FÜR ALLE GÄNGIGEN SQL-DATENBANKEN

SQL PERFORMANCE EXPLAINED

DEUTSCHE AUSGABE

ALLES, WAS ENTWICKLER ÜBER SQL-PERFORMANCE WISSEN MÜSSEN

MARKUS WINAND

SQL Performance Explained

Markus Winand

Medieninhaber (Verleger):
Markus Winand

Maderspergerstasse 1-3/9/11
1160 Wien
AUSTRIA
<office@winand.at>

Copyright © 2012 Markus Winand

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten.

Ungeachtet der Sorgfalt, die auf die Erstellung von Text, Abbildungen und Programmen verwendet wurde, können weder Verlag noch Autor für mögliche Fehler und deren Folgen eine juristische Verantwortung oder irgendeine Haftung übernehmen.

Die in diesem Werk wiedergegebenen Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. können auch ohne besondere Kennzeichnung Marken sein und als solche den gesetzlichen Bestimmungen unterliegen.

Dieses Buch gibt ausschließlich die Meinung des Autors wieder. Die genannten Datenbankhersteller haben das Werk weder finanziell unterstützt noch inhaltlich überprüft.

Druck:
digitaldruck.at Druck- und Handelsgesellschaft mbH – Leobersdorf

Umschlaggestaltung:
tomasio.design – Mag. Thomas Weninger – Wien

Titelfoto:
Brian Arnold – Turriff – UK

Lektorat:
textservice-stroblmayr.at – Mag. Dr. Tanja Stroblmayr BA

ISBN: 978-3-9503078-1-8

2012-05-01

SQL Performance Explained

Alles, was Entwickler über SQL Performance wissen müssen

Markus Winand
Wien

Inhalt

Vorwort ... vi

1. Anatomie eines Indexes ... 1
    - Die Index-Blätter ... 2
    - Der Suchbaum (B-Tree) ... 4
    - Langsame Indizes, Teil I ... 6

2. Die Where-Klausel ... 9
    - Der Gleichheitsoperator ... 9
        - Primärschlüssel ... 10
        - Zusammengesetzte Schlüssel ... 12
        - Langsame Indizes, Teil II ... 18
    - Funktionen ... 24
        - Groß- und Kleinschreibung mit UPPER ignorieren ... 24
        - Benutzer-definierte Funktionen ... 29
        - Über-Indizierung ... 31
    - Parametrisierte Abfragen ... 32
    - Nach Werte-Bereichen suchen ... 39
        - Größer, Kleiner und BETWEEN ... 39
        - LIKE-Filter indizieren ... 45
        - Indizes kombinieren ... 49
    - Partielle Indizes ... 51
    - NULL in der Oracle Datenbank ... 53
        - NULL Indizieren ... 54
        - NOT NULL-Constraints ... 56
        - Partielle Indizes emulieren ... 60
    - Verschleierte Bedingungen ... 62
        - Datums-Typen ... 62
        - Numerische Strings ... 68
        - Spalten zusammenfügen ... 70
        - Schlaue Logik ... 72
        - Mathematik ... 77

3. Performance und Skalierbarkeit 79
Auswirkungen des Datenvolumens 80
Auswirkungen der Systemlast 85
Antwortzeit und Durchsatz 87

4. Die Join-Operation 91
Nested Loops – verschachtelte Schleifen 92
Hash-Join 101
Sort-Merge 109

5. Daten-Cluster 111
Index-Filterprädikate gezielt einsetzen 112
Index-Only-Scan 116
Index-organisierte Tabellen 122

6. Sortieren und Gruppieren 129
Order By Indizieren 130
ASC, DESC und NULLS FIRST/LAST indizieren 134
GROUP BY indizieren 139

7. Teilergebnisse 143
Top-N-Zeilen abfragen 143
Durch Ergebnisse blättern 147
Mit Window-Funktionen blättern 156

8. Schreiboperationen 159
Insert 159
Delete 162
Update 163

A. Ausführungspläne 165
Oracle Datenbank 166
PostgreSQL 172
SQL Server 180
MySQL 188

Index 193

Vorwort

Entwickler müssen indizieren

Probleme mit der Performance von SQL-Abfragen sind so alt wie SQL selbst. Manche sagen sogar, dass diese Sprache von Natur aus langsam sei. Das mag zwar gestimmt haben, als SQL noch jung war, heute ist es aber nicht mehr so. Dennoch sind SQL-Performance-Probleme alltäglich – wie kommt das?

SQL ist die vielleicht erfolgreichste Programmiersprache der vierten Generation (4GL). Dabei ist es das Ziel, das „*Was*“ vom „*Wie*“ zu trennen: Eine SQL-Anweisung beschreibt, *was* benötigt wird, ohne Anweisungen zu geben, *wie* das Ergebnis erreicht wird. Das folgende Beispiel verdeutlicht das:

```
SELECT date_of_birth
  FROM employees
 WHERE last_name = 'WINAND'
```

Diese SQL-Abfrage kann fast als englischer Satz gelesen werden. Er beschreibt, welche Daten benötigt werden. SQL-Anweisungen kann man generell ohne jegliches Wissen über das Storage-System (Festplatten, Dateien, Blöcke) und die Interna der Datenbank verfassen. Eine SQL-Anweisung beinhaltet keine Instruktionen, welche Schritte auszuführen sind, welche Dateien geöffnet werden müssen, oder wie man den gesuchten Tabelleneintrag findet. Man kann viele Jahre mit SQL arbeiten, ohne zu wissen, wie die Datenbank zu den Ergebnissen kommt.

Obwohl die Trennung der Zuständigkeiten bei SQL ungewöhnlich gut funktioniert, ist die Abstraktion nicht perfekt. Das Problem zeigt sich bei der Performance: Der Autor einer SQL-Anweisung muss per Definition nicht wissen, *wie* die Anweisung ausgeführt wird. Folgerichtig kann er auch nicht für die Ausführungsgeschwindigkeit verantwortlich sein. Die Erfahrung lehrt uns jedoch das Gegenteil: Der Autor muss ein wenig über die Datenbank wissen, um effiziente SQL-Anweisungen zu schreiben.

Bei näherer Betrachtung stellt sich heraus, dass Entwickler nur wissen müssen, wie man einen Index richtig einsetzt. Denn zur richtigen Indizierung

ist die Kenntnis darüber, wie das Storage-System konfiguriert ist oder welche Hardware für die Datenbank benutzt wird, nicht wichtig. Zur erfolgreichen Indizierung muss man wissen, wie auf die Daten zugegriffen wird. Dieses Wissen über die Zugriffswege können sich Datenbankadministratoren (DBAs) oder externe Berater nur durch zeitaufwendiges Reverse-Engineering erarbeiten. In der Entwicklung ist es aber vorhanden.

Dieses Buch erklärt Entwicklern den Umgang mit Indizes – und nur das. Genau gesagt deckt das Buch nur den wichtigsten Index-Typ ab: den *B-Tree-Index*.

Der B-Tree-Index funktioniert in allen Datenbanken gleich. Das Buch verwendet zur besseren Lesbarkeit die Begriffe der Oracle® Datenbank, das Wissen kann dennoch auf andere Datenbanken angewandt werden. Randnotizen geben die entsprechenden Hinweise zu SQL Server®, MySQL und PostgreSQL.

Die Struktur des Buches entspricht den Bedürfnissen von Entwicklern: Die meisten Kapitel entsprechen einem bestimmten Teil einer SQL-Anweisung.

