 56
- Systemmonitor 78
- Vorlagen 57
- wieder einspielen 54

ADO.NET 272
Aggregat 252
Aktivitätsmonitor 50
- aktuell wertvolle Abfragen 52
- aktuelle Prozesse 51
- Datendatei E/A 52
- Ressourcenwartevorgänge 52
- Übersicht 51

Algebrizer 35, 182
ALL 250
ALTER INDEX 148
alter index 147
ALTER RESOURCE GOVERNOR 316
ANY 250
APP_NAME() 315
Arbeitsauslastungsgruppe
- default 317

Auflistelement 285, 291
Auflistsatz 285
- fehlende Indizes 290
- überflüssige Indizes 290

Ausführungsplan
- anzeigen 39
  - SET SHOWPLAN_ALL ON 42
  - SET SHOWPLAN_TEXT ON 42
  - SET SHOWPLAN_XML ON 42
  - SET STATISTICS PROFILE ON 42
  - SET STATISTICS XML ON 42
- fehlende Indizes 151
- geschätzte Werte 264
- geschätzter 39
- grafischer 40
- Kosten 41
- MissingIndexes 151
- Parametrisierung 213
- physikalischer 34
- tatsächliche Werte 264
- tatsächlicher 39
- trivialer 37
- Vereinfachung 37
- Wiederverwendbarkeit 212

Ausführungspläne 93
- Analyse 100
- Anzeige im Profiler 102
- Filter-Operatoren 101
- Index Spool 101
- Kompilierung 184
- nicht optimale 268
- Operatoren 94
- parallele Ausführung 101
- prozentuale Operator-Kosten 100
- Re-Kompilierung 184
- Scan-Operationen 101
- Schlüsselsuche (Clustered) 101
- Sort-Operatoren 101
- Table Spool 101
- Warnungen 102
- wiederverwenden 181, 186

auto_create_statistics 198
auto_update_statistics 200
auto_update_statistics_async 200
AVG_RANGE_ROWS 194

B

Benchmarking 47
Berichte 112, 276
- Abfragestatistik-Verlauf 121
- Anzahl von Fehlern 117
- Arbeitsspeichernutzung 115
- Datenträgerverwendung 117, 120
- Leistung 277
- Leistung – Bachausführungs-statistik 116
- Schemaänderungsverlauf 118
- Serveraktivität – Verlauf 118
- Serverdashboard 114

BETWEEN 252
Blockierungen 29

C

Cache
  Dirty Pages 115
  Free Pages 115
  Latched Pages 115
  Stolen Pages 115
checkpoint 208
Clustered Index Scan 95, 253
Clustered Index Seek 95
colmodctr 201
COMMIT 25
Controller 322
  Firmware 322
CREATE INDEX 136, 295
  DROP_EXISTING=ON 149
create index 254
create resource pool 314
CREATE STATISTICS 196, 198
create workload group 314
Cursor 339

D

DataBase Consistency Checker
  siehe DBCC
Database Tuning Advisor, siehe
  Datenbankoptimierungsratgeber
Dateigruppen 321
Datenauflister
  Ablaufverfolgungen 284
  DCEXEC.EXE 112
Datenauflistungen 105, 280, 289
  Abfrageaktivitäts-Auflistertyp 108
  Auflistsatz für Abfragestatistiken 110
  Auflistsatz für
    Datenträgerverwendung 109
  Auflistsatz für Serveraktivität 109
  konfigurieren 108
  Leistungsindikatoren-
    Auflistertyp 108
  SQL-Ablaufverfolgungs-
    Auflistertyp 108
  T-SQL-Abfrageauflistertyp 108
Datenbank
  Datendateien 23
  Protokolldateien 23
Datenbanken
  Autogrow 336
  Datendateien 320
  Indexdateien 321
  Protokolldateien 320
Datenbanklayout
  physikalisches 320
Datenbankoptimierungsratgeber 303
  Analyse 307
  Berichte 310
  Empfehlungen bewerten 309
  Optimierungsoptionen 307
  Tipps 309
Datencache 24, 199
Datenverteilung 190
DBCC 89
DBCC DROPCLEANBUFFERS 89
dbcc flushprocindb 185
DBCC FREEPROCCACHE 89
dbcc freeproccache 185
DBCC MEMORYSTATUS 213
dbcc show_statistics 205
dbo.syscollector_collection_items 286
declare cursor 340
dense_rank() 274
DISKPART.EXE 321
DISTINCT_RANGE_ROWS 194
DROP INDEX 141, 149
DTA, siehe
  Datenbankoptimierungsratgeber
DTA.EXE 310
Dynamische Verwaltungssichten 79, 265
  aktuelle Aktivität 81
  archivieren 273
  E/A-Vorgänge 81
  Leistungsindikatoren 84
  Wartezustände 82
Dynamisches SQL 338

