Otto-von-Guericke Universität Magdeburg

Fakultät für Informatik

Institut für Technische und Betriebliche

Informationssysteme

Diplomarbeit

Last-balancierte Indexstrukturen

Verfasser :
Sebastian Mundt
29. Februar 2008

Betreuer :
Prof. Dr. rer. nat. habil. Gunter Saake
Dr.-Ing. Eike Schallehn

Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D-39016 Magdeburg
Germany

Mundt, Sebastian
Last-balancierte Indexstrukturen
Diplomarbeit, Otto-von-Guericke-Universität
Magdeburg, 2008.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

BCNF	**B**oyce-**C**odd-**N**ormal**f**orm
DBMS	**D**aten**b**ank**m**anagement**s**ystem
DBS	**D**aten**b**anks**y**stem
DB	**D**aten**b**ank
DML	**D**ata **M**anipulation **L**anguage
DMS	**D**atabase **M**anaged **S**torage
ERP	**E**nterprise **R**esource **P**lanning
FCL	**F**eedback **C**ontrol **L**oop
ISP	**I**ndex **S**election **P**roblem
LOB	**L**arge **B**inary **O**bject
MAPE	**M**onitor **A**nalyze **P**lan **E**xecute
MPL	**M**ulti**p**rogramming **L**evel
OLAP	**O**n**l**ine-**A**nalytical-**P**rocessing
ORDBS	**O**bjekt**r**elationale **D**aten**b**anks**y**steme
QUIET	**Qu**ery-driven **I**nd**e**x **T**uning
SQL	**S**tructured **Q**uery **L**anguage
TA	**T**rans**a**ktion

1. Einleitung

1.1. Aufgabenstellung

In Datenbankmanagementsystemen (DBMS) sind B+-Bäume und verschiedene Derivate heutzutage Standard für Sekundärindexe. Dabei werden über den zu indexierenden Wertebereich einer oder mehrerer Indexspalten alle Tupel einer Relation effizient zugreifbar gemacht. B+-Bäume sind vollständig, vollständig höhenbalanciert und nach der Werteverteilung angeordnet. Für das Self-Tuning kann es jedoch sinnvoll sein, Indexe nur teilweise für besonders häufig zugegriffene Datenbereiche zu erzeugen, da sie dann weniger Ressourcen verbrauchen, jedoch an den Blättern nur dünnbesetzt wären. An einem Indexpfad häufig benutzter Daten sollten nur wenige Seiten bzw. Einträge hängen, auf seltener benutzte Daten kann auch durch größere Scans über viele Seiten zugegriffen werden. Eine solche Indexstruktur muss allerdings dynamisch sein bzgl.

- Einfügen und Löschen von Daten (wie bei normalen Indexen)
- Veränderung der Zugriffsverteilung (neu)
- Veränderung der zugewiesenen Ressourcen (Schrumpfen oder Wachsen)

In der vorliegenden Diplomarbeit werden Ansätze für ein geeignetes Konzept ähnlich einem B-Baum (vor allem feste Knotengröße mit vielen Einträgen) entwickelt, das auch implementiert und experimentell evaluiert werden kann.

1.2. Motivation

Ein Datenbankindex ist eine Zugriffsstruktur, die eine Suche über bestimmte Felder einer Tabelle beschleunigen soll. Ein solcher Index erfüllt nur dann seinen Zweck, wenn das DBMS die erfragten Daten mit Index effizienter liefern kann als ohne Index.
Mit einem Index wird versucht die Objektmenge, auf der nach einem Ergebnis gesucht wird, zu verringern, um gleichermaßen auch die Antwortzeiten zu reduzieren. Auf der anderen Seite erfordert der Unterhalt eines Index Speicherplatz und Rechenzeit. Nur wenn die Kosten in einem positiven Verhältnis zum Nutzen stehen, sollte ein Index beibehalten werden.
In der Praxis gibt es verschiedenste Anwendungen, in denen die Zugriffshäufigkeit auf bestimmte Attributausprägungen in einer Tabelle asymmetrisch verteilt ist. Werden z. B. Kundenstammdaten, wie zum Beispiel Adresse und Name des Ansprechpartners, in einer Tabelle abgelegt, so ist zu erwarten, dass Kunden, die oft bestellen, eine höhere

Zugriffshäufigkeit aufweisen, als solche, die bisher nur ein einziges Mal geordert haben. Würde die Anzahl der Zugriffe registriert werden und anhand dessen über die Besetzung eines Indexknotens entschieden, könnte der Speicherplatzverbrauch für weniger häufig aufgesuchte Knoten eingeschränkt werden. Hingegen bestünde für häufiger besuchte Knoten mit einer solchen Gewichtung die Möglichkeit die Tupel auf mehr, aber dafür dünnbesetzte Knoten zu verteilen.

1.3. Zielsetzung

Das Ziel des wissenschaftlichen Teils dieser Diplomarbeit ist die Entwicklung eines Konzepts, das die Vor- und Nachteile eines klassischen B-Baums mit den aktuellen Arbeiten auf dem Gebiet des zugriffs- und speicherplatzoptimierten Self-Tunings verbindet. Um eine Entscheidung über den Zeitpunkt und die Art der Optimierung zu treffen, ist die Zugriffshäufigkeit bestimmter Knoten eines Index zu registrieren und zu speichern. Davon ausgehend ist die Besetzung der einzelnen Indexknoten so einzustellen, dass häufiger benutzte Daten effizient, d. h. mit einem geringeren Verbrauch an Ressourcen, erreichbar sind. In diesem Zusammenhang wird ein Kostenmodell aufgestellt, mit dessen Hilfe der Nutzen der Maßnahmen bewertet werden kann.
Ein solcher Index wird bestehende Indexstrukturen nicht ersetzen, sondern ergänzen. Für ganz bestimmte Anwendungen wird ein Index als nützlich erachtet, der so wenig Speicherplatz wie nötig verbraucht, aber den Zugriff auf die relevanten Tupel beschleunigt. Es wird gezeigt, dass sich in Kombination mit anderen Techniken wertvolle Anwendungsmöglichkeiten für eine Last-balancierten B-Baum in DBMS ergeben.

1.4. Gliederung

Zunächst werden einige der verbreiteten DBMS vorgestellt und die jeweiligen Verfahren zur Speicherverwaltung beleuchtet.
Im anschließenden Kapitel finden sich die gängigsten Zugriffsstrukturen, die in DBMS verwendet werden, damit die Grundbegriffe für die spätere Nennung definiert sind.
Die bekanntesten Self-Tuning-Ansätze werden im darauf folgenden Kapitel vorgestellt und auf deren Eignung zur Lösung des Problems der last-balancierten Indexstrukturen hin überprüft.

Das folgende Kapitel befasst sich mit dem Ansatz eines zugriffsbalancierten Binärbaums und entwickelt diesen weiter in einen zugriffsbalancierten B-Baum und erläutert dessen Grundlagen und Mechanismen.
Nachfolgend wird die Implementierung und Evaluation des gewählten Ansatzes dargelegt und hiernach zusammengefasst.

2. Datenbankmanagementsysteme

Datenbanksysteme sind heutzutage allgegenwärtig. Die bereits 1948 vom US-amerikanischen Wissenschaftler Norbert Wiener vorhergesagte Informationsgesellschaft produziert täglich eine ungeheure Datenmenge, die gespeichert werden muss und auf die zugegriffen werden soll. Datenbanksysteme wurden entwickelt, um dauerhaft, effizient und konsistent die Speicherung und Wiedergewinnung von großen Datenmengen abzuwickeln.
Der Teil eines Datenbanksystems, der für die Verwaltung der Daten zuständig ist, wird als DBMS bezeichnet. Den anderen Teil eines Datenbanksystems bilden die Daten in der eigentlichen Datenbank (DB) selbst.

