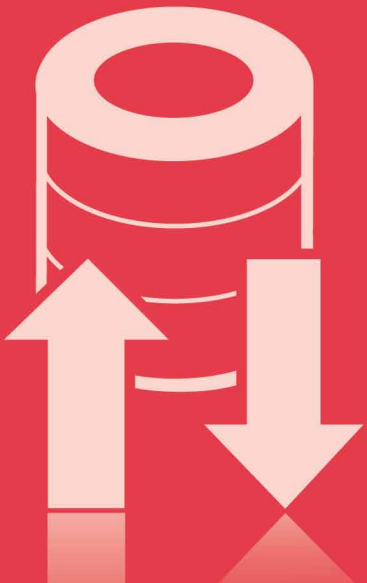
entwickler.press

SQL-Abfragen optimieren

Was Entwickler über Performance wissen müssen

Robert Panther



schnell + kompakt

Robert Panther

SQL-Abfragen optimieren

Was Entwickler über Performance wissen müssen

schnell+kompakt

entwickler.press

Robert Panther
SQL-Abfragen optimieren. Was Entwickler über Performance wissen müssen
schnell+kompakt
ISBN: 978-3-86802-310-7

ein Imprint der Software & Support Media GmbH

http://www.entwickler-press.de
http://www.software-support.biz

Ihr Kontakt zum Verlag und Lektorat: lektorat@entwickler-press.de

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

Lektorat: Theresa Vögle
Korrektorat: Frauke Pesch
Copy-Editor: Nicole Bechtel, Lisa Pychlau-Ezli
Satz: Dominique Kalbassi
Umschlaggestaltung und Titelbild: Maria Rudi

Inhaltsverzeichnis

Vorwort

Die Performance von Datenbanken ist oft ein kritischer Faktor für die effektive Nutzbarkeit von Businessanwendungen. Um eine gute Performance zu gewährleisten, müssen verschiedene Seiten der Datenbank betrachtet werden.

Während die Konfiguration und Überwachung der Datenbankserver idealerweise in den Händen von Datenbankadministratoren liegt, spielen auch die Datenbankzugriffe selbst eine entscheidende Rolle. Diese liegen normalerweise im Verantwortungsbereich der Entwickler (wobei ich hier Anwendungsentwickler und Datenbankentwickler der Einfachheit halber zusammenfasse). Aber auch von Datenbankadministratoren wird gelegentlich verlangt, SQL-Skripte zu schreiben, beispielsweise um Wartungsaufgaben zu implementieren, die dann regelmäßig automatisiert ausgeführt werden.

Dieses Buch erklärt in kompakter Form, wie man performante SQL-Abfragen formuliert. Der Inhalt ist in sechs Kapitel aufgeteilt:

- Die ersten Kapitel behandeln einige wichtige Grundlagen zu performancerelevanten Themen; Kapitel 1 startet mit einem Überblick über Indizes
- Kapitel 2 liefert Informationen zum besseren Verständnis der internen Verarbeitung von Abfragen
- Kapitel 3 schließt den Grundlagenteil mit einer Behandlung von Sperren und Transaktionen

- Ab Kapitel 4 geht es um die eigentliche Abfrageoptimierung, beginnend mit der Optimierung einzelner SQL-Abfragen
- Kapitel 5 behandelt Ansätze zur abfrageübergreifenden Optimierung, dabei werden auch verschiedene Varianten diskutiert, mit denen die Datenbanklogik von der Anwendung selbst auf den Server verlagert werden kann und wie sich dies auf die Performance auswirkt
- Kapitel 6 behandelt einige bewährte Problemstellungen aus der Praxis und zeigt, wie sich diese mit SQL möglichst performant lösen lassen

Der gesamte Text sowie insbesondere die Codebeispiele orientieren sich an Microsoft SQL Server ab Version 2005 bis hin zur gerade veröffentlichten 2014er-Version. Da es gerade bei Serveranwendungen üblich ist, dass auch ältere Versionen noch lange Zeit eingesetzt werden, wird stets darauf hingewiesen, wenn ein Optimierungsansatz erst mit einer neueren Version von SQL Server genutzt werden kann und – sofern möglich – auch eine Alternativlösung für ältere SQL-Server-Versionen aufgezeigt. Viele Optimierungsansätze sind sogar auf SQL-basierte Datenbankmanagementsysteme anderer Hersteller anwendbar. Somit erhält jeder, der selbst SQL-Abfragen schreibt – egal, ob Anwendungsentwickler, Datenbankentwickler oder DB-Administrator – wertvolle Hinweise für die Praxis.