E

E/A-Belastung 269, 273
E/A-Durchsatz 321
  messen 324
E/A-Last 275
E/A-Muster 322
E/A-Operationen 266, 283
  OLAP-Systeme 268
  OLTP-Systeme 267
  physische 319
  Rangliste 267
  Verlauf 274
E/A-System
  optimieren 319
  Planung 335
  Sättigung 329
  testen 319, 322
Einschränkungen 343
EQ_ROWS 193
EQUI JOIN 261
EXEC() 213

F

FILLFACTOR 139
Filter 97
Filterbedingungen 267
fn_trace_gettable() 66, 278
FORCESEEK 209

Foreign Keys, siehe Fremdschlüssel
Fremdschlüssel 261, 343
  MERGE JOIN 262
Fremdschlüsselbeziehung 262
  löschen 263

G

Gespeicherte Prozeduren
  bedingte Ausführung von
    Anweisungen 226
  Parameter 224
GROUPING SETS 299

H

HASH JOIN 245
Hash Match 96
Heap 125, 130
HOST_NAME() 315

I

IAM, siehe Index Allocation Map
IN 250
Index 125
  abdeckender 257, 300
  automatische Erstellung 137
  B* Baum 141
  Blattseiten 128
  CLUSTERED 136
  Defragmentierung 149
  eingeschlossene Spalten 132
  erstellen 135, 148
  fehlende 151
  fehlende 259, 267, 269
    Einschränkungen 157
  Fragmentierung 144
  Füllfaktor 138
  gefilterter 133
  gruppierter 127, 251
  INCLUDED 257
  Indexbaum 128
  löschen 141
  neu aufbauen 140
  nichtgruppierter 130 f.
  NONCLUSTERED 136
  Partitionierung mit 299
  PRIMARY KEY 137
  Reorganisation 144
  reorganisieren 147
  Selektivität 253
  Sortierung 253
  überflüssige 158
  UNIQUE 136, 338
  verwalten 143
  Wurzel 128
Index Allocation Map 125
Indexbaum 127
Indexentwurf 249
Indexhinweis 259
Indexstatistiken 192
Indexüberwachung 289
Indexverwendung
  Auflistelement 293
Indizierte Sichten 304
INNER JOIN 330–34
INSTEAD OF-Trigger 168
IS_SRVROLEMEMBER() 315

J

JOIN-Operatoren
  physikalische 242

K

Kardinalität 191
Kardinalitätsschätzung 189, 196, 228
  allgemeine Selektivität 231
Klassifizierungsfunktion 316
Klassifizierungsfunktion, siehe
  Ressourcenkontrolle
Kompilierung 71–72
Komprimierung 172
  Abfrageleistung 173
  Arten 172
  Seitenkomprimierung 172
  Speicherplatz berechnen 175
  Zeilenkomprimierung 172

L

Lazy Writer 25, 83
LEFT JOIN 168
Leistungsindikatoren 68, 70, 284
LUN 330

M

master.dbo.spt_monitor 87
MAX_CPU_PERCENT 313
MAX_MEMORY_PERCENT 313
MAXDOP 208
MERGE JOIN 244
Merge Join 96
MIN_CPU_PERCENT 313
MIN_MEMORY_PERCENT 313
msdb 273, 286

N

NESTED LOOPS 247
Nested Loops 96, 100
NEWSEQUENTIALID() 253
Nonclustered Index Scan 95
Nonclustered Index Seek 95

NOT EXISTS 250
NOT IN 250
NOT LIKE 250

O

Objekte
  temporäre 268
OLAP 71, 268, 302, 311, 326
OLTP 71, 267, 302, 311, 326
Operatoren
  Anzahl Ausführungen 99
  Eigenschaften 97
  erneute Bindungen 99
  geschätzte Anzahl Ausführungen 99
  geschätzte Anzahl Zeilen 99
  geschätzte CPU-Kosten 99
  geschätzte E/A-Kosten 98
  geschätzte Operatorkosten 99
  geschätzte Unterstrukturkosten 99
  geschätzte Zeilengröße 99
  logischer Vorgang 98
  physikalische 39
  physischer Vorgang 98
  tatsächliche Anzahl Zeilen 98
  Zurückspulvorgänge 99
Optimierer 35
OPTIMIZE FOR 233
OPTIMZE FOR
  UNKNOWN 234
OPTION (RECOMPILE) 255
OUTER JOIN 33–34