KAPITEL 1 — Anatomie eines Indexes

Das erste Kapitel ist das einzige, in dem keine SQL-Anweisungen vorkommen. Es beschreibt die Index-Struktur. Dieses Kapitel ist die Grundlage für alle folgenden und sollte keinesfalls übersprungen werden.

Obwohl das Kapitel nur acht Seiten hat, versteht man danach bereits, warum ein Index langsam sein kann.

KAPITEL 2 — Die Where-Klausel

Das Kapitel erklärte alles rund um die **where**-Klausel; es beginnt mit einfachen Abfragen auf einzelnen Spalten und endet bei komplexen Bereichsabfragen und Spezialfällen wie zum Beispiel `LIKE`.

Das Kapitel bildet den Hauptteil des Buches. Wer die hier beschriebenen beherrscht, wird bereits besseres SQL schreiben.

KAPITEL 3 — Performance und Skalierbarkeit

Performance ist von vielen Faktoren abhängig. Hier werden einige davon gezeigt und erklärt, warum horizontale Skalierung (mehr Hardware) nicht die beste Antwort auf langsame Abfragen ist.

KAPITEL 4 — Die Join-Operation

Zurück zu SQL: Wie benutzt man einen Index, um Joins zu optimieren?

KAPITEL 5 — DATEN-CLUSTER

Macht es einen Performanceunterschied, ob man nur eine Spalte oder alle Spalten selektiert? Die Antwort findet sich in diesem Kapitel – zusammen mit einem Trick für noch bessere Performance.

KAPITEL 6 — SORTIEREN UND GRUPPIEREN

Wie `order by` und `group by` von einem Index profitieren können.

KAPITEL 7 — TEILERGEBNISSE

Dieses Kapitel zeigt, wie man von einer Ausführung „am Fließband“ profitieren kann, wenn man nur die ersten Zeilen benötigt.

KAPITEL 8 — INSERT, DELETE UND UPDATE

Wie beeinflussen Indizes Schreibzugriffe? Indizes sind nicht kostenlos – verwende sie gezielt und sparsam!

ANHANG A — AUSFÜHRUNGSPLÄNE

Der Datenbank das „Wie“ entlocken: Wie wird die Anweisung ausgeführt?

Kapitel 1

Anatomie eines Indexes

„*Ein Index macht die Abfrage schnell*" ist die einfachste Beschreibung, die ich jemals für einen Datenbank-Index gehört habe. Obwohl sie seine wichtigste Eigenschaft gut erfasst, ist diese Erklärung für dieses Buch nicht ausreichend. Dieses Kapitel beschreibt die Struktur eines Indexes daher etwas genauer, ohne sich dabei in Details zu verlieren. Es gibt gerade genügend Einblick, um die Performanceaspekte im weiteren Verlauf des Buches zu verstehen.

Ein Index ist eine selbständige Datenstruktur, die mit dem **`create index`**-Kommando angelegt wird. Er benötigt seinen eigenen Speicherplatz und besteht hauptsächlich aus redundanten Informationen, die aus der Tabelle in den Index übernommen werden. Das Anlegen eines neuen Indexes ändert den Tabelleninhalt nicht. Es wird lediglich eine neue Datenstruktur angelegt, die auf die Tabellendaten verweist. Ein Datenbank-Index ähnelt also dem Index am Ende eines Buches: Er hat seinen eigenen Speicherplatz, besteht großteils aus Redundanzen und verweist auf die eigentliche Information, die an einer anderen Stelle gespeichert ist.

Clustered Indizes

SQL Server und MySQL (mit InnoDB) fassen den Begriff „Index" etwas weiter und bezeichnen Tabellen, die nur aus einer Indexstruktur bestehen, als „*Clustered Index*". In der Oracle Datenbank werden solche Tabellen als Index-Organized Table (IOT) bezeichnet.

Kapitel 5, „*Daten-Cluster*", beschreibt solche Tabellen genauer und erklärt die Vor- und Nachteile im Detail.

Die Suche in einem Datenbankindex entspricht am ehesten der Suche in einem gedruckten Telefonbuch. Das Grundkonzept ist, dass die Einträge in einer wohldefinierten Reihenfolge vorliegen. Wenn die Daten sortiert sind, kann man einzelne Einträge schnell finden, da sich die Position des Eintrags aus der Sortierreihenfolge ergibt.

Ein Datenbankindex ist dennoch etwas komplexer als ein Telefonbuch, da er ständig aktualisiert wird. Bei einem Telefonbuch wäre es unmöglich, jede Aktualisierung sofort durchzuführen. Schon alleine deshalb nicht, weil zwischen zwei bestehenden Einträgen kein Platz für einen neuen ist. Daher werden die gesammelten Änderungen bei einem Telefonbuch erst mit dem nächsten Druck berücksichtigt. Solange kann eine SQL-Datenbank aber nicht warten. Jede `insert`-, `delete`- und `update`-Anweisung muss sofort durchgeführt werden und dabei die Indexreihenfolge wahren.

Die Datenbank kombiniert zwei Datenstrukturen, um dieser Herausforderung zu begegnen: eine doppelt verkettete Liste und einen Suchbaum. Anhand dieser beiden Strukturen kann nahezu das gesamte Performanceverhalten einer Datenbank erklärt werden.

Die Index-Blätter

Die Hauptaufgabe eines Indexes ist es, die indizierten Daten in der Indexreihenfolge zu verwalten. Dazu genügt es nicht, die Einträge hintereinander abzuspeichern, da jede `insert`-Anweisung die dahinterliegenden Einträge verschieben müsste, um Platz für den neuen Eintrag zu schaffen. Das Verschieben großer Datenmengen ist jedoch sehr zeitaufwendig, sodass die `insert`-Anweisung sehr langsam werden würde. Die Lösung des Problems besteht darin, eine logische Reihenfolge herzustellen, die unabhängig von der physischen Reihenfolge im Speicher ist.

Die logische Reihenfolge der Indexeinträge wird durch eine doppelt verkettete Liste hergestellt. Ähnlich einer Kette hat jeder Listeneintrag (Knoten, engl. Node) eine Verbindung zu zwei Nachbargliedern. Neue Einträge werden zwischen zwei bestehenden eingefügt, indem die Verweise der beiden angrenzenden Einträge auf den neuen umgebogen werden. Der physische Speicherplatz des neuen Eintrags ist dabei irrelevant – die logische Reihenfolge wird nur durch die Verkettung hergestellt.

Da jeder Knoten dieser Struktur sowohl eine Referenz auf den vorangehenden Knoten als auch auf den folgenden Knoten hat, heißt diese Datenstruktur doppelt verkettete Liste. Die doppelte Verkettung ermöglicht es, den Index nach Bedarf vorwärts oder rückwärts zu lesen. Dennoch können neue Einträge in konstanter Zeit – das heißt unabhängig von der Listenlänge – eingefügt werden.

Doppelt verkettete Listen werden auch für Container und Collections in Programmiersprachen verwendet.

Programmiersprache	Name
Java	java.util.LinkedList
.NET Framework	System.Collections.Generic.LinkedList
C++	std::list

Datenbanken verwenden eine doppelt verkettete Liste, um die sogenannten Blattknoten (engl. Leaf-Nodes) miteinander zu verbinden. Dabei wird jeder Knoten in einem Datenblock (auch Seite oder Page) gespeichert. Alle Datenblöcke eines Indexes haben dieselbe Größe – meist einige Kilobyte. Die Datenbank nutzt den Platz optimal aus und speichert in jedem Block so viele Indexeinträge wie möglich. Daraus folgt, dass die Indexreihenfolge auf zwei Ebenen gewartet werden muss: einmal innerhalb der einzelnen Blöcke und dann die Blöcke untereinander durch die Verkettung.