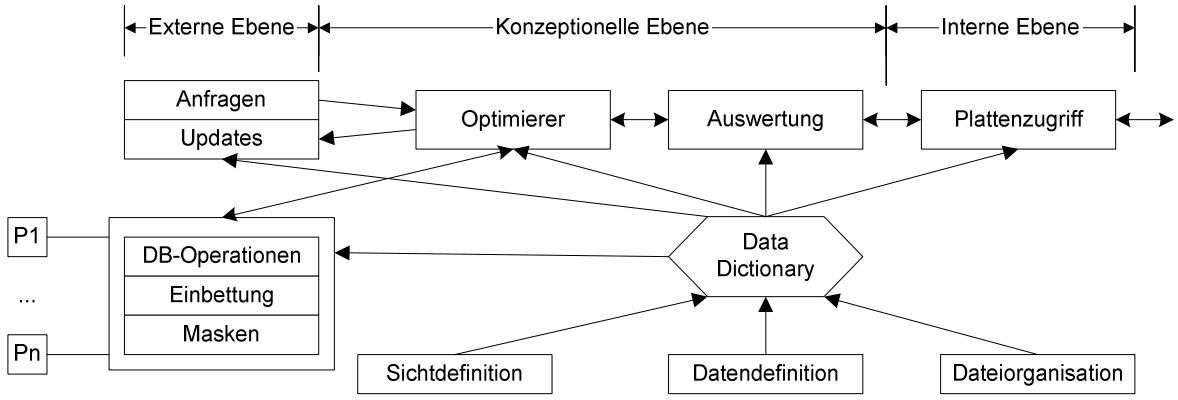


Abbildung 1: Vereinfachte Architektur eines DBMS nach [SH99]

Die verschiedenen Sichten auf ein DBMS werden in der **Abbildung 1** dargestellt. Die externe Ebene beschreibt die Sicht einer Anwendung auf die in der DB abgelegten Daten. Die logische und einheitliche Gesamtsicht, wird in der konzeptionellen Ebene hergestellt, während Fragen der tatsächlichen Datenspeicherung in der Internen Ebene behandelt werden. Über eine interaktive Schnittstelle oder über die Einbettung einer Datenbank(abfrage)sprache (z. B. *Embedded SQL, vgl. 2.1.5*) in einem Anwendungsprogramm kann auf den Datenbestand zugegriffen werden. Die Zugriffsanfragen werden optimiert, d. h. das DBMS versucht mit Hilfe des Data Dictionary die beste Zugriffsstruktur für den Plattenzugriff auszuwählen, der dann realisiert wird. Die Ergebnisse der Anfrage oder Änderungen werden ausgewertet und an die Schnittstelle zurückgegeben. [SH99]

2.1. Datenbankkonzepte

2.1.1. Relationale Datenbanksysteme

Das relationale Datenmodell basiert auf mengentheoretischen Relationen, die als Teilmengen des kartesischen Produkts zweier oder mehrerer Mengen definiert sind. Die anschauliche Vorstellung ist die Speicherung der Daten in 2-dimensionalen Tabellen, wobei die Spalten den Attributen der Relation entsprechen. Die Zeilen enthalten die eigentlichen Daten in Form eines Tupels[1]. Die Daten können nur in atomarer[2] Form in den einzelnen Zellen der Tabelle gespeichert werden. – dies entspricht der so genannten 1. Normalform des relationalen Modells.

Von den relationalen DMBS (RDBMS) wird das relationale Modell der Datenhaltung implementiert. Die heutigen Aufgabenstellungen werfen für klassische RDBMS jedoch eine Reihe von Problemen auf. So erfolgt die Speicherung eines Objektes im Sinne der objektorientierten Programmierung typischerweise in mehreren Relationen. Dabei werden die die Daten segmentiert abgelegt. Die Anwendungsobjekte bestehen im Allgemeinen selbst aus Objekten und Wertelisten, so dass das Zusammenfügen bzw. das Abfragen eines Objektes eine Reihe von teueren[3] Verbundoperationen beinhaltet.

Nach [Lu03] ist vor allem das „magere Typsystem“ Ziel der Kritik geworden. Zum einen sollen umfangreiche Daten, die nicht im Rahmen von Datenbankanwendungen entstanden sind oder gepflegt werden, wie z. B. Texte, Bilder, Office-Dokumente oder Geoinformationen, in das Datenbanksystem eingebunden werden, um den integrierten Datenzugriff und Persistenzmechanismen nutzen zu können. Und zum anderen erzwingt ein zu einfaches Typensystem die bereits angesprochene unnatürliche Verteilung von komplex modellierten Daten mit den bekannten Folgen.

2.1.2. Objektrelationale Datenbanksysteme

Das geschaffene Bindeglied zwischen relationalen und objektorientierten Datenbanken sind die so genannten objektrelationalen Datenbanksysteme (ORDBS).

Die komplexen und vielfältig verbundenen Datenobjekte (wie z. B. ein Kundenauftrag) bilden die Basis objektorientierter Anwendungen. Ein Objekt, als Instanz einer Klasse, umfasst

[1] Zusammenfassung von n geordneten Elementen, z. B. das Paar aus Definitions- und Wertebereich

[2] hier: nicht weiter zerlegbar

[3] gemeint ist: hoher Ressourcenbedarf.

neben den Daten selbst auch die Methoden zum Lesen und Ändern dieser Daten. Wird ein Relationales Datenbanksystem um objektorientierte Datentypen und Methoden erweitert, wird von einem Objektrelationale Datenbanksystem gesprochen. Die objektorientierte Spracherweiterung wurde dabei als SQL:1999 normiert. Das beinhaltet den Umgang mit großen Datenmengen (LOB-Typen, Large Binary Object), nutzerdefinierte (strukturierte) Datentypen und Routinen, Typengeneratoren und typisierte Tabellen. [Lu03]
Am Beispiel eines Objekts Kundenauftrag könnten die strukturierten Datentypen Merkmalskombinationen für verschiedene Produktionsstufen sein, ein LOB-Typ wäre eine Explosionszeichnung des zu fertigenden Produkts.
Da die in Datenbanksystemen gespeicherten Daten in Zukunft immer komplexer und umfangreicher werden, gingen Stonebreaker and Moore davon aus, dass objektrelationale DBMS die relationalen DBMS vollständig ersetzen werden. Aktuell sind nur noch wenige anerkannte DBMS am Markt erhältlich, die keine objektrelationalen Erweiterungen anbieten. Es ist dabei nicht definiert, wie hoch der Grad an Objektorientierung zu sein hat, damit eine DBMS als ORDBMS gilt, so dass mittlerweile fast alle bekannten Datenbanken als objektrelational bezeichnet werden können. [Bo03]

2.1.3. Anforderungen an ein DBMS nach Codd

Der britische Mathematiker Edgar F. Codd formulierte in seinen „12 rules“ eine Reihe von Regeln – zunächst 12, später auf 18 erweitert – an ein DBMS, genauer gesagt. an ein Relationes Datenbankmanagementsystem (RDBMS) [Co90].

Informationsregel

Die erste Regel besagt, dass die Daten ausschließlich in Tabellen, nämlich als Werte in Spaltenpositionen innerhalb von Zeilen, abgelegt werden müssen. Ein Wert ist also immer als Intersektion einer Zeile (auch: „Tupel“) mit einer Spalte („Attribut“).

Garantierte Zugriffsregel

Alle Werte einer Datenbank müssen eindeutig identifizierbar und zugreifbar sein, d. h. jeder Wert muss logisch durch Tabellenname, Spaltenname und durch einen Primärschlüssel der beinhaltenden Zeile erreichbar sein.

Systematische Behandlung von NULL-Werten

Der Wert NULL ist nicht identisch mit der Zahl 0, sondern steht für eine fehlende oder unzureichende Information. Steht z. B. für eine Ware noch kein Preis fest, so darf aus offensichtlichen Gründen nicht der Wert 0 dafür verwendet werden. Stattdessen muss die fehlende Information mittels eines einheitlichen Ansatzes repräsentiert werden können.

Relationaler Online-Katalog

Datenbeschreibungen oder Metadaten beschreiben die Struktur einer Datenbank und die Verbindungen ihrer Objekte untereinander. Dieses „Data Dictionary“ muss für den Benutzer mit den gleichen Abfragemitteln zugänglich sein, wie die (Nutz-) Daten in der Datenbank.