Als Beispieldatenbank wird für die meisten Codebeispiele die Datenbank AdventureWorks in der 2012er-Variante genutzt, die bei Codeplex über den folgenden Link kostenfrei erhältlich ist: *http://msftdbprodsamples.codeplex.com/releases/view/93587*

Die 2014er-Variante der AdventureWorks-Beispieldatenbank war zur Fertigstellung dieses Texts leider noch nicht verfügbar. Falls Sie eine ältere Version als SQL Server 2012 einsetzen, sollten die Beispiele auch (notfalls mit geringen Anpassungen) auf den dazu

passenden Versionen der AdventureWorks-Datenbank nachvollziehbar sein.

Zum Verständnis des Texts und der Beispiele werden zumindest SQL-Grundkenntnisse vorausgesetzt, da eine ausführliche Darstellung der grundlegenden SQL-Befehle den Umfang dieses Buchs sprengen würde. Dazu gibt es für den Einstieg in die Programmierung mit SQL bereits genügend andere gute Bücher auf dem Markt.

Das Optimieren von SQL-Abfragen ist ein komplexes Thema. Zu vielen Problemstellungen gibt es keine eindeutig beste Lösung, sondern mehrere verschiedene Ansätze, mit denen das Problem angegangen werden kann. Wenn Sie also zum einen oder anderen beschriebenen Ansatz einen anderen alternativen Lösungsweg bevorzugen, oder auch einfach nur allgemeines Lob oder Kritik zu diesem Text loswerden möchten, würde ich mich über eine E-Mail an *sqlserver@rpanther.de* freuen.

Zum Abschluss dieses Vorworts möchte ich noch einen persönlichen Hinweis anbringen:

Ich möchte dieses Buch meinem Vater Eckart widmen, der im Jahr vor dem Erscheinen dieses Buches unerwartet früh verstorben ist. Er hat mich bereits im Jahr 1995 dazu motiviert, mich mit dem Thema SQL Performance Tuning zu beschäftigen und dies damals auch zum Thema meiner Diplomarbeit (Optimierung von Datenbankanwendungen) zu machen. Es freut mich im Nachhinein sehr, dass diese Idee seitdem in mir weiterlebt und auch nach fast zwei Jahrzehnten immer noch ein hochinteressantes und stets aktuelles Thema geblieben ist.

Robert Panther

Königstein im April 2014

Indizes

Bevor wir uns mit der Optimierung von SQL-Abfragen beschäftigen, ist es notwendig, ein paar Grundlagen zu kennen, damit man die weiter hinten im Buch beschriebenen Optimierungsansätze besser nachvollziehen kann.

Eines der wesentlichsten Elemente für einen performanten Datenbankzugriff sind Indizes. Dabei handelt es sich um zusätzliche Datenstrukturen, die den gezielten Zugriff auf bestimmte Datensätze beschleunigen sollen. Wie sich später zeigen wird, befasst sich auch ein guter Teil der Abfrageoptimierung mit der Problematik, ob existierende Indizes sinnvoll verwendet werden können. Ganz ohne Indizes kann keine größere relationale Datenbank performant laufen. Die Definition der richtigen Indizes erfordert jedoch etwas Hintergrundwissen, das auch für die Verwendung von Indizes in Abfragen hilfreich ist.

1.1 Grundlegende Funktionsweise von Indizes

Im einfachsten Fall kann man sich einen Index wie ein Stichwortverzeichnis in einem Buch vorstellen. Wenn man nun Informationen zu einem bestimmten Stichwort sucht, müsste man ohne das Stichwortverzeichnis das gesamte Buch lesen. Bei Datenbanktabellen entspricht dies einem so genannten *Table Scan*.

Ein Stichwortverzeichnis kann man – davon ausgehend, dass das Stichwortverzeichnis alle Begriffe umfasst (was bei einem Index der Fall ist) – nach dem Begriff durchsuchen (man nennt dies dann *Index Scan*) und dann gezielt auf die passenden Seiten des Buches zugreifen (entspricht einem *Row Lookup*).