Abbildung 1.1. Index Blattknoten und Tabellendaten

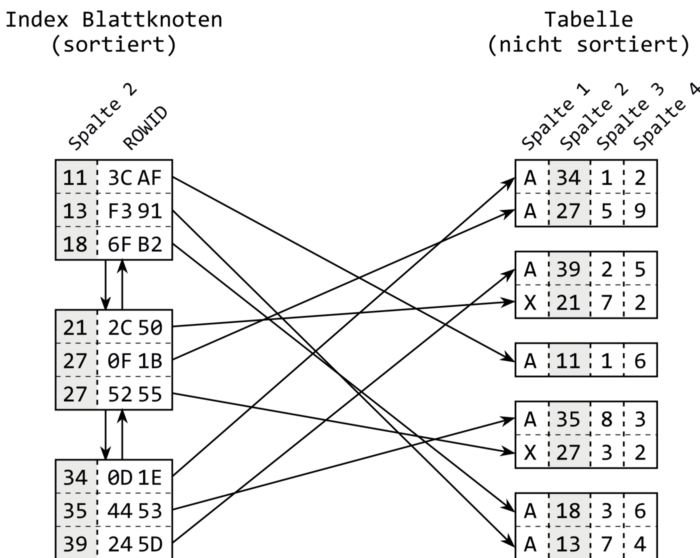


Abbildung 1.1 stellt einige Blattknoten und die entsprechenden Tabellendaten dar. Jeder Blattknoten-Eintrag besteht aus einer Kopie der indizierten Spalte (`Spalte 2`) und einem Verweis auf den entsprechenden Tabelleneintrag (`ROWID` oder `RID`). Im Gegensatz zu den Indexeinträgen sind die Tabelleneinträge in einer Heap-Struktur unsortiert gespeichert. Es gibt keinerlei Verbindung zwischen den Einträgen im selben Tabellenblock, noch sind die Tabellenblöcke untereinander verbunden.

Der Suchbaum (B-Tree)

Die physische Reihenfolge der Indexseiten ist willkürlich und lässt keinen Rückschluss auf die logische Position im Index zu. Die Suche darin ist also so, als wären die Seiten eines Telefonbuches durcheinander. Wenn man nach „Schmied“ sucht und das Telefonbuch bei „Müller“ öffnet, ist keineswegs sichergestellt, dass „Schmied“ weiter hinten kommt. Um die Indexeinträge dennoch schnell zu finden, wird eine zweite Struktur verwaltet: ein balancierter Suchbaum, kurz B-Tree genannt.

Abbildung 1.2. Struktur eines Suchbaumes

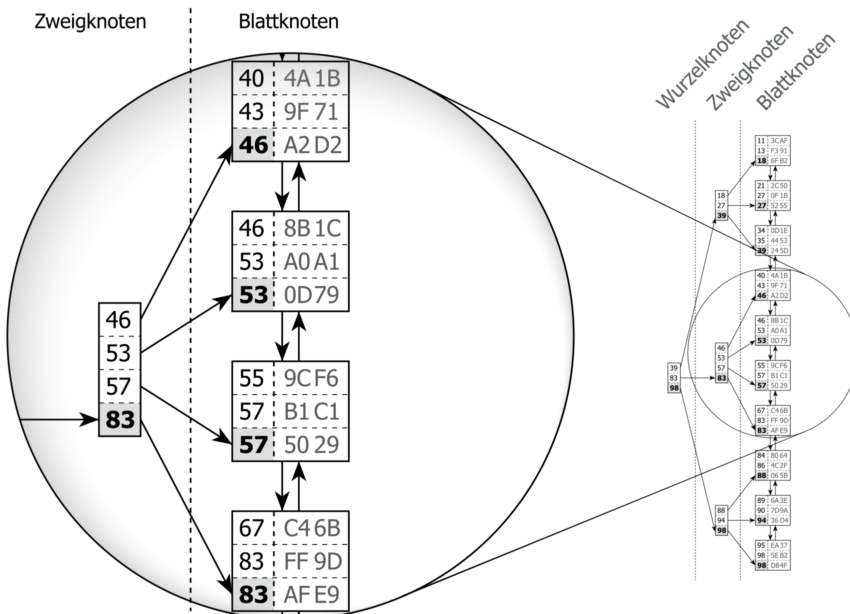


Abbildung 1.2 zeigt einen Index mit 30 Einträgen. Die Reihenfolge der Blattknoten wird durch die doppelte Verkettung hergestellt. Die Wurzel- und Zweigknoten (Root- und Branch-Nodes) dienen der Suche nach bestimmten Einträgen.

Die Abbildung hebt hervor, wie ein Zweigknoten auf die darunter liegenden Blattknoten verweist. Dabei entspricht der jeder Eintrag Zweigkoten dem größten Wert im Blattknoten. Da der größte Wert im ersten Blattknoten 46

ist, ist der erste Eintrag im Zweigknoten ebenfalls 46. Dasselbe gilt für alle weiteren Blattknoten, sodass der Zweigknoten letztendlich die Einträge 46, 53, 57 und 83 enthält. Nach diesem Schema wird eine Verzweigungsschicht aufgebaut, bis alle Blattknoten von einem Zweigknoten erfasst sind.

Die nächste Schicht ist ebenso aufgebaut, nur dass dabei auf die Zweigknoten verwiesen wird. Das Ganze wiederholt sich so lange, bis eine Schicht entsteht, die nur aus einem Knoten besteht – dem Wurzelknoten oder *Root-Node*. Das Besondere an dieser Baumstruktur ist, dass sie eine einheitliche Tiefe hat: Der Weg zwischen dem Wurzelknoten und den Blattknoten ist überall gleich lang.

Beachte

B-Tree steht für Balanced Tree – nicht Binary Tree.

Nachdem ein Index angelegt wurde, wird er von der Datenbank automatisch gepflegt. Der Index wird also mit jeder **`insert`**-, **`delete`**- und **`update`**-Anweisung aktualisiert. Die Datenbank hält den Index auch „im Gleichgewicht" (balanced). Der dadurch entstandene Wartungsaufwand kann durchaus signifikant werden. Kapitel 8, „*Schreiboperationen*", geht genauer darauf ein.

Abbildung 1.3. Durchwandern des Suchbaumes

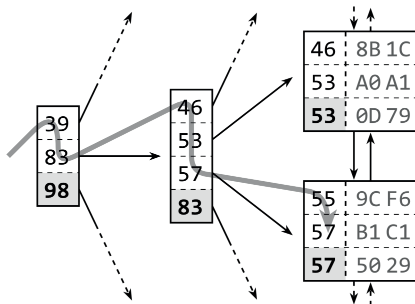


Abbildung 1.3 zeigt, wie der Suchbaum genutzt wird, um den Eintrag „57" zu finden. Die Suche beginnt links, beim Wurzelknoten. Die Einträge des Wurzelknotens werden der Reihe nach mit dem Suchbegriff „57" verglichen, bis ein Eintrag gefunden wird, der größer-gleich (>=) dem Suchbegriff ist. In der Abbildung trifft das auf den Eintrag „83" zu. Die Datenbank folgt dem Verweis auf den entsprechenden Zweigknoten und wiederholt die Prozedur so lange, bis sie einen Blattknoten erreicht.