Datensprache

Codd fordert eine Sprache, die Datendefinition, Datenmanipulation, Authentisierung, Integritätseinschränkungen und Transaktionsverarbeitung unterstützt. Diese Sprache soll sowohl interaktiv als auch eingebettet benutzt werden können. Als de facto Standard hat sich SQL (Structured Query Language) etabliert. SQL ist eine lineare, nichtprozedurale oder deklarative Sprache. Sie erlaubt dem Benutzer auszudrücken, was er von der Datenbank will, ohne angeben zu müssen, wo die Daten liegen oder wie diese erhalten werden sollen. [Ka04]

Benutzersichten

Die Präsentation der Daten darf nicht auf Tabellen beschränkt sein. Das DBMS muss daher die Erzeugung, Verwaltung und Kontrolle von Sichten unterstützen, die als eine Art „virtuelle Tabelle“ agieren und dem Benutzer dynamisch erzeugte Teile einer oder mehrerer Tabellen zeigen.

High-level Operationen

Das DBMS muss Operationen unterstützen, die das Einfügen (insert), Ändern (update) und Löschen (delete) von beliebigen Sets, nicht nur von einzelnen Zeilen einer einzelnen Tabelle, ermöglichen. Mit Sets sind alle Abfrageergebnisse gemeint, die durch Operationen, wie Selektion, Projektion, Join, Union, Intersektion, Division und Differenz entstehen können.

Physikalische Datenunabhängigkeit

Änderungen auf der physischen Ebene der Datenhaltung dürfen keinesfalls einer Änderung an einer Applikation auf dieser Struktur nach sich ziehen. Das bedeutet, dass für eine Applikation die physische Datenrepräsentation transparent ist.

Logische Datenunabhängigkeit

Änderungen auf der logischen Ebene (Tabellen, Spalten, Zeilen) dürfen keine Änderung an Applikationen auf dieser Struktur nach sich ziehen.

Integritätssicherung

Das DBMS stellt eine Konsistenzüberwachung zur Verfügung, die die Gewährleistung für die Korrektheit von Datenbankinhalten übernimmt. Änderungen können nur ausgeführt werden, wenn sie die Konsistenz nicht verletzen. Diese Konsistenzüberwachung wird im Katalog abgelegt.

Verteilungsunabhängigkeit

Die Verteilung der Daten auf verschiedene Orte (Speicherbereiche, Server, Datenbanken) muss für den Benutzer der Datenbank transparent sein.

Nichtgefährdung

Ein Zugriff auf das DBMS unter Umgehung der Datenintegrität muss verhindert werden.

Regel 0

1990 wurde von Codd eine Regel 0 hinzugefügt, die besagt, dass jedes System, das beansprucht, eine relationale Datenbank zu sein, in der Lage sein muss, Datenbanken ausschließlich durch seine relationalen Fähigkeiten zu verwalten.

Weitere Forderungen

Codd hat seiner Liste 1990 weitere Punkte hinzugefügt, die u. a. Regeln für den Katalog, für Datentypen (Domänen) und Autorisierung umfassen.

Ein weiter Eckpunkt relationaler Datenbanken ist das Transaktionskonzept. Eine Transaktion ist eine Menge von Datenmanipulationen, die zu einer Funktionseinheit zusammengefasst werden. Diese Funktionseinheit wird als Ganzes ausgeführt und permanent gespeichert oder als Ganzes abgelehnt. Mehrere Transaktionen verschiedener Benutzer, die um die gleiche Ressource konkurrieren, müssen vom DBMS synchronisiert werden, so dass gegenseitige Störungen vermieden werden.

Einige der Regeln, z. B. die logische Datenunabhängigkeit, sind so streng, dass kein bis heute erhältliches DBMS von sich behaupten kann, ein vollständig relationales Datenbanksystem gemäß Codd zu sein.

2.1.4. Normalisierung

Um bei der Speicherung von Daten in einer Datenbank Inkonsistenzen vermeiden zu können, sollen die Daten möglichst redundanzfrei abgelegt werden. Durch Normalisierungen soll gewährleistet werden, dass dieses Ziel erreicht wird. Unter der Normalisierung der Daten wird die schrittweise Zerlegung von Relationen in mehrere Teilrelationen auf der Basis funktionaler Abhängigkeiten verstanden.
Eine funktionelle Abhängigkeit besteht, wenn ein Attribut einer Relation die Ausprägungen anderer Attribute dieser Relation eindeutig bestimmt. In einer Datenbank über die Mitarbeiter eines Unternehmens würde z. B. die Personalnummer eindeutig den Namen und die Abteilung eines Mitarbeiters bestimmen. Wenn ein oder mehrere Attribute eindeutig die Werte aller Attribute dieser Relation bestimmen, wird von einem Superschlüssel gesprochen. Im obigen Beispiel wäre die *Personalnummer* ein Superschlüssel der Relation *Mitarbeiter*.

P	N	A
1020	Müller	EW
1030	Schulze	BT
1040	Müller	BT

Tabelle 1: Superschlüssel P (Personalnummer)

In diesem Beispiel bestimmt die Personalnummer P eindeutig den Namen N und die Abteilung A, d. h. P → N, A.

A	B	C
1	1	3
1	1	3
1	2	4

Tabelle 2: A,B → C

Im diesem Beispiel ist A,B → C. C ist jedoch nicht allein funktional abhängig von A. Es gelten auch die funktionalen Abhängigkeiten „A ist funktional abhängig von C“, „A ist funktional abhängig von B“ und „C ist funktional abhängig von B“.

Formale Definition: Funktionale Abhängigkeit

Sei r(R) eine Relation mit dem Relationenschema R und seien α und β Teilmengen von Attributen von R. Sei $t \in R$ ein Tupel aus r. Dann ist t[α] die Einschränkung von t auf die

Attribute aus α. Die funktionale Abhängigkeit α → β (β ist funktional abhängig von α) gilt auf R, wenn für jede zulässige Relation r(R) gilt:

$$\forall t_1, t_2 \in r : t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

Das heißt, für alle Tupel t_1,t_2 aus r mit den gleichen Attributen α ($t_1[\alpha] = t_2[\alpha]$), gilt, dass ihre β-Attribute auch gleich sind. Die Werte der Attribute aus der Attributmenge α bestimmen also eindeutig die Werte der Attribute aus der Attributmenge β.

Die Attributmenge β ist genau dann **voll funktional abhängig** von α, wenn aus α kein Attribut entfernt werden kann, so dass die Bedingung immer noch gilt.

Aus der funktionalen Abhängigkeit in der **Tabelle 2:** A,B → C lässt sich das Attribut A entfernen, so dass sich die volle funktionale Abhängigkeit B → C ergibt. [MS04]

Erste Normalform

Die erste Normalform (NF1) wird mit zwei Forderungen umrissen:

- Attribute müssen atomar sein
- die Relation muss frei von sich wiederholende Gruppen sein

Nach der ersten Forderung dürfen einem Attribut nicht mehrere Werte aus einem definierten Gültigkeitsbereich zugeordnet werden. Geschachtelte, mengenwertige oder zusammengesetzte Attribute sind nicht erlaubt.

Produkt_ID	Produkt	Bestandteile
1	Hajos Erdbeerbrause	{1. Wasser, 2. Zucker, 3. Aroma, 4. Farbstoffe}
2	Griff Reinil	{1. Tenside, 2. Parfüm}

Tabelle 3: NF1 – Produkte

Das obige Beispiel verstößt mehrfach gegen die Auflagen der ersten Normalform. So liegen in der Spalte „Bestandteile“ mehrere eigentlich atomare Attribute verknüpft vor und das Feld „Produkt“ beinhaltet Hersteller und Produktname. Dabei entstehen mehrere Probleme bei der Abfrage. So kann nicht sinnvoll über den Produktnamen sortiert werden und die Bestandteile lassen sich nur als Liste (String) ausgeben.

Daher werden die Felder in atomare Attributwerte aufgespalten:

Produkt_ID	Hersteller	Produkt	Bestandteil_Nr	Bestandteile
1	Hajos	Erdbeerbrause	1	Wasser
1	Hajos	Erdbeerbrause	2	Zucker
1	Hajos	Erdbeerbrause	3	Aroma

1	Hajos	Erdbeerbrause	4	Farbstoffe
2	Griff	Reinil	1	Tenside
2	Griff	Reinil	2	Parfüm

Tabelle 4: NF1 – Produkte

Alle Attributwertebereiche der Tabelle sind jetzt atomar. Durch die Normalisierung ergibt sich ein Primärschlüssel über Produkt_ID und Bestandteil_Nr.