P

PAD_INDEX 139
Parallelisierung 299
Parameter Sniffing 223, 230, 233
  Probleme 223, 232
Parametrisierung
  einfache 220
  erzwungene 219–220
  FORCED 241
  Probleme 218
  SIMPLE 241
PARAMETRIZATION FORCED 220
Parser 34
Partitionierung 163
  horizontale 164, 300, 304
  Sichten 165
  vertikale 166, 303
Plan Guides, siehe Planhinweislisten
Plancache 181, 222, 281
  Ad-hoc-Abfragen 182
  Ausführungspläne 269
  gespeicherte Prozeduren 182
  parametrisierte Abfragen 183
  Trefferquote 186
Planhinweislisten 237
  OBJECT 238
  SQL 238
  TEMPLATE 238
Primärschlüssel 252
  gruppierter Index 335
  GUIDs 336
Profiler 277
Profiler siehe SQL Server Profiler

R

RAID 25, 321
RAID 1+0 335
RAID 5 335
RANGE_HI_KEY 193
RANGE_ROWS 193
Read Ahead 24, 73
RECOMPILE 235–236
  CREATE PROCEDURE 236
Re-Compilierung 71, 206
Ressourcen
  kontrollieren 311
Ressourcenkontrolle 311
  aktivieren 316
  Arbeitsauslastungsgruppe 313, 314
  einrichten 313
  Funktionsweise 312
  Klassifizierungsfunktion 315
  Priorisierung 312
Ressourcenpools 313
RowId Lookup 130, 191, 253

S

SARG 249, 257
  Funktionen 250
Scans 338
SCHEMABINDING 138
Schlüsselsuche 253
Schlüsselsuche (Clustered) 95
SELECT * 257
SET ANSI_NULL_DFLT_OFF 187
SET ANSI_NULL_DFLT_ON 187
SET ANSI_NULLS 187–188
SET ANSI_PADDING 187
SET ANSI_WARNINGS 187
SET ARITHABORT 187
SET CONCAT_NULL_YIELDS_
  NULL 187
SET DATEFIRST 187
SET DATEFORMAT 187
SET FORCEPLAN 187
SET LANGUAGE 187
SET NO_BROWSETABLE 187
SET NUMERIC_ROUNDABORT 187
SET QUOTED_IDENTIFIER 187
SET STATISTICS IO ON 49

SET STATISTICS TIME ON 49
SHRINK DATABASE 337
Sicht
indizierte 134, 137
signal_wait_time_ms 84
snapshots.io_virtual_file_stats 283
snapshots.notable_query_plan 283
snapshots.notable_query_text 283
snapshots.os_wait_stats 283
snapshots.performance_counters 284
snapshots.query_stats 283
snapshots.trace_data 286
SOME 250
Sort 97
sp_configure 69, 87, 139
sp_control_plan_guide 242
sp_create_plan_guide 238
sp_createstats 198
sp_estimate_data_compression_savings 175, 177
sp_executesql 214, 221–222, 232, 240
sp_get_query_template 217, 241, 279
sp_lock 87
sp_monitor 87
sp_spaceused 88
sp_syscollector_create_collection_item 109, 291
sp_syscollector_create_collection_set 291
sp_syscollector_update_collection_item 280
sp_trace_create 64
sp_trace_setevent 64
sp_trace_setstatus 64
sp_updatestats 201, 2050–206, 337
sp_who 88
sp_who2 88
Spindeln 320
Spool 97
SQL
dynamisches 232
SQL Server Betriebssystem, siehe SQLOS
SQL Server Profiler 53
Ablaufverfolgungsdateien 66
eigene Vorlagen erstellen 63
Fehlersuche 53
Laufzeitverhalten 54
SQL-Ablaufverfolgungs-Auflistertyp 284
SQLDiag
fortlaufender Modus 91
Vorlagen 91
SQLdiag 90
SQLIO 324, 327
Ergebnisse 330
Konfigurationsdatei 325
Parameter 324
SQLIOSIM 322
Konfigurationsdatei 322
Protokoll 323
SQLIOSIM.EXE 322
SQLOS 185
Statistik 192
abhängige 211
aktualisieren 197, 200, 337
Datenverteilung 195
Detaildaten 195
erstellen 197
fehlende 304
Histogramm 205, 210
nicht aktuelle 202
Probleme 207
Stichprobe 201
veraltete 203
Statistische Systemfunktionen 85
@@Connections 85
@@Cpu_Busy 85
@@Idle 85
@@IO_Busy 85
@@Pack_Received 85
@@Pack_Sent 85
@@Timeticks 85–86
@@Total_Read 85
@@Total_Write 85
arithmetischer Überlauf 86
stats_date() 202
Stoppuhr 48
Suchargumente, siehe SARG
SUSER_NAME() 315
sys.dm_db_index_physical_stats 128, 137, 140, 146, 149
detailed 146
sys.dm_db_index_usage_stats 159
sys.dm_db_missing_index_columns 156
sys.dm_db_missing_index_details 156
sys.dm_db_missing_index_group_stats 155
sys.dm_db_missing_index_groups 155
sys.dm_exec_cached_plans 103, 151, 153, 183
plan_handle 103
sys.dm_exec_plan_attributes 183, 188
sys.dm_exec_query_plan 103, 153, 183
sys.dm_exec_query_stats 104, 183, 188, 216, 270, 272
execution_count 188, 216
last_execution_time 275
query_hash 270
query_plan_hash 271
sys.dm_exec_requests 81, 109
sys.dm_exec_sessions 86, 109
sys.dm_exec_sql_text 81, 104, 183, 216, 272
sys.dm_io_virtual_file_stats 110, 272
sys.dm_os_performance_counters 84, 189
sys.dm_os_virtual_file_stats 81
sys.dm_os_wait_stats 83
sys.dm_os_waiting_tasks 109