Wichtig

Mit dem Indexbaum findet man einen Blattknoten sehr schnell.

Das Durchwandern des Suchbaumes ist sehr effektiv. So effektiv, dass ich diesen Vorgang als die *erste Macht der Indizierung* bezeichne. Das liegt einerseits daran, dass der Baum immer balanciert ist. Andererseits wächst der Suchbaum nur sehr langsam in die Tiefe. Ein Baum mit vier oder fünf Verzweigungen kann bereits einige Millionen Einträge abdecken. Eine Indextiefe von sechs wird nur ausgesprochen selten erreicht. Der Kasten „Logarithmische Skalierbarkeit" erklärt das Indexwachstum etwas genauer.

Langsame Indizes, Teil I

Obwohl das Durchwandern des Indexbaumes sehr effektiv ist, kann eine Indexsuche trotzdem langsam sein. Dieser Wiederspruch wird oft auf Mythos vom „defekten Index" zurückgeführt. Dieser Mythos verspricht eine Wunderheilung, wenn man den Index neu anlegt. Die wahre Ursache eines „langsamen Indexes" kann man anhand der Index- und Tabellenstruktur erklären.

Die erste Zutat zu einem langsamen Index ist die Blattknoten-Kette. Betrachten wir dazu nochmal die Suche nach dem Wert „57" in Abbildung 1.3. Offenbar gibt es für diese Suche zwei Treffer. Um genau zu sein, *mindestens* zwei Treffer. Denn der nächste Blattknoten könnte noch weitere Einträge für „57" enthalten. Die Datenbank muss also den nächsten Blattknoten laden und auf weitere Treffer prüfen. Bei einem Indexzugriff muss die Datenbank also nicht nur den Baum durchwandern, sondern auch die Blattknoten-Kette verfolgen.

Die zweite Zutat zu einem langsamen Index ist der Tabellenzugriff. Denn selbst ein einzelner Blattknoten kann viele Treffer liefern – oft hunderte. Die entsprechenden Tabellenzugriffe sind dann oft auf viele Datenblöcke verteilt (siehe Abbildung 1.1). Das bedeutet, dass für jeden Treffer noch ein Tabellenzugriff notwendig ist.

Insgesamt besteht ein Indexzugriff also aus drei Schritten: (1) das Durchwandern des Baumes; (2) das Verfolgen der Blattknoten-Kette; (3) der Tabellenzugriff. Aber nur beim ersten Schritt gibt es durch die Baumstruktur eine Obergrenze für die Anzahl der Lesezugriffe. Für die anderen Schritte müssen unter Umständen sehr viele Blöcke gelesen werden. Durch diese Schritte kann ein Indexzugriff langsam werden.

Logarithmische Skalierbarkeit

Mathematisch gesprochen ist der Logarithmus die Umkehroperation des Potenzierens. Er löst also die Gleichung $a = b^x$ nach dem Exponenten x auf [Wikipedia[1]].

Bei einem Suchbaum entspricht die Anzahl der Einträge pro Zweigknoten der Basis b und die Baumtiefe dem Exponenten x. Der Index aus Abbildung 1.2 kann offenbar bis zu vier Einträge pro Zweigknoten fassen und hat eine Tiefe von drei. Daraus resultiert, dass die Blattknoten bis zu 64 (4^3) Einträge fassen können. Wenn der Index um eine Verzweigungsschicht wächst, kann er bereits 256 (4^4) Einträge fassen. Jede neue Schicht vervierfacht also die Anzahl der möglichen Einträge. Der Logarithmus kehrt dieses Verhältnis um. Die Baumtiefe entspricht also $\log_4$(Indexeinträge).

Durch das logarithmische Wachstum kann der Beispielindex eine Million Einträge mit nur zehn Ebenen abdecken. Ein echter Index ist aber noch effektiver. Denn der Hauptfaktor für die Baumtiefe ist die Anzahl der Einträge pro Zweigknoten – mathematisch gesprochen, die Basis des Logarithmus. Je größer die Basis, desto flacher bleibt der Baum.

Baumtiefe	**Indexeinträge**
3	64
4	256
5	1.024
6	4.096
7	16.384
8	65.536
9	262.144
10	1.048.576

Datenbanken nutzen diesen Effekt aus und speichern so viele Einträge wie möglich in den einzelnen Knoten – oft hundert oder mehr. Das bedeutet, dass jede weitere Verzweigungsschicht die Indexkapazität verhundertfacht.

[1] http://de.wikipedia.org/wiki/Logarithmus

Der Mythos des defekten Indexes ist der Irrglaube, dass ein Indexzugriff nur den Baum durchwandern muss und daher immer schnell ist. Daraus folgt dann, dass ein langsamer Indexzugriff von einem „defekten“ oder „entarteten“ Index verursacht werden muss. Die Wahrheit ist, dass man Datenbanken sogar fragen kann, welche Schritte sie für einen Indexzugriff durchführen. Die Oracle Datenbank ist in diesem Zusammenhang besonders mitteilsam und hat für einen Indexzugriff gleich drei Operationen:

INDEX UNIQUE SCAN
: Der `INDEX UNQIUE SCAN` durchwandert nur den Index-Baum. Die Oracle Datenbank verwendet diese Operation, wenn ein Constraint sicherstellt, dass das maximal ein Eintrag dem Abfragekriterium entspricht.

INDEX RANGE SCAN
: Der `INDEX RANGE SCAN` durchwandert den Index-Baum und folgt anschließend der Blattknoten-Kette, um alle Treffer zu finden. Diese Operation wird verwendet, wenn mehrere Treffer möglich sind.

TABLE ACCESS BY INDEX ROWID
: Lädt eine Zeile anhand einer `ROWID` aus der Tabelle. Diese Operation wird für jeden Treffer der vorangegangen Indexoperation durchgeführt.

Wichtig ist, dass ein `INDEX RANGE SCAN` einen sehr großen Teil des Indexes lesen kann. Wenn dann noch ein Tabellenzugriff für jede Zeile durchgeführt wird, kann eine Abfrage trotz Index sehr langsam sein.

Kapitel 2

Die Where-Klausel

Im vorigen Kapitel haben wir die Struktur eines Indexes untersucht und die Ursachen für langsame Indizes kennengelernt. Im nächsten Schritt betrachten wir, wie man diese Ursachen in SQL-Anweisungen erkennen und vermeiden kann. Dazu beginnen wir mit der `where`-Klausel.

Die `where`-Klausel drückt die Suchbedingungen einer SQL-Anweisung aus und fällt damit in die Kernkompetenz eines Indexes: Daten schnell zu finden. Doch obwohl die `where`-Klausel großen Einfluss auf die Performance hat, wird sie oft sorglos formuliert, sodass ein großer Indexbereich durchsucht werden muss. Das heißt, eine schlecht formulierte `where`-Klausel liefert die erste Zutat für eine langsame Abfrage.

Dieses Kapitel erklärt, wie sich die verschiedenen Operatoren auf die Indexnutzung auswirken und wie man sicherstellt, dass ein Index von möglichst vielen Abfragen genutzt werden kann. Zum Abschluss werden häufige Fehler vorgestellt und Alternativen aufgezeigt.

Der Gleichheitsoperator

Der Gleichheitsoperator ist sowohl der einfachste als auch der meistgenutzte Operator. Dennoch kann man bei der Indizierung Fehler machen, die sich maßgeblich auf die Performance auswirken. Das trifft vor allem auf `where`-Klauseln zu, die mehrere Bedingungen kombinieren.