Zweite Normalform

Die zweite Normalform (NF2) liegt vor, wenn NF1 erfüllt ist und zusätzlich alle Nicht-Schlüsselattribute voll funktional abhängig vom Schlüsselkandidaten sind.
Der Vorteil der zweiten Normalform liegt im Zwang mehrere Sachverhalte einer Relation in eine neue Relation auszulagern. Geschieht dies nicht, besteht die Gefahr, die Integrität der Daten zu verletzen. Würde das Beispiel der **Tabelle 4:** NF1 – Produkte so in einer Datenbank realisiert, würde ein Update auf das Feld „Hersteller“ in der ersten Zeile die Daten inkonsistent werden lassen (Update-Anomalie[4]).

Produkt_ID	Hersteller	Produkt	Bestandteil_Nr	Bestandteile
1	Peters	Erdbeerbrause	1	Wasser
1	Hajos	Erdbeerbrause	2	Zucker
1	Hajos	Erdbeerbrause	3	Aroma
1	Hajos	Erdbeerbrause	4	Farbstoffe
2	Griff	Reinil	1	Tenside
2	Griff	Reinil	2	Parfüm

Tabelle 5: NF1 – Produkte

Mit dem Produkt 1 (Erdbeerbrause) wären nun zwei Hersteller verknüpft. In den Anwendungsprogrammen müssten aufwändige Routinen gepflegt werden, die die Datenkonsistenz sicherstellen. Über den interaktiven Zugriff wären solche Operationen dennoch erlaubt.

[4] Die sog. Delete- oder die Insert-Anomalie können weitere Inkonsistenzen hervorrufen.

Das Aufspalten der Tabelle „Produkte“ in zwei getrennte Relationen „Produkte“ und „Bestandteile“ genügt der zweiten Normalform und macht oben beschrieben Inkonsistenzen unmöglich.

Produkt_ID	Hersteller	Produkt
1	Hajos	Erdbeerbrause
2	Griff	Reinil

Tabelle 6: NF2 – Produkte

Produkt_ID	Bestandteil_Nr	Bestandteile
1	1	Wasser
1	2	Zucker
1	3	Aroma
1	4	Farbstoffe
2	1	Tenside
2	2	Parfüm

Tabelle 7: NF2 – Bestandteile

In diesem Beispiel ist die Produkt_ID der Primärschlüssel der Relation „Produkte“ und gleichzeitig der Fremdschlüssel in der Relation „Bestandteile“. Der Primärschlüssel von „Bestandteile“ besteht aus den Feldern „Produkt_ID“ und „Bestandteil_Nr“ und ist somit ein zusammengesetzter Primärschlüssel.

Dritte Normalform

Das Ziel der dritten Normalform (NF3) ist die Beseitigung von Abhängigkeiten zwischen Nichtschlüsselattributen. Würde in obige Tabelle „Produkte“ z. B. ein Attribut „Firmensitz“ eingefügt werden, wäre dieses Nichtschlüsselattribut nicht vom Primärschlüssel „Produkt_ID“, sondern vom Nichtschlüsselattribut „Hersteller“ abhängig.

Produkt_ID	Hersteller	Produkt	Firmensitz
1	Hajos	Erdbeerbrause	Braunschweig
2	Griff	Reinil	Wolfenbüttel

Tabelle 8: NF2 → NF3 – Produkte

Die dritte Normalform fordert das Bestehen der NF2 und die Aufhebung der transitiven Abhängigkeit eines Nichtschlüssels von einem Schlüsselkandidaten.

Das Attribut A_2 (im Beispiel „Firmensitz“) hängt über ein anderes Attribut A_1 („Hersteller“) vom Primärschlüssel P_1 („Produkt_ID“) der Relation ab, aber gleichzeitig ist A_2 nicht direkt von P_1 abhängig.

$$P_1 \rightarrow A_1 \wedge A_1 \rightarrow A_2 \Rightarrow P_1 \rightarrow A_2$$

Hängt Attribut A_1 vom Primärattribut P_1 ab und Attribut A_2 hängt von A_1 ab, dann ist A_2 transitiv abhängig von P_1.

Das Überführen einer Relation von NF2 in die NF3 durch die Aufhebung transitiver Abhängigkeiten erzeugt monothematische Relationen. Diese Spalten werden zunächst entfernt und in neue Relationen ausgelagert.

Produkt_ID	Hersteller	Produkt
1	Hajos	Erdbeerbrause
2	Griff	Reinil

Tabelle 9: NF3 – Produkte

Hersteller	Firmensitz
Hajos	Braunschweig
Griff	Wolfenbüttel

Tabelle 10: NF3 – Firma

Die durch die Auslagerung des transitiven Attributs entstandene Relation wird durch das Nichtschlüsselattribut der ursprünglichen Tabelle ergänzt, welches damit zum Fremdschlüssel wird.

Zusammenfassend müssen zur Erfüllung der dritten Normalform alle Attribute einer Tabelle vollständig durch den Primärschlüssel determinierbar sein. Bestehen zwischen den Attributen untereinander Abhängigkeiten, so sind diese in eigene Tabellen zu isolieren. [Au06]

Weitere Normalformen

Die Boyce-Codd-Normalform (BCNF) gilt gemeinhin als Verschärfung der dritten Normalform, in der zusätzlich alle voll funktionalen Abhängigkeiten vom Primärschlüssel ausgehen. Umgekehrt gesagt, muss für die BCNF jede Determinante ein Schlüssel sein. Ziel

der BCNF ist die Verhinderung von Anomalien bei zusammengesetzten Schlüsseln, bei denen sich die Attribute überlappen.

Mit der vierten Normalform (NF4) werden mehrwertige Abhängigkeiten in separate Relation ausgelagert. Mehrwertige Abhängigkeiten lägen vor, wenn zu einem Mitarbeiter einer Firma (Personalnummer) noch dessen Haustier und das Auto gespeichert werden soll. Besitzt ein Mitarbeiter mehr als ein Haustier und mehr als ein Fahrzeug, gibt es zwischen den Attributen Haustier und Fahrzeug zwar keine funktionalen Abhängigkeiten aber eine mehrwertige Abhängigkeit.

Personalnummer	Haustier	Fahrzeug
1	Hund	Trabant
1	Goldfisch	Volvo 960

Tabelle 11: Mehrwertige Abhängigkeit

Es ist offensichtlich, dass es keine funktionalen Abhängigkeiten zwischen A_1 (Haustier) und A_2 (Fahrzeug) gibt, denn weder Hund noch Goldfisch besitzen ein Fahrzeug. Als Lösung wird die Relation in zwei Tabellen aufgespaltet, die dann in der NF4 vorliegen:

Personalnummer	Haustier
1	Hund
1	Goldfisch

Tabelle 12: Vierte Normalform – Haustiere

Personalnummer	Fahrzeug
1	Trabant
1	Volvo 960

Tabelle 13: Vierte Normalform - Fahrzeuge

Eine Relation, die sich in der vierten Normalform befindet und für deren Join-Abhängigkeiten (R_1 bis R_n) gilt:

- die Join-Abhängigkeit ist trivial oder
- jede R_i ist Schlüsselkandidat der Relation

befindet sich in der fünften Normalform. Solche Relationen lassen sich nicht in weitere Relationen aufspalten, ohne dass Information verloren geht. [Ke98]

2.1.5. SQL

Die Structured Query Language (SQL) ist eine mengenorientierte, nicht-prozedurale Sprache der vierten Generation (4GL - Fourth-Generation-Language), die zur Abfrage, Manipulation, Definition und Kontrolle von relationalen Datenbanken benutzt wird. Sie enthält alle erforderlichen Sprachelemente, um sämtliche Aufgaben, die im Bereich der relationalen Datenbank anfallen, zu erfüllen. SQL wird als de-facto-Standard in nahezu allen gängigen relationalen Datenbanksystemen benutzt. Mit SQL wird lediglich formuliert, *welche* Daten zu selektieren (bzw. zu ändern, zu löschen) sind, aber nicht auf welchem Wege dies geschehen soll. In einer klassischen Programmiersprache (wie z. B. C) müsste zunächst die die Information enthaltene Datei spezifiziert und geöffnet werden, bevor die Daten satzweise extrahiert und auf Übereinstimmung mit den abgefragten Bedingungen überprüft werden können.