sys.indexes 149, 159
sys.master_files 81
sys.partitions 110
sys.stats 202
sys.traces 63
syscollector_collector_types 290
sysschedules_localserver_view 290
Systemmonitor 67
  Ablaufverfolgungen 78
  Leistungsindikatoren 76
  Sammlungssatz 76
Systemprozeduren 87
Systemressourcenpool 316

T

TABLE HINT 240
Table Scan 95, 126, 254
tempdb 268
Transaktionsgröße 28
Transaktionsprotokoll 25
Transaktionsrate 28
T-SQL-Skript
  lokale Variablen 229

U

UNIQUEIDENTIFIER 252, 336
UPDATE STATISTICS 200

V

VDWH 106, 266, 280, 290
  Abfragestatistik 280
  Hochladen von Daten 294
  konfigurieren 107
  manuelle Abfragen 282
  Sicherheit 107
Verwaltungs-Data Warehouse siehe VDWH

W

wait_time_ms 84
Wartezustände 283
  LOGMGR_QUEUE 83
WHERE-Klausel 341
WorkTable 246
Write Ahead Log 25
WRITELOG 28–29

THE SIGN OF EXCELLENCE

Für Anwender, Administratoren und Entwickler. Dieses Buch beleuchtet die neuen SharePoint-Technologien im Detail, bietet Anleitung zur Migration und Entscheidungshilfe für den Einsatz des Microsoft Office SharePoint Servers bzw. der Windows SharePoint Services 3.0.

Torsten Mollien; Thomas Hauser; Dieter Scharnagl
ISBN 978-3-8273-2456-6
49.95 EUR [D]

www.addison-wesley.de

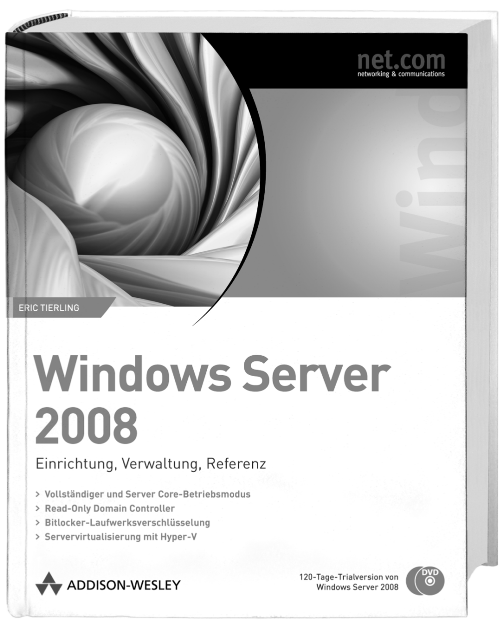
THE SIGN OF EXCELLENCE

Mit der neuen Version des Exchange Servers hat Microsoft einen gewaltigen Sprung nach vorn gemacht. 64-Bit, vereinfachtes Clustering, deutlich mehr Performance, verteilte Serverrollen, mehr Regeln, vollständige Verwaltung über die PowerShell - dies und noch viel mehr sind Themen, die Administratoren interessieren und die in diesem Buch behandelt werden. Berücksichtigt Service Pack 1.