Dieser Abschnitt erklärt, wie man kontrolliert, ob ein Index benutzt wird, und wie man zusammengesetzte Indizes für kombinierte Bedingungen verwendet. Dazu wird eine langsame Abfrage analysiert und gezeigt, wie sich die beiden Zutaten aus Kapitel 1 in der Praxis zeigen.

Primärschlüssel

Zunächst betrachten wir die einfachste, aber dennoch häufigste **where**-Klausel: die Suche mit dem Primärschlüssel. Dafür legen wir zuerst die Tabelle EMPLOYEES an, die für sämtliche Beispiele dieses Kapitels verwendet wird:

```
CREATE TABLE employees (
   employee_id   NUMBER         NOT NULL,
   first_name    VARCHAR2(1000) NOT NULL,
   last_name     VARCHAR2(1000) NOT NULL,
   date_of_birth DATE           NOT NULL,
   phone_number  VARCHAR2(1000) NOT NULL,
   CONSTRAINT employees_pk PRIMARY KEY (employee_id)
);
```

Mit der Definition des Primärschlüssels wird automatisch ein Index auf EMPLOYEE_ID angelegt. Daher ist in diesem Fall keine **create index**-Anweisung notwendig.

Die folgende Abfrage sucht einen Angestellten mit dem Primärschlüssel und zeigt seinen Vor- und Nachnamen an:

```
SELECT first_name, last_name
  FROM employees
 WHERE employee_id = 123
```

Der Primärschlüssel garantiert, dass die Abfrage maximal einen Eintrag liefert. Die Datenbank muss also nur den Indexbaum durchwandern, nicht aber der Blattknoten-Kette folgen. Zur Kontrolle kann man den sogenannten *Ausführungsplan* heranziehen.

```
---------------------------------------------------------------
|Id |Operation                   | Name          | Rows | Cost |
---------------------------------------------------------------
| 0 |SELECT STATEMENT            |               |    1 |    2 |
| 1 | TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |    1 |    2 |
|*2 |  INDEX UNIQUE SCAN         | EMPLOYEES_PK  |    1 |    1 |
---------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPLOYEE_ID"=123)
```

Der Ausführungsplan der Oracle Datenbank zeigt einen `INDEX UNIQUE SCAN` – jene Operation, die nur den Indexbaum durchwandert. Dadurch wird die logarithmische Skalierung des Indexbaumes voll ausgenutzt, sodass der Eintrag nahezu unabhängig von der Tabellengröße sehr schnell gefunden wird.

Tipp

Der Ausführungsplan (auch *Explain-Plan* oder *Query-Plan* genannt) zeigt die Ausführungsschritte der Datenbank an. Anhang A zeigt, wie man Ausführungspläne in verschiedenen Datenbanken abfragt und interpretiert.

Nach der Indexsuche muss die Datenbank noch einen weiteren Schritt durchführen, um den Vor- und Nachnamen aus der Tabelle zu laden. Die `TABLE ACCESS BY INDEX ROWID` Operation stellt diesen Tabellenzugriff dar. Obwohl dieser Zugriff – wie in „Langsame Indizes, Teil I" beschrieben – zum Performanceproblem werden kann, ist das bei einem `INDEX UNIQUE SCAN` nicht der Fall. Diese Operation kann nämlich nur einen Eintrag liefern und daher auch nur einen Tabellenzugriff auslösen. Bei einem `INDEX UNIQUE SCAN` können die Zutaten zu einer langsamen Abfrage also nicht auftreten.

Primärschlüssel ohne Unique Index

Ein Primärschlüssel erfordert nicht zwangsläufig einen Unique Index – man kann dafür auch einen Non-Unqiue Index verwenden. Das führt dazu, dass die Oracle Datenbank einen `INDEX RANGE SCAN` verwendet. Dennoch ist die Eindeutigkeit des Primärschlüssels gewährleistet, sodass die Indexsuche maximal einen Eintrag liefert.

Einer der Gründe einen Non-Unique Index für einen Primärschlüssel zu verwenden, sind verzögerte Constraints (deferrable constraints). Im Gegensatz zu normalen Constraints werden verzögerte erst geprüft, wenn die Transaktion mittels `commit` abgeschlossen wird. Verzögerte Constraints werden benötigt, um Tabellen mit zirkulären Abhängigkeiten zu befüllen.

Zusammengesetzte Schlüssel

Obwohl die Datenbank den Primärschlüssel automatisch indiziert, kann man den Index dennoch verfeinern, wenn der Primärschlüssel aus mehreren Spalten besteht. Dafür wird ein zusammengesetzter Index verwendet. Das ist *ein Index*, der alle Spalten des Primärschlüssels enthält. Die Reihenfolge der Spalten im Index hat jedoch einen großen Einfluss darauf, welche Abfragen ihn nutzen können. Die Spalten-Reihenfolge muss also sehr sorgfältig gewählt werden.

Zur Demonstration stellen wir uns eine Betriebsübernahme vor. Dabei werden die Mitarbeiter anderer Unternehmen in die EMPLOYEES-Tabelle übernommen, sodass diese insgesamt auf das Zehnfache anwächst. Das einzige Problem ist, dass die EMPLOYEE_ID nicht mehr eindeutig ist. Daher erweitern wir den Primärschlüssel um eine weitere Spalte: die Zweigestellen-Nummer. Der neue Primärschlüssel besteht also aus zwei Spalten: der EMPLOYEE_ID wie gehabt, und der SUBSIDIARY_ID, um die Eindeutigkeit wiederherzustellen.

Der Index für den Primärschlüssel ist also wie folgt definiert:

```
CREATE UNIQUE INDEX employee_pk
    ON employees (employee_id, subsidiary_id);
```

Eine Abfrage nach einem bestimmten Mitarbeiter muss den vollständigen Primärschlüssel verwenden, also auch die SUBSIDIARY_ID:

```
SELECT first_name, last_name
  FROM employees
 WHERE employee_id   = 123
   AND subsidiary_id = 30;

---------------------------------------------------------------
|Id |Operation                   | Name         | Rows | Cost |
---------------------------------------------------------------
| 0 |SELECT STATEMENT            |              |    1 |    2 |
| 1 | TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |    1 |    2 |
|*2 |  INDEX UNIQUE SCAN         | EMPLOYEES_PK |    1 |    1 |
---------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPLOYEE_ID"=123 AND "SUBSIDIARY_ID"=30)
```

Wenn der Zugriff mit dem vollständigen Primärschlüssel erfolgt, kann die Datenbank einen `INDEX UNIQUE SCAN` durchführen – egal wie viele Spalten der Index hat. Aber was geschieht, wenn die Abfrage nur eine der beiden Spalten verwendet? Wenn zum Beispiel alle Mitarbeiter einer Zweigestelle gesucht werden:

```
SELECT first_name, last_name
  FROM employees
 WHERE subsidiary_id = 20;

----------------------------------------------------
| Id | Operation         | Name      | Rows | Cost |
----------------------------------------------------
|  0 | SELECT STATEMENT  |           |  106 |  478 |
|* 1 |  TABLE ACCESS FULL| EMPLOYEES |  106 |  478 |
----------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("SUBSIDIARY_ID"=20)
```

Der Ausführungsplan zeigt, dass der Index nicht benutzt wird. Stattdessen wird eine `TABLE ACCESS FULL`-Operation durchgeführt. Dabei liest die Datenbank die ganze Tabelle und vergleicht jede Zeile mit der **`where`**-Klausel. Die Ausführungszeit steigt proportional mit der Tabellengröße. Wenn die Tabelle auf das Zehnfache wächst, dauert der `TABLE ACCESS FULL` auch zehn Mal so lange. Das Tückische an dieser Operation ist, dass sie in kleinen Entwicklungsumgebungen oft schnell genug ist, in Produktion aber zu Problemen führt.