Die Firma IBM entwickelte in den 1970er Jahren im Rahmen eines Forschungsprojektes „System R“. Dabei handelte es sich um das erste relationale DBMS überhaupt. Auf alle Daten in „System R“ wurde mittels der eigens dafür geschaffenen Sprache SQL (vorher „SEQUEL2“ – Structured English Query Language 2 - genannt) zugegriffen. [BAC99]
Im Jahre 1986 wurde SQL als „SQL1“ von der ANSI (American National Standards Institute) und 1987 von der ISO (Internationale Organisation für Normung) als Norm verabschiedet und in den folgenden Jahren mehrfach erweitert zum aktuellen SQL:2003.

SQL kann in verschiedene Untergruppen aufgeteilt werden, die jeweils nach den zu erfüllenden Aufgaben benannt sind.

1. Datenabfrage, gelegentlich „Data Query Language“ (DQL) genannt, z. B. SELECT
2. Datenänderung, „Data Manipulation Language“ (DML), z. B. UPDATE, DELETE
3. Datendefintion, „Data Definition Language“ (DDL), z. B. CREATE, TRUNCATE
4. Rechteverwaltung, „Data Control Language“ (DCL), z. B. GRANT, REVOKE

Mit Hilfe der SQL-Sprachkonstrukte COMMIT und ROLLBACK werden Änderungen in einer Datenbank in einer Transaktion, als Folge von Operationen, gekapselt. Alle von einem Benutzer bzw. einem Programm durchgeführten Änderungen sind in anderen Sitzungen[5] erst dann sichtbar, wenn sie mit dem Abschluss einer Transaktion (durch „COMMIT“) persistent

[5] Engl. „session“, gemeint ist die Verbindung eines Clients mit einem Server (hier: der Datenbank).

gemacht werden. Das Beenden einer Transaktion, bei dem alle Änderungen verworfen werden sollen, nennt man, wie das Gleichlautende SQL-Kommando, Rollback.
SQL bietet weiterhin Kommandos aus dem DDL-Komplex zur Überwachung, genauer: zur Erzwingung, der referenziellen Integrität (RI). Damit ist die Korrektheit zwischen Attributen von Relationen und der Erhaltung der Eindeutigkeit ihrer Schlüssel gemeint. Das bedeutet, dass jede Ausprägung eines Fremdschlüssels auch als Wert des zugehörigen Primärschlüssels existieren muss. [Ke98]

Um in Anwendungsprogrammen auf relationale Datenbanken zugreifen zu können, kann SQL mit anderen Programmiersprachen verbunden werden. Bei „Embedded SQL“ werden SQL-Statements einfach innerhalb des Quelltexts (C, C++, Pascal o. ä.) eingefügt und vom Precompiler in Funktionsaufrufe übersetzt. Beispiele für Implementierungen von Embedded SQL sind „Pro*C“ (C, C++) und „SQLJ“ (Java).

Über die standardisierte Datenbankschnittstelle ODBC (Open Database Connectivity) oder das Java-Pendant JDBC (Java Database Connectivity) können SQL-Befehle über einen Funktionsaufruf direkt an das DBMS übergeben werden.

Als dritte Möglichkeit einer Programmierschnittstelle stehen so genannte Persistenz-Frameworks für Java („Hibernate“) oder C# („NHibernate“) zur Verfügung, die es ermöglichen den Zustand eines Objektes in einer relationalen Datenbank abzulegen.

Wird SQL in Anwendungsprogrammen verwendet, so kann zwischen statischem und dynamischem SQL unterschieden werden. Bei statischem SQL ist die SQL-Anweisung dem DBMS zum Zeitpunkt der Kompilierung bekannt, bei dynamischem SQL wird eine Abfrage erst zur Laufzeit bekannt, z. B. wenn sie aus Strings zusammengesetzt wird.

Der SQL-Standard wurde ab 1995 mit mehreren Paketen erweitert, um Funktionen, die einige Datenbankhersteller schon in ihre Systeme integriert hatten, auf Basis eines Standards zu stellen. Mit dem „Call Level Interface“ (SQL/CLI) wurde eine aufrufbare Schnittstelle spezifiziert, die im Gegensatz zu den statischen Eigenschaften von Embedded SQL eine hohe Dynamik für Ad-hoc-Anwendungen bereitstellt. [MKF03]
Mit dem Erscheinen der Norm „Persistent Stored Modules“ (SQL/PSM) im Jahre 1996 können prozedurale Konstrukte nahtlos in SQL eingebunden werden. [MKF03]

Dadurch wird es möglich Anwendungslogik in die Datenbank auszulagern. Neben dem Ablegen von Prozeduren und Funktionen im Data Dictionary können auch Trigger direkt in der Datenbank programmiert werden. Mit Hilfe dieser Konstrukte kann die Datenbank selbständig[6] eine erweiterte Integritätssicherung gewährleisten, die ereignisgesteuert nach oder vor einem Insert, Update oder Delete bestimmte Operationen durchführt.
Darüber hinaus existieren SQL-Erweiterungen, um XML-Dokumente in der Datenbank zu speichern (SQL/XML), um nicht-relationale Datenquellen, wie z. B. Betriebssystemdateien, abzufragen und zu manipulieren (SQL/MED, „Management of External Data“), sowie um die Schnittstelle zwischen Java und SQL zu verbessern (SQL/JRT, „Routines and Types Using the Java Programming Language“) und weitere hier nicht genannte Standards.

2.2. Anbieter

Gegenwärtig ist eine Vielzahl RDBMS mit den verschiedensten Lizenzmodellen am Markt erhältlich. An dieser Stelle wird nur auf die bekanntesten Systeme Oracle, IBM DB/2, PostgreSQL, MySQL, MaxDB und Microsoft SQL Server eingegangen. In dieser Diplomarbeit werden insbesondere die verwendeten Zugriffsstrukturen und Verfahren zur Speicherverwaltung im Mittelpunkt stehen.

2.2.1. Oracle Database

Im Jahre 1977 wurde von Larry Ellison, Bob Minder und Ed Oates die Firma *Software Development Laboratories* gegründet, die damit begann, ein zum *System R Database* der Firma IBM (vgl. 2.1.5) kompatibles relationales Datenbanksystem zu entwickeln. Es entstand die erste Version eines *Oracle* genannten Datenbanksystems. Von 1979 bis 1983 hieß das Unternehmen *Relational Software, Inc*. (RSI), bevor es in *Oracle* umbenannt wurde. Mittlerweile wird fast die Hälfte des weltweiten Umsatzes mit relationalen Datenbanken von Oracle[7] erwirtschaftet.
Die Oracle Database ist für eine ganze Reihe von Betriebssystemen erhältlich, darunter Windows, Linux , Unix und MacOS. Aber auch für VMS-Derivate und z/OS bietet Oracle sein Produkt an.

[6] ohne Einwirkung eines Anwendungsprogramms von „außen“.

[7] Quelle: Gartner, Inc.

Die aktuelle Version ist „10g“, der Nachfolger „11g“ genannt ist bereits am 11. Juli 2007 offiziell gestartet, aber derzeit[8] noch nicht verfügbar. Besonders die fortgeschrittene Partitionierung und Kompression, das Einspielen von Upgrades und Patches ohne Shutdown der Datenbank und auf die Wiederherstellung einer Datensituation zu einem beliebigen Zeitpunkt[9] gehören zu den von Oracle hervorgehobenen Neuerungen.
Oracle bietet eine kostenlose nutzbare „Express Edition (XE)“ an, die neben diversen Einschränkungen beim Support eine Datenobergrenze von 4 GB aufweist. Weitere Lizenzmodelle basieren auf der Anzahl der benannten Benutzer bzw. auf der Anzahl der Prozessoren, auf der die DB läuft. Für größere Unternehmen sind ebenfalls so genannte „Standortlizenzen“ erhältlich, mit der beliebige viele Benutzer in beliebig vielen Datenbanken arbeiten dürfen.