Da das Stichwortverzeichnis sortiert ist, muss man nicht einmal das ganze Verzeichnis lesen, sondern kann darin gezielt nach den gewünschten Einträgen suchen, indem man in der Mitte des Stichwortverzeichnisses nachschaut und dann diejenige Hälfte davor bzw. danach weiter untersucht, in der das gesuchte Wort stehen muss. Mit dieser Hälfte verfährt man dann genauso wie im Schritt davor mit dem gesamten Verzeichnis (man nennt dies binäre Suche, da sich die zu durchsuchende Menge mit jedem Schritt halbiert). Somit sind insgesamt nur deutlich weniger Indexeinträge zu lesen, um den gesuchten Begriff zu finden. Bei einem Datenbankindex nennt man diesen Vorgang *Index Seek*, da der Index nicht komplett gelesen (gescannt), sondern stattdessen gezielt durchsucht wird.

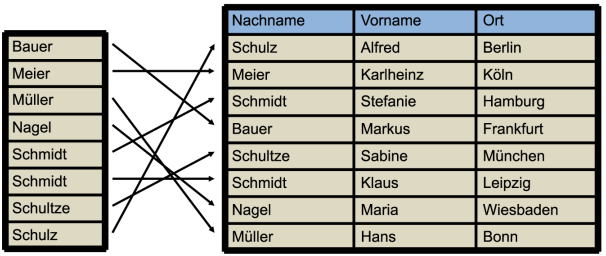


Abbildung 1.1: Vereinfachte Darstellung eines Index

Auch wenn dies in Büchern eher nicht der Fall sein wird, könnte man auch hier mehrere Indizes verwenden, die verschiedene Informationstypen speichern. So kann einer davon das eigentliche Stichwortverzeichnis sein und ein weiterer beispielsweise ein Abbildungs- oder ein Namensverzeichnis. Wenn nun nach einer bestimmten Person gesucht wird, kann dafür direkt das Namensverzeichnis angesprochen werden, was sicherlich deutlich kompakter und damit effektiver sein wird als das gesamte Stichwortverzeichnis.

Es gibt aber noch eine Sonderform von Indizes, nämlich die gruppierten (oder engl. clustered) Indizes. Bei ihnen wird kein zusätzliches Verzeichnis benötigt, sondern die Inhalte selbst sind in der richtigen Reihenfolge angeordnet. Dadurch ergibt sich allerdings auch, dass es pro Tabelle nur einen gruppierten Index geben kann. Die Entsprechung aus dem Verlagswesen ist ein Lexikon, in dem man ja auch mit relativ wenig Blättern die Informationen zu einem Begriff finden kann, ohne das ganze Buch lesen zu müssen (*Clustered Index Scan*). Stattdessen würde man ganz intuitiv irgendwo in der Mitte des Buches aufschlagen, um nachzuschauen, ob der gesuchte Begriff davor oder dahinter liegt und dann so fortfahren, wie bereits bei der binären Suche beschrieben. Der Unterschied zum nicht gruppierten Index liegt dann darin, dass – sobald man den gesuchten Begriff gefunden hat – dort kein Verweis auf dessen Beschreibung erfolgt, sondern diese direkt an der gefundenen Stelle steht. Der *Row Lookup* ist also nicht mehr erforderlich.

1.2 Realisierung von Indizes bei SQL Server

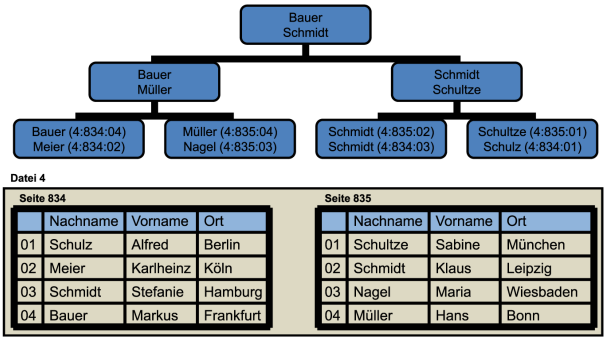
Bei SQL Server sind Indizes technisch etwas komplexer realisiert als es in der letzten Abbildung zu sehen war. Um dies korrekt darstellen zu können, muss man allerdings etwas weiter ausholen.

Wie bereits in den vorangegangenen Kapiteln erwähnt, speichert SQL Server seine Daten (das gilt in diesem Fall sowohl für Zeilen- als auch für Indexdaten) in Dateien. Diese Dateien sind in 8 KB große *Speicherseiten* unterteilt, in denen die eigentlichen Daten zu finden sind.

Im Fall einer Tabelle ohne gruppierten Index sind die Daten unsortiert in den Speicherseiten zu finden. Man nennt diese Anordnung daher auch *Heap* (= Haufen).