Full-Table-Scan

Die Operation `TABLE ACCESS FULL`, auch *Full-Table-Scan* genannt, kann in manchen Fällen dennoch die beste Zugriffsmethode sein. Das trifft vor allem auf Abfragen zu, die einen großen Teil der Tabelle liefern.

Das liegt einerseits daran, dass der Indexzugriff selbst einen Mehraufwand darstellt, der bei einem `TABLE ACCESS FULL` nicht auftritt. Andererseits müssen die Datenblöcke bei einer Indexsuche einzeln gelesen werden. Schließlich erfährt die Datenbank erst durch das Lesen eines Blockes, welcher als Nächstes benötigt wird. Im Gegensatz dazu braucht ein `TABLE ACCESS FULL` ohnehin alle Blöcke und kann daher größere Tabellenbereiche auf einmal lesen (*multi block read*). Dabei werden zwar mehr Daten gelesen, es erfolgen aber nicht unbedingt mehr Zugriffe.

Die Datenbank verwendet den Index nicht, da man die einzelnen Spalten eines zusammengesetzten Indexes nicht willkürlich verwenden kann. Zum Verständnis betrachten wir den Aufbau eines zusammengesetzten Indexes genauer.

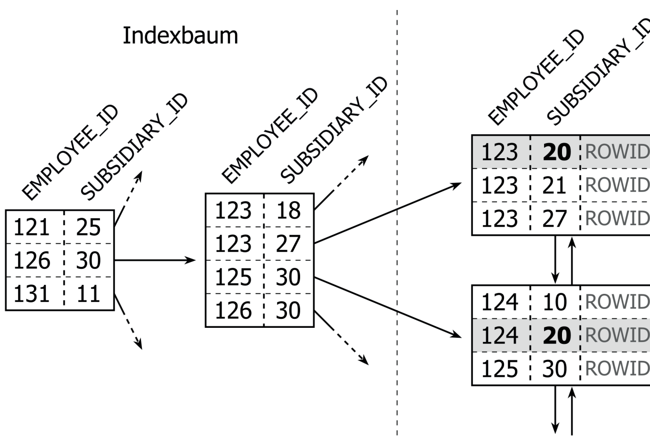
Auch ein zusammengesetzter Index ist ein B-Tree-Index, der die indizierten Daten als sortierte Liste verwaltet. Bei der Sortierung werden die Spalten entsprechend ihrer Position im Index berücksichtigt. Die erste Spalte wird also als vorrangiges Sortierkriterium herangezogen. Die zweite Spalte bestimmt die Reihenfolge nur dann, wenn zwei Einträge denselben Wert in der ersten Spalte haben. Und so fort.

Wichtig

Ein zusammengesetzter Index ist *ein Index über mehrere Spalten*.

Die Sortierung eines zweispaltigen Indexes erfolgt also wie bei einem Telefonbuch, das zuerst nach Nachname, dann nach Vorname sortiert ist. Daraus folgt, dass ein zweispaltiger Index nicht zur Suche auf der zweiten Spalte alleine verwendet werden kann. Das wäre, als würde man in einem Telefonbuch mit einem Vornamen suchen.

Abbildung 2.1. Zusammengesetzter Index



Der Indexausschnitt in Abbildung 2.1 zeigt, dass die Einträge mit `SUBSIDIARY_ID = 20` nicht an einer zentralen Stelle liegen. Es ist auch zu sehen, dass `SUBSIDIARY_ID = 20` nicht im Indexbaum aufscheint, obwohl einsprechende Einträge in den Blattknoten sind. Der Indexbaum ist für diese Abfrage völlig nutzlos.

Tipp

Die Visualisierung eines Indexes hilft zu verstehen, welche Abfragen unterstützt werden. Dazu kann man sich die Indexreihenfolge mit einer Abfrage ansehen (SQL:2008 Syntax, siehe Seite 144 für proprietäre Lösungen mit LIMIT, TOP oder ROWNUM):

```
SELECT <INDEX SPALTENLISTE>
  FROM <TABELLE>
 ORDER BY <INDEX SPALTENLISTE>
 FETCH FIRST 100 ROWS ONLY;
```

Wenn man die Indexdefinition und den Tabellennamen einsetzt, erhält man einen Auszug aus dem Index. Dabei sollte man sich fragen, ob die gesuchten Daten an einer zentralen Stelle zusammengefasst sind. Andernfalls kann man den Indexbaum nicht nutzen, um diese Stelle zu finden.

Um die Abfrage dennoch zu beschleunigen, kann man natürlich einen zweiten Index auf SUBSIDIARY_ID anlegen. Es gibt jedoch eine bessere Alternative – zumindest wenn man davon ausgeht, dass die Suche nach einer EMPLOYEE_ID alleine keinen Sinn ergibt.

Dafür macht man sich zunutze, dass die erste Spalte eines Indexes immer für Suchen herangezogen werden kann. In einem Telefonbuch kann man ja auch ohne einen Vornamen zu kennen, nach einem Nachnamen suchen. Der Trick besteht also darin, die Spaltenreihenfolge des Indexes umzudrehen, sodass SUBSIDIARY_ID an erster Position steht:

```
CREATE UNIQUE INDEX EMPLOYEES_PK
    ON EMPLOYEES (SUBSIDIARY_ID, EMPLOYEE_ID);
```

Beide Spalten zusammen sind nach wie vor eindeutig, sodass der Zugriff über den vollständigen Primärschlüssel weiterhin als INDEX UNIQUE SCAN erfolgt. Die Index-Sortierung hat sich jedoch grundlegend geändert. Der neue Index ist vorrangig nach der SUBSIDIARY_ID sortiert. Das heißt, dass alle Mitarbeiter einer Zweigstelle unmittelbar hintereinander im Index stehen. Dadurch kann man den Indexbaum auch nutzen, um sie zu finden.

Wichtig

Die wichtigste Überlegung beim Erstellen eines zusammengesetzten Indexes ist, wie man die Spaltenreihenfolge wählt, damit er möglichst viele SQL-Anweisungen unterstützten kann.

Der Ausführungsplan zeigt, dass der „umgedrehte" Index benutzt wird. Da die Spalte `SUBSIDIARY_ID` alleine nicht eindeutig ist, muss die Datenbank die Blattknoten verfolgen, um alle Einträge zu finden. Es findet also ein `INDEX RANGE SCAN` statt:

```
--------------------------------------------------------------
|Id |Operation                   | Name        | Rows | Cost |
--------------------------------------------------------------
| 0 |SELECT STATEMENT            |             |  106 |   75 |
| 1 | TABLE ACCESS BY INDEX ROWID| EMPLOYEES   |  106 |   75 |
|*2 |  INDEX RANGE SCAN          | EMPLOYEE_PK |  106 |    2 |
--------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("SUBSIDIARY_ID"=20)
```

Ein zusammengesetzter Index kann benutzt werden, wenn mit den führenden Spalten gesucht wird. Das heißt, ein Index mit drei Spalten kann für Suchen mit der ersten Spalte alleine, mit den ersten beiden oder mit allen drei Spalten gemeinsam verwendet werden.