2.2.2. IBM DB2

Mit DB2 ist der Nachfolger des bereits genannten *System R* von IBM am Markt vertreten. Nach dessen Start im Jahre 1978 offerierte IBM 1982 zwei Weiterentwicklungen: SQL/DS und DB2. Diese beiden Zweige hielten sich bis in die 1990er Jahre, als IBM beide Produkte unter dem Markennamen DB2 zusammenfasste.
Das DBMS war zunächst nur für IBM Mainframe-Betriebssysteme erhältlich, bis – ebenfalls in den 1990er Jahren – die so genannten LUW-Versionen erschienen: „Linux, Unix, Windows“.
Im Jahre 2001 erwarb IBM die Firma „Informix“ und erweiterte DB2 um eine ganze Reihe von objektrelationalen Komponenten, so dass auch DB2 heutzutage als ORDBMS gilt.
Seit Anfang des Jahres 2007 ist die Version 9 von DB2 erhältlich, die über erweiterte XML-Funktionen und Datentypen, OLAP-Unterstützung (RANK, DENSE_RANK, ROW_NUMBER), Schnitt-, Differenzmengen- und Merge-Operationen verfügt. Ebenfalls neu ist das bei den Mitbewerbern schon länger verfügbare prozedurale Erweiterungen (SQLPM) und das TRUNCATE-Kommando zum sofortigen Leeren einer Tabelle.
Auch IBM bietet, wie Oracle, eine Gratisversion seiner Datenbank an. Die „Express-C“ genannte Lizenz erlaubt ebenfalls nur maximal 4 GB an Daten und limitiert auch die Anzahl der Prozessoren auf zwei. Weitere Lizenzmodelle basieren ebenfalls auf der Anzahl der benannten Benutzer oder der Anzahl der Prozessoren.

[8] August 2007

[9] „Total Recall“

2.2.3. PostgreSQL

Anders als die beiden sich aus *System R* ableitenden DBMS „Oracle“ und „DB2“ liegen die Wurzeln von PostgreSQL in einem „Ingres“ genannten Projekt der US-Amerikanischen „University of California“ in Berkley unter der Leitung von Michael Stonebreaker. Nach einem zwischenzeitlichen Intermezzo in der freien Wirtschaft von 1982 – 1985 kehrte Stonebreaker an die Berkley-Universität zurück und begann die Arbeit an einem Nachfolger für Ingres: Post-Ingres oder kurz „Postgres“.
Bei der Entwicklung von Postgres in den Jahren 1986 – 1994 war es das erklärte Ziel, ein DBMS zu entwickeln, das sich zum einen streng an den relationalen Grundsätzen von E. F. Codd (vgl. 2.1.3) orientiert und zum anderen objektrelationale Technologie mit einbezieht.
Mit der Ersetzung der bisher implementierten Abfragesprache „POSTQUEL“ durch SQL im Jahre 1995 entstand eine „Postgres95“ genannte Zwischenstufe, die 1996 erstmals unter dem heutigen Namen „PostgreSQL“ als Open-Source-Software in der Version 6.0 veröffentlicht wurde [2007a].
Seit April 2007 ist die aktuelle Version 8.2.4 verfügbar, die laut postgresql.org die ANSI-SQL-Standards SQL92 und SQL99 abdeckt und auch Teile von SQL:2003 implementiert. Dabei werden Aggregatfunktionen für statistische Auswertungen, VALUE-Zuweisungen für mehrere Zeilen zugleich, UPDATE RETURNING und Aggregate über mehreren Spalten unterstützt.
Das PostgreSQL DBMS ist für diverse Unix-Plattformen, für Linux und für Windows erhältlich. Dabei werden sogar noch die MS-DOS-basierten Windows-Versionen „Windows 95“, „Windows 98“ und „Windows Me“ über eine Cygwin-Zwischenschicht unterstützt. Es existieren ebenfalls Portierungen für OS/2 und Novell Netware 6.
PostgreSQL wird unter der klassischen BSD-Lizenz vertrieben. Das bedeutet, dass jeder Anwender den Quellcode beliebig ändern darf. Daraus gegebenenfalls erstellte Binärdateien dürften sogar kostenpflichtig und ohne Quellcodeoffenlegung vertrieben werden.
Durch das Fehlen der bei Oracle und DB2 unter Umständen doch beträchtlichen Ausgaben für Lizenzen, ergibt sich ein Kostenvorteil, den mittlerweile auch bekannte Unternehmen wie „Sun Microsystems“, „Cisco“ oder „Fujitsu“[10] nutzen.

[10] Quelle: http://www.postgresql.org/about/users

2.2.4. MySQL

Das DBMS „MySQL“ der schwedischen Firma „MySQL AB“ ist als die Datenbank des World Wide Web zu großer Bekanntheit gelangt.
Das erste veröffentlichte Release der MySQL-Datenbank aus dem Jahr 1995 trug bereits die Versionsnummer 3.21 und sollte offensichtlich eine lange Entwicklungsgeschichte suggerieren. Diese Version bot mit dem einzig möglichen Tabellentyp „ISAM“ weder Transaktionen noch referenzielle Integrität und war somit für Anwendungen etwa in unternehmenskritischen Bereichen nicht interessant - ganz im Gegensatz zu der Verwendung auf Webservern (meist: „Apache“) in Kombination mit der Skriptsprache PHP.
Auch mit der aktuellen Version 5.0 werden nur Teile der SQL:99 Spezifika abgedeckt. Im Vergleich zu früheren Versionen werden ab 5.0 u. a. prozedurale Erweiterungen, Trigger und Views unterstützt. Ein interessantes Feature ist ferner die Master/Slave-Replikation, bei der ein Master ein binäres Log schreibt, das die Slaves lesen und die enthaltenen Queries ausführen.
Mittels des Tabellentyps „MyISAM“ – als Nachfolger von „ISAM“ – lassen sich speziell auf Lesegeschwindigkeit optimierte Tabellen erstellen, bei denen Transaktionen nicht relevant sind, und somit zeitraubende Locking-Mechanismen entfallen können.
Seit Oktober 2006 ist MySQL in zwei verschiedenen Varianten erhältlich. Zum einen als „MySQL Community Server“ und als „MySQL Enterprise Server“. Letztere Variante kann mit regelmäßigen und kontrollierten Updates und mit einem erweiterten Support punkten, der derzeit ab 595 US-Dollar pro Jahr und Server kostet – je nach gewünschter Reaktionszeit des Supports. Die Community-Version ist weiterhin kostenlos.
MySQL ist für Linux, Windows (auch in der 64-Bit-Variante), verschieden Unix-Derivate (z. B. HP, Solaris, AIX), für MacOS, FreeBSD und weitere Plattformen erhältlich. Zusätzlich besteht natürlich, wie auch bei PostgreSQL, die Möglichkeit den Quelltext auf einer beliebigen Plattform zu übersetzen.

2.2.5. MaxDB

Das heute unter dem Namen „MaxDB“ erhältliche DBMS hat eine wechselvolle Geschichte hinter sich. Die Anfänge liegen im Jahr 1977 in einem Projekt von Dr. Rudolf Munz an der TU Berlin in Zusammenarbeit mit der Nixdorf Computer AG. Der erste Name lautete „Verteilte Datenbanksysteme Nixdorf“ (VDN) und wurde später in „Relationales Datenbanksystem“ (RDS) umbenannt. Nach weiteren Umbenennungen (Reflex, Supra 2,

DDB/4) wurde das DBMS im Zeitraum von 1981 bis 1989 von der Nixdorf Computer AG vertrieben. Anschließend erfolgte die Ausgliederung unter dem neuen Namen „Entire SQL-DB“ in das Unternehmen SQL Datenbanksysteme GmbH in Berlin mit Dr. Rudolf Munz als Geschäftsführer. Die Rechte wurden 1992 an die Software AG verkauft, wo der Name „Adabas D“ für das Produkt entstand. Nachdem die SAP AG das „Adabas D“ DBMS ab 1994 für ihre ERP-Software SAP R/3 benutzte, wurde das Produkt 1997 erworben und in „SAP DB“ umbenannt. [JH2005]

Im Jahr 2003 schlossen SAP and MySQL ein Kooperationsabkommen, das es der MySQL AB ermöglich, das DBMS „SAP DB“ mit der Version 7.5 unter dem Namen „MaxDB“ zu vertreiben. Die Verantwortung für die Weiterverentwicklung ist bei der SAP AG verblieben.

Laut MaxDB-Dokumentation wird lediglich der SQL-92-Standard unterstützt, wobei einige Anforderungen späterer Standards, wie zum Beispiel SQL/PSM zur Bereitstellung von prozeduralen Erweiterungen, ebenfalls abgedeckt werden.