Walter Steinsdorfer; Marc Jochems
ISBN 978-3-8273-2484-9
39.95 EUR [D]

www.addison-wesley.de

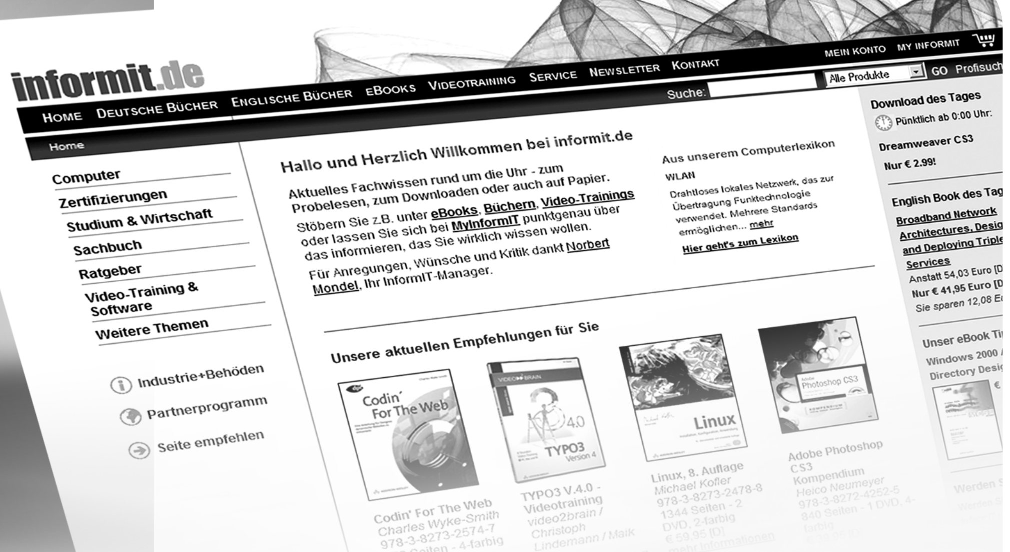
T H E   S I G N   O F   E X C E L L E N C E



Dieses Buch zu Windows Server 2008 knüpft an den Bestseller zu Windows Server 2003 an und widmet sich eingehend den Neuerungen der 2008er-Serverversion. Erstkonfiguration, Rollen und Features, überarbeiteter Server-Manager, Server Core-Installationsoption, BitLocker-Laufwerksverschlüsselung, Read-Only-Domänencontroller (RODC), Netzwerkzugriffsschutz (NAP), RemoteApp-Programme für die Terminaldienste, Failover-Clustering sowie die Servervirtualisierung mittels der neuen Technologie Hyper-V sind einige der Highlights, die im Buch beschrieben sind.

Eric Tierling
ISBN 978-3-8273-2637-9
59.95 EUR [D]

www.addison-wesley.de



informit.de, Partner von Addison-Wesley, bietet aktuelles Fachwissen rund um die Uhr.

www.informit.de

In Zusammenarbeit mit den Top-Autoren von Addison-Wesley, absoluten Spezialisten ihres Fachgebiets, bieten wir Ihnen ständig hochinteressante, brandaktuelle deutsch- und englischsprachige Bücher, Softwareprodukte, Video-Trainings sowie eBooks.

wenn Sie mehr wissen wollen ...

www.informit.de

Copyright

Daten, Texte, Design und Grafiken dieses eBooks, sowie die eventuell angebotenen eBook-Zusatzdaten sind urheberrechtlich geschützt. Dieses eBook stellen wir lediglich als persönliche Einzelplatz-Lizenz zur Verfügung!

Jede andere Verwendung dieses eBooks oder zugehöriger Materialien und Informationen, einschließlich

- der Reproduktion,
  der Weitergabe,
  des Weitervertriebs,
- der Platzierung im Internet,
  in Intranets,
  in Extranets,
- der Veränderung,
- des Weiterverkaufs

und der Veröffentlichung bedarf der schriftlichen Genehmigung des Verlags.

Insbesondere ist die Entfernung oder Änderung des vom Verlag vergebenen Passwortschutzes ausdrücklich untersagt!

Bei Fragen zu diesem Thema wenden Sie sich bitte an: info@pearson.de

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf unseren Websites ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.

Hinweis

Dieses und viele weitere eBooks können Sie rund um die Uhr und legal auf unserer Website

http://www.informit.de

herunter laden.