Bei einem nicht gruppierten Index sind die Indexeinträge in Form eines *B-Baums* gespeichert. Somit kommt man relativ schnell zu dem gesuchten Eintrag. Erst in der Blattebene des Baums ist dann ein Zeiger auf Datei-, Seiten- und Zeilennummer des Heaps gespeichert, sodass hierüber direkt auf die richtige *Datenseite* im Heap zugegriffen werden kann.

Bei einem gruppierten Index dagegen befinden sich die eigentlichen Daten direkt auf der Blattebene des B-Baums. Nicht gruppierte Indizes auf Tabellen mit gruppiertem Index verweisen dann auch nicht direkt auf die Speicherseite im Heap, sondern auf den gruppierten Index für die entsprechende Datenzeile. Dieses Verfahren bringt sowohl Vor- als auch Nachteile mit sich. Nachteilig ist, dass alle nicht gruppierten Indizes auf eine Tabelle neu angelegt werden müssen, wenn auf dieser Tabelle ein gruppierter Index neu angelegt oder gelöscht wird. Von Vorteil ist dafür, dass ein Reorganisieren des gruppierten Index kein Reorganisieren der nicht gruppierten Indizes nach sich zieht, da diese ja nicht direkt auf die Speicherseiten verweisen.



	Nachname	Vorname	Ort
01	Schulz	Alfred	Berlin
02	Meier	Karlheinz	Köln
03	Schmidt	Stefanie	Hamburg
04	Bauer	Markus	Frankfurt

	Nachname	Vorname	Ort
01	Schultze	Sabine	München
02	Schmidt	Klaus	Leipzig
03	Nagel	Maria	Wiesbaden
04	Müller	Hans	Bonn

Abbildung 1.2: Korrekte Darstellung eines Index (als B-Baum)

Sowohl bei gruppierten als auch bei nicht gruppierten Indizes befinden sich die Indexdaten ebenfalls auf 8 KB großen Speicherseiten. Kommt nun ein neuer Indexeintrag hinzu, so wird er auf der Speicherseite eingefügt, sofern der Platz noch ausreicht. Ist das nicht der Fall, so wird die Seite geteilt (*Page Split*), sodass daraus zwei maximal halbvolle Speicherseiten entstehen und der neue Indexeintrag problemlos eingefügt werden kann. Für die zusätzliche Speicherseite muss im Indexbaum auf der Ebene darüber natürlich ebenfalls ein neuer Eintrag erzeugt werden, sodass auch hier die Gefahr besteht, dass ein Page Split durchgeführt werden muss. Das funktioniert zwar soweit recht gut, hat aber drei entscheidende Nachteile:

- Der Page Split selbst kostet Zeit
- Durch Page Splits wird die B-Baum-Struktur immer unausgeglichener und der Index zunehmend fragmentiert
- Da die zusätzliche Indexseite an einer freien Stelle eingefügt wird, liegen die Indexseiten nicht mehr physikalisch in der

richtigen Reihenfolge vor; wenn in einer Abfrage ein größerer Bereich des Index sortiert gelesen werden muss, kann das nicht mehr kontinuierlich an einem Stück geschehen, sondern es werden zusätzliche Neupositionierungen der Festplattenköpfe nötig, was wiederum Zeit kostet

Um das Problem der Indexfragmentierung zu lösen, sollte man alle Indizes regelmäßig reorganisieren (REORG) oder besser noch neu erzeugen (REBUILD).

Beim SQL Server wird zum Reorganisieren die folgende Anweisung verwendet:

```
ALTER INDEX indexname
ON tabellenname REORGANIZE
```

Wird der Index komplett neu aufgebaut, kann dabei noch optional ein Füllfaktor angegeben werden, der definiert, bis zu welchem Prozentsatz die Speicherseiten des Index gefüllt werden (der frei gelassene Platz wird genutzt, damit der Index nicht so schnell neu fragmentiert wird):

```
ALTER INDEX indexname
ON tabellenname REBUILD
WITH (FILLFACTOR = fuellfaktor)
```

Die Wartung von Indizes (sowie auch der dazu gehörenden Statistiken) ist ein relativ komplexes Thema, das allerdings eher in den Aufgabenbereich von Datenbankadministratoren gehört.

Da es in diesem Text primär um die Optimierung von Abfragen geht, würde eine weitere Vertiefung dieses Themas an dieser Stelle zu weit führen. Weiterführende Informationen hierzu finden Sie in den Books online oder in meinem SQL-Server-Performance-Ratgeber, der ebenfalls bei entwickler.press erschienen ist.