Die Datenbank ist als Open-Source-Software unter der GPL-Lizenz[11] erhältlich, kann aber auch unter einer kommerziellen Lizenz von SAP oder MySQL erworben werden.

MaxDB ist für Windows, Linux und für drei der bekanntesten UNIX-Derivate – HP, Solaris und AIX – erhältlich.

2.2.6. Microsoft SQL Server

Die Wurzeln des SQL Server RDBMS liegen in der Kooperation von Microsoft mit der Firma Sybase seit Ende der 1980er Jahre. Die ersten Versionen wurden für das Betriebssystem OS/2 veröffentlich, das bis 1991 von Microsoft und IBM gemeinsam entwickelt wurde. Der Microsoft SQL Server in der Version 4.2 erschien 1992 und markierte das Ende der Kooperation mit Sybase, deren Produkt dann den Namen „Adaptive Server Enterprise“ bekam. Mit der Version 4.21. erschien das erste Release für Microsofts neues Betriebssystem Windows NT. In der darauf folgenden Version 6.0 löste sich Microsoft vom Sybase-Quellcode und entwickelte ein komplett eigenes DBMS. Die aktuell letzte Version erschien im Jahr 2005 und trägt den Namen Microsoft SQL Server 2005.

Microsoft benutzt eine T-SQL (Transact-SQL) genannte SQL-Variante, die eine proprietäre Erweiterung des SQL-92-Standards darstellt. Die zusätzlichen Features umfassen eine so genannte Control-of-Flow Language mit der u. a. If-The-Else-Prädikate, While-Schleifen und

[11] GPL – GNU General Public License

BEGIN-END-Blöcke in eine Abfrage eingeflochten werden können. Weiterhin besteht die Möglichkeit der Deklaration von lokalen Variablen und die DML-Kommandos UPDATE und DELETE wurden um die Möglichkeit des Einfügens von Joins erweitert, wohingegen im Standard mit Subselects gearbeitet werden muss. Diese Abweichungen vom Standard machen eine spätere Revidierung der Entscheidung für Microsoft SQL Server zugunsten eines anderen DBMS mit Standard-SQL sehr teuer, da die proprietären Erweiterungen zunächst entfernt bzw. umprogrammiert werden müssen.

Naturgemäß ist der Microsoft SQL Server nur für das Windows-Betriebssystem erhältlich. Es gibt eine ganze Reihe von verschiedenen Editionen zur Auswahl, die unterschiedlichsten Einschränkungen unterliegen. Analog zu Oracle und DB2 ist jedoch auch eine kostenlose, SQL Server 2005 Express Edition genannte, Variante verfügbar, die ähnliche Einschränkungen hinsichtlich Prozessoranzahl und maximaler Datenbankgröße aufweist, wie die Pendants der direkten Mitbewerber Oracle und DB2. [MS05]

3. Zugriffsstrukturen

In dieser Diplomarbeit werden Ansätze für flexible Indexstrukturen beschrieben, die sich nach einer Lastverteilung optimieren. Dazu ist es notwendig, die verwendeten Begriffe und Konzepte an dieser Stelle vorzustellen und zu erläutern.

3.1. Schlüssel

In einer Datenbank werden Schlüssel verwendet, um ein Tupel oder eine Menge von Tupel identifizieren zu können. Anders ausgedrückt, handelt es sich bei einem Schlüssel um eine oder mehrere Spalten einer Tabelle, deren Ausprägung dazu benutzt werden kann, die restlichen Werte dieser Zeile anzusprechen.

3.1.1. Superschlüssel

Eine beliebige Menge an Attributen einer Relation, die die Tupel eindeutig identifiziert, wird Superschlüssel oder Oberschlüssel genannt.
In einer Relation $R(A)$ über die Menge der Attribute $A := \{A_1 \dots A_n\}$ ist $\alpha \subseteq A$ genau dann ein Superschlüssel der Relation R, wenn $\alpha \rightarrow A$. Vereinfacht gesagt, haben zwei Tupel immer die gleichen Werte in den Attributen A, wenn auch die Ausprägungen des Schlüssels α gleich sind.
Vorausgesetzt, dass in einer Relation keine zwei völlig identischen Tupel existieren, handelt es sich um den trivialen Fall des Superschlüssels, der aus allen Attributen der Relation besteht.

3.1.2. Schlüsselkandidat

Würden aus dem trivialen Fall des alle Attribute umfassenden Superschlüssels diejenigen Attribute eliminiert, die für die eindeutige Identifizierung eines der Tupel einer Relation nicht erforderlich sind, ergibt sich ein Schlüsselkandidat. Wenn also eine Menge an Attributen A von einer Untermenge $\alpha \subseteq A$ voll funktional abhängig ist ($\alpha \rightarrow A$), wird von einem Schlüsselkandidaten gesprochen. Schlüsselkandidaten sind die minimal identifizierenden Attributmengen, sie können nicht weiter reduziert werden, ohne dass die Schlüsseleigenschaft verloren ginge.

3.1.3. Primärschlüssel und Sekundärschlüssel

Schlüssel lassen sich in Primärschlüssel und Sekundärschlüssel unterteilen.
Die wichtigste Anforderung an einen Primärschlüssel ist die Duplikatfreiheit der dadurch zugeordneten Attribute. Dadurch bietet sich ein Primärschlüssel immer auch für eine Zugriffsstruktur an, über die auf die Relation bei Abfragen oder Verbundoperation zugegriffen wird. Ein Beispiel für einen Primärschlüssel ist z. B. eine Auftragsnummer, die in der Regel eindeutig[12] gewählt wird.
Im Gegensatz dazu orientiert sich ein Sekundärschlüssel in erster Linie an den prognostizierten Abfragen auf eine Relation. So sind die Attributmengen eines Sekundärschlüssels in aller Regel nicht duplikatfrei und besitzen somit auch keine Schlüsseleigenschaft. Dennoch sind Sekundärschlüssel ein außerordentlich wichtiges Konstrukt eines DBMS, da Zugriffsstrukturen in vielen Fällen entlang eines Sekundärschlüssels gebildet werden. In einer Tabelle mit Aufträgen, in der die Auftragsnummer den Primärschlüssel bildet, ist leicht einsehbar, dass ein Feld wie z. B. „Kundennummer“ zumindest dann nicht duplikatfrei sein kann, wenn mindestens ein Kunde mehr als eine Bestellung beauftragt hat. Wenn so eine Abfrage nach einem Kunden häufig zu erwarten ist, sollte sie durch eine Zugriffsstruktur unterstützt werden.

3.1.4. Surrogatschlüssel

In den oben genannten Beispiel von Auftrags- und Kundennummer sind Primär- und Sekundärschlüssel existierende Datenobjekte, d. h. sowohl eine Auftragsnummer als auch eine Kundennummer wird z. B. auf einem Etikett oder Versandpapier auftauchen. Oft ist es in einer Datenbank nicht zweckmäßig einen natürlichen Schlüssel zu verwenden. Aus einem Kundennamen geht nicht immer eindeutig hervor, um welchen Kunden es sich handelt. So kann ein Unternehmen mehrere Standorte haben oder es gibt zwei Firmen, die den gleichen Namen tragen. Es ist ein zusätzliches Attribut notwendig, um sicherzustellen, dass der korrekte Kunde ausgewählt wird. Hier bietet sich z. B. eine Kundennummer, etwa in Form einer entsprechend langen Sequenznummer, an. Ein solcher künstlich erzeugter Schlüssel heißt Surrogat- oder Kunstschlüssel. Sie vereinfachen die Abfrage nach einem Tupel, da statt eines aus n Attributen zusammengesetzten Schlüssel lediglich ein einzelnes Attribut verwaltet werden muss.

[12] zumindest für einen hinreichend großen Zeitraum

3.2. Index

Ein Index ist eine von den Daten getrennte Struktur innerhalb eines Datenbanksystems, die Teile der Daten in einer speziellen Form enthält, so dass der Zugriff (Selektion, Sortierung) auf bestimmte Bereiche der Daten beschleunigt werden kann.