Obwohl die Lösung mit einem zweiten Index auch sehr gute **`select`**-Performance liefert, ist die Variante mit einem Index vorzuziehen. Dadurch wird nicht nur Speicherplatz gespart, sondern auch der Wartungs-Aufwand, den jeder Index nach sich zieht. Je weniger Indizes eine Tabelle hat, desto besser ist die **`insert`**-, **`delete`**- und **`update`**-Performance.

Zur optimalen Indexdefinition muss man also nicht nur wissen, wie ein Index funktioniert, man muss auch wissen, wie die Anwendung auf die Daten zugreift. Konkret heißt das, dass man wissen muss, welche Spaltenkombinationen in der **`where`**-Klausel vorkommen.

Für externe Berater kann es daher sehr schwierig sein, einen optimalen Index anzulegen, da die Übersicht über die Zugriffspfade fehlt. Daher wird oft nur eine einzelne SQL-Abfrage berücksichtigt. Den Mehrwert, den ein besser definierter Index für andere Abfragen bringen könnte, wird nicht ausgeschöpft. Datenbankadministratoren sind in einer ähnlichen Situation.

Sie kennen die Daten zwar besser, einen umfassenden Überblick über die Zugriffspfade der Applikation haben sie aber auch nicht.

Die einzige Stelle, an der sowohl das Applikationswissen als auch Datenbankwissen vorhanden ist, ist die Entwicklung. Entwickler haben ein Gefühl für die Daten und kennen die Zugriffspfade. Damit können sie ohne großen Aufwand richtig Indizieren, um einen optimalen Nutzen erzielen.

Langsame Indizes, Teil II

Der vorherige Abschnitt zeigte, wie man durch eine veränderte Spaltenreihenfolge einen zusätzlichen Nutzen aus einem Index ziehen kann. Dabei wurden aber nur zwei Abfragen berücksichtigt. Wenn man einen bestehenden Index ändert, sind jedoch alle Zugriffe auf die Tabelle betroffen. Dieser Abschnitt erklärt, wie die Datenbank einen Index auswählt, und zeigt die möglichen Nebenwirkungen einer Indexänderung.

Der neue `EMPLOYEE_PK`-Index verbessert die Performance für Abfragen, die nur mit der `SUBSIDIARY_ID` suchen. Der Index wird dadurch aber für alle Abfragen nutzbar, die einen `SUBSIDIARY_ID`-Filter benutzen – unabhängig davon, ob sie noch andere Suchkriterien verwenden. Das sind also auch Abfragen, die zuvor einen anderen Index, mit einem anderen Teil der **`where`**-Klausel, verwendet haben. In diesem Fall, wenn es mehrere Zugriffswege für eine Abfrage gibt, ist es die Aufgabe des Optimizers, den besten Zugriffsweg auszuwählen.

Der Optimizer

Der *Optimizer*, auch *Query Planner* genannt, ist jene Datenbank-Komponente, die den Ausführungsplan für eine SQL-Anweisung erstellt. Dieser Vorgang wird auch „compiling“ oder „parsing“ genannt. Man unterscheidet zwei Optimizer-Varianten.

Ein kostenbasierter Optimizer (CBO) erstellt eine Vielzahl möglicher Ausführungspläne und bewertet jeden mit einem Cost-Wert. Die Berechnung des Cost-Wertes erfolgt anhand der durchgeführten Operationen und des Mengengerüstes (Zeilenzahl). Der Cost-Wert dient dann als Benchmark, um den besten Ausführungsplan auszuwählen.

Ein regelbasierter Optimizer (RBO) erstellt den Ausführungsplan nach einem fest programmierten Regelwerk und ist daher weniger flexibel. Regelbasierte Optimizer sind heutzutage kaum mehr in Verwendung.

Die Änderung des Indexes kann auch unangenehme Nebenwirkungen haben. In unserem Beispiel trifft es die interne Telefonbuch-Applikation, die seit der Betriebszusammenlegung sehr langsam geworden ist. Bei der ersten Analyse wurde die folgende SQL-Anweisung als Ursache identifiziert.

```
SELECT first_name, last_name, subsidiary_id, phone_number
  FROM employees
 WHERE last_name  = 'WINAND'
   AND subsidiary_id = 30;
```

Dabei wird der folgende Ausführungsplan verwendet:

Beispiel 2.1. Ausführungsplan mit geändertem Index

```
---------------------------------------------------------------
|Id |Operation                     | Name         | Rows | Cost |
---------------------------------------------------------------
| 0 |SELECT STATEMENT              |              |    1 |   30 |
|*1 | TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |    1 |   30 |
|*2 |  INDEX RANGE SCAN           | EMPLOYEES_PK |   40 |    2 |
---------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
  1 - filter("LAST_NAME"='WINAND')
  2 - access("SUBSIDIARY_ID"=30)
```

Der Ausführungsplan hat einen Cost-Wert von 30 und verwendet einen Index. Auf den ersten Blick sieht er also gut aus. Es ist aber auffällig, dass der geänderte Index verwendet wird. Der Verdacht liegt also nahe, dass die Indexänderung die Ursache des Problems ist. Wenn man dann noch bedenkt, dass der Index mit der alten Definition (`EMPLOYEE_ID`, `SUBSIDIARY_ID`) nicht für die Abfrage benutzt werden konnte, kann kein Zweifel mehr bestehen: Die Indexänderung machte die Abfrage langsam.

Zur weiteren Analyse wäre der Vergleich mit dem Ausführungsplan mit dem alten Index interessant. Dafür könnte man den Index natürlich zurückbauen. Viele Datenbanken bieten aber einfachere Möglichkeiten an, die Indexnutzung für eine Abfrage zu unterbinden. Das folgende Beispiel verwendet dafür einen Optimizer-Hint der Oracle Datenbank.

```
SELECT /*+ NO_INDEX(EMPLOYEES EMPLOYEE_PK) */
       first_name, last_name, subsidiary_id, phone_number
  FROM employees
 WHERE last_name  = 'WINAND'
   AND subsidiary_id = 30;
```

Der Ausführungsplan, der vermutlich vor der Indexänderung verwendet wurde, verwendet überhaupt keinen Index:

```
---------------------------------------------------
| Id | Operation         | Name      | Rows | Cost |
---------------------------------------------------
|  0 | SELECT STATEMENT  |           |    1 |  477 |
|* 1 |  TABLE ACCESS FULL| EMPLOYEES |    1 |  477 |
---------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("LAST_NAME"='WINAND' AND "SUBSIDIARY_ID"=30)
```

Obwohl der Full-Table-Scan die gesamte Tabelle lesen und verarbeiten muss, ist er in diesem Fall offenbar schneller als der Indexzugriff. Das ist besonders ungewöhnlich, da die Abfrage nur einen Treffer liefert. Gerade das Finden weniger Einträge sollte mit einem Index deutlich schneller sein. Ist es in diesem Fall aber nicht – der Index ist scheinbar langsam.

In solchen Fällen ist es am besten, den problematischen Ausführungsplan Schritt für Schritt durchzudenken. Die Ausführung beginnt mit einem `INDEX RANGE SCAN` auf dem `EMPLOYEE_PK`-Index. Da die Spalte `LAST_NAME` nicht im Index ist, kann der `INDEX RANGE SCAN` nur die `SUBSIDIARY_ID` berücksichtigen. Die Oracle Datenbank zeigt das im „Predicate Information" Bereich des Ausführungsplanes an. Dort sieht man also, welche Bedingungen bei welcher Operation aufgelöst werden.

Tipp

Anhang A, „*Ausführungspläne*", erklärt, wie der „Predicate Information" Bereich bei anderen Datenbanken angezeigt wird.

Der `INDEX RANGE SCAN` mit der Id 2 (Beispiel 2.1 auf Seite 19) verwendet nur die Bedingung `SUBSIDIARY_ID=30`. Das bedeutet, dass der Indexbaum benutzt wird, um den ersten Eintrag mit `SUBSIDIARY_ID` 30 zu finden. Danach wird die Blattknoten-Liste verfolgt, um alle Einträge mit dieser `SUBSIDIARY_ID` zu finden. Das Ergebnis des Indexzugriffs ist also eine Liste von `ROWID`s, die der `SUBSIDIARY_ID`-Bedingung entsprechen. Das können, abhängig von der Größe der Zweigstelle, einige wenige oder aber tausende sein.

Der nächste Schritt ist der `TABLE ACCESS BY INDEX ROWID`. Er verwendet die `ROWID`s aus dem vorherigen Schritt, um die vollständigen Zeilen – alle

Spalten – aus der Tabelle zu laden. Erst wenn die `LAST_NAME`-Spalte verfügbar ist, kann der übrige Teil der **`where`**-Klausel geprüft werden. Das heißt, dass die Datenbank alle Zeilen für `SUBSIDIARY_ID` 30 laden muss, bevor sie die `LAST_NAME`-Bedingung anwenden kann.

Die Antwortzeit dieser Abfrage hängt also nicht von der Größe des Ergebnisses, sondern von der Anzahl der Mitarbeiter in der jeweiligen Zweigstelle ab. Wenn es nur wenige sind, wird der Indexzugriff schneller sein. Wenn es aber viele sind, kann ein `FULL TABLE SCAN` vorteilhaft sein, da er große Tabellenbereiche auf einmal lesen kann (siehe „Full-Table-Scan" auf Seite 13).

Die Abfrage ist langsam, weil der Indexzugriff sehr viele Zeilen liefert – alle Mitarbeiter der ursprünglichen Firma – und die Datenbank jede Zeile einzeln aus der Tabelle laden muss. Die beiden Zutaten für einen langsamen Index treffen also zusammen: Es wird ein großer Indexbereich gelesen und anschließend ein Tabellenzugriff für jede Zeile durchgeführt.

Die Wahl des besten Ausführungsplanes hängt also auch von Tabellendaten ab. Daher verwendet der Optimizer Statistiken über den Tabelleninhalt. In diesem Fall ein Histogramm über die Verteilung der Mitarbeiter auf die Zweigstellen. Dadurch kann der Optimizer abschätzen, wie viele Zeilen der Indexzugriff auf eine bestimmte `SUBSIDIARY_ID` liefert und diese Zahl bei der Berechnung des Cost-Wertes berücksichtigen.

Statistiken

Ein kostenbasierter Optimizer nutzt Statistiken auf Tabellen-, Index- und Spalten-Ebene. Die meisten Statistiken werden auf Spaltenebene geführt: die Anzahl der unterschiedlichen Werte, der kleinste und größte Wert, die Zahl der `NULL`-Einträge und das Histogramm, das die Werteverteilung anzeigt. Auf Tabellenebene sind es vor allem die Tabellengröße (in Zeilen und Blöcken) und die durchschnittliche Zeilenlänge.

Die wichtigsten Index-Statistiken sind die Tiefe des Baumes, die Zahl der Blattknoten, die Zahl der unterschiedlichen Einträge und der Clustering-Faktor (siehe Kapitel 5, „*Daten-Cluster*").

Der Optimizer verwendet diese Werte, um die Selektivität der einzelnen Bedingungen abzuschätzen.

Wenn es keine Statistiken gibt – weil sie zum Beispiel für diese Demonstration absichtlich gelöscht wurden – verwendet der Optimizer Standardwerte. Die Oracle Datenbank geht dann von einem kleinen Index und einer mittleren Selektivität aus. Daraus ergibt sich die Schätzung, dass der INDEX RANGE SCAN 40 Zeilen liefert. Dieser Wert wird in der Rows-Spalte des Ausführungsplanes angezeigt (noch immer Beispiel 2.1 auf Seite 19). Offenbar eine grobe Fehleinschätzung, da in der betroffenen Zweigstelle 1000 Mitarbeiter beschäftigt sind.

Stellt man dem Optimizer akkurate Statistiken zur Verfügung, kann er eine bessere Abschätzung durchführen. Im folgenden Ausführungsplan geht er daher davon aus, dass der INDEX RANGE SCAN 1000 Einträge liefert. Dementsprechend hoch wird auch der Cost-Wert für den Tabellenzugriff angesetzt.

```
---------------------------------------------------------------
|Id |Operation                   | Name         | Rows | Cost |
---------------------------------------------------------------
| 0 |SELECT STATEMENT            |              |    1 |  680 |
|*1 | TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |    1 |  680 |
|*2 |  INDEX RANGE SCAN          | EMPLOYEES_PK | 1000 |    4 |
---------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
  1 - filter("LAST_NAME"='WINAND')
  2 - access("SUBSIDIARY_ID"=30)
```

Der Cost-Wert ist mit 680 sogar höher als beim Ausführungsplan ohne Index (477, siehe Seite 20). Daher bevorzugt der Optimizer in dieser Situation automatisch den Ausführungsplan mit der TABLE ACCESS FULL Operation.

Dieses Beispiel eines „langsamen Indexes“ soll aber nicht darüber hinwegtäuschen, dass ein wohl definierter Index die beste Lösung ist. Ein Index auf LAST_NAME unterstützt die Abfrage zum Beispiel sehr gut:

```
CREATE INDEX emp_name ON employees (last_name);
```

Mit diesem Index ermittelt der Optimizer den Cost-Wert 3.

Beispiel 2.2. Ausführungsplan mit dezidiertem Index

```
-------------------------------------------------------------
| Id | Operation                    | Name      | Rows | Cost |
-------------------------------------------------------------
|  0 | SELECT STATEMENT             |           |    1 |    3 |
|* 1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES |    1 |    3 |
|* 2 |   INDEX RANGE SCAN          | EMP_NAME  |    1 |    1 |
-------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("SUBSIDIARY_ID"=30)
   2 - access("LAST_NAME"='WINAND')
```

Der Indexzugriff mit LAST_NAME liefert, nach Schätzung des Optimizers, nur eine Zeile. Daher muss auch nur eine Zeile aus der Tabelle geladen werden. Das ist definitiv schneller als die ganze Tabelle zu lesen. Ein wohldefinierter Index liefert also bessere Performance als der ursprüngliche Full-Table-Scan.

Der Unterschied der beiden Ausführungspläne in Beispiel 2.1 (Seite 19) und Beispiel 2.2 ist nur sehr gering. Die Datenbank führt die gleichen Operationen durch und der Optimizer hat vergleichbare Cost-Werte ermittelt. Dennoch ist die Performance des zweiten Ausführungsplanes deutlich besser. Die Effizienz eines Indexzugriffes kann stark variieren – insbesondere wenn ein Tabellenzugriff folgt. Nur weil ein Index benutzt wird, heißt das nicht zwangsläufig, dass die Abfrage optimal ausgeführt wird.