3.2.1. Primärindex und Sekundärindex

Nutzt ein Zugriffspfad die Dateiorganisationsform[13] der internen Relation aus, wird von einem Primärindex gesprochen. Im häufigsten Fall ist dies eine Indexdatei, die über einem Primärschlüssel gebildet wird. Die Einträge in diesem Index sind einelementig und in der Regel sortiert. Ein Primärindex kann jedoch auch über einem anderen Schlüssel der Relation, sogar über einem Sekundärschlüssel, gebildet werden. [SH99]
Die interne Relation bezeichnet eine Menge von internen Tupeln in der Ebene des Zugriffssystems. Alle weiteren Zugriffspfade auf diese interne Relation werden als Sekundärindex bezeichnet. Dabei kann die Dateiorganisationsform der internen Relation von einem Sekundärindex nicht ausgenutzt werden. Daher besitzt ein Sekundärindex meist eine andere Struktur als ein Primärindex. [SH99]

3.2.2. Dünn- und dichtbesetzter Index

Ein weiteres Unterscheidungsmerkmal von Zugriffspfaden ist die Besatzdichte der internen Relation. In einem dünnbesetzten Index existiert nicht für jeden Schlüsselwert ein Eintrag in einer Indexdatei. Der Index verweist dann mit seinen Zugriffsattributwerten jeweils auf den Seitenanführer der internen Relation, die nach den Zugriffsattributen sortiert sein muss.

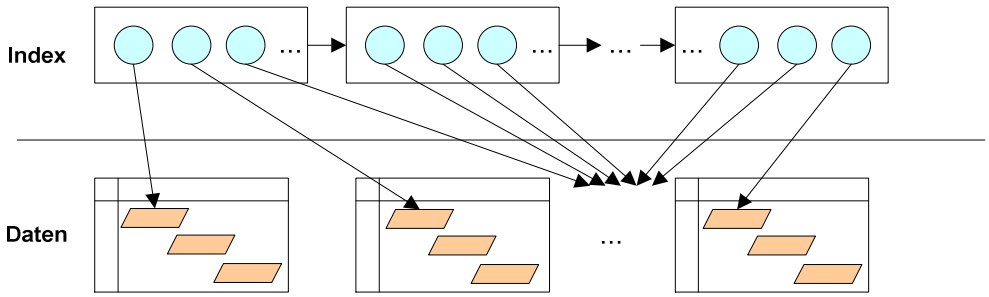


Abbildung 2: Dünnbesetzter Index nach [SH99]

[13] die Form der Speicherung der internen Relation [SH99]

In der **Abbildung 2:** Dünnbesetzter Index nach [SH99] ist ein dünnbesetzter Index schematisch dargestellt. Für das Beispiel einer Relation, die eine Auftragsnummer enthält, würde nicht jede Auftragsnummer im Index enthalten sein. Unter der Annahme, dass jede Speicherseite der internen Relation zehn Einträge enthielte, würde ein dünnbesetzter Index nur jeden zehnten Zugriffsattributwert enthalten. Dieser Wert verweist jeweils auf den Seitenanführer[14], der folgende Wert auf den Seitenanführer der Folgeseite usw.
Der Vorteil eines dünnbesetzten Index liegt in erster Linie im reduzierten Speicherbedarf im Vergleich zu einem Index der alle Zugriffsattributwerte speichern muss.
Im Gegensatz dazu wird in einem dichtbesetzten Index für jeden Datensatz in der internen Relation auch ein Eintrag in der Indexdatei angelegt.

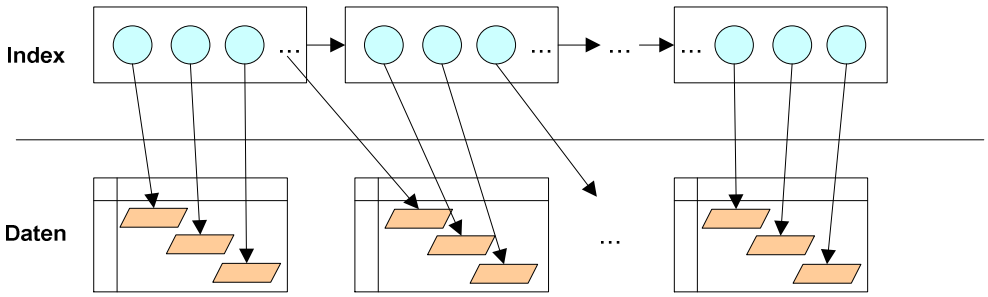


Abbildung 3: Dichtbesetzter Index nach [SH99]

Aus der schematischen Darstellung in **Abbildung 3:** Dichtbesetzter Index nach [SH99] geht der erhöhte Aufwand im Vergleich zu einem dünnbesetzten Index deutlich hervor. Demgegenüber steht der Vorteil der breiteren Anwendbarkeit eines dichtbesetzten Index. Im Beispiel der Auftragsnummer und des Kunden wird ein Index über den Kundennamen als dichtbesetzter Index realisiert werden, da ein solches Attribut in der internen Relation „Auftrag“ nicht sortiert vorliegt. Ein weiterer Vorteil ist, dass sich bestimmte Abfragen allein durch Zugriff auf den Index beantworten lassen. Dabei sind z. B. Existenztest-, Häufigkeits- und Min/Max-Anfragen zu nennen. [SH99], [St06]
Ein dichtbesetzter Index ist ebenso ein Kompromiss aus Speicherplatzbedarf und Zugriffsgeschwindigkeit wie ein dünnbesetzter Index. Das sequentielle Lesen einer Seite kostet Zeit und das Speichern eines Eintrages für jeden Datensatz kostet Speicherplatz. Demgegenüber ist nicht über die tatsächliche Nutzung des Index bekannt. Selbst wenn der Index als Ganzes benötigt wird, weil häuf nach z. B. einem Kunden selektiert wird, bleibt die

[14] den bezüglich der Ordnung ersten Wert auf einer Seite im Speicher [SH99]

Frage, ob es Kunden gibt, nach denen häufiger selektiert wird. Würde eine solche Häufigkeitsverteilung die Grundlage für die Organisationsform des Index bilden, können die Vorteile beider, dicht- und dünnbesetzter, Indexe kombiniert werden.

3.2.3. Geclusterter und nicht-geclusterter Index

Durch die Sortierung in der internen Relation ergibt sich ein weiteres Unterscheidungsmerkmal. Liegen die Einträge eines Index in der gleichen sortierten Form vor wie die Datensätze der internen Relation und werden neu einzufügende Sätze innerhalb des Index dicht nebeneinander abgelegt, handelt es sich um einen geclusterten Index. Wenn eine Relation „Auftrag“ nach der Auftragsnummer sortiert ist, so wird der Index über das Attribut Auftragsnummer in der Regel ein geclusterter Index sein. Jeder dünnbesetzte Index ist automatisch geclustert, aber die Clusterung setzt nicht notwendigerweise einen dünnbesetzten Index voraus.

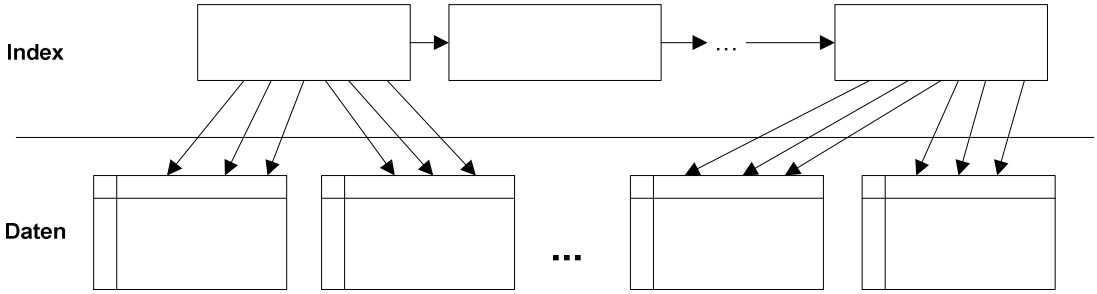


Abbildung 4: Geclusterter Index nach [SH99]

Ein nicht-gelusterter Index ist dagegen anders organsisiert als die interne Relation, die durch den Index verzeigert wird. Enthält die Beispielrelation „Auftrag“ auch ein Feld „Kundenname“ und ist die Relation über die Auftragsnummer (Primärschlüssel) sortiert, so wird ein Index über den Kundennamen immer ein nicht-geclusterter Sekundärindex sein. [SH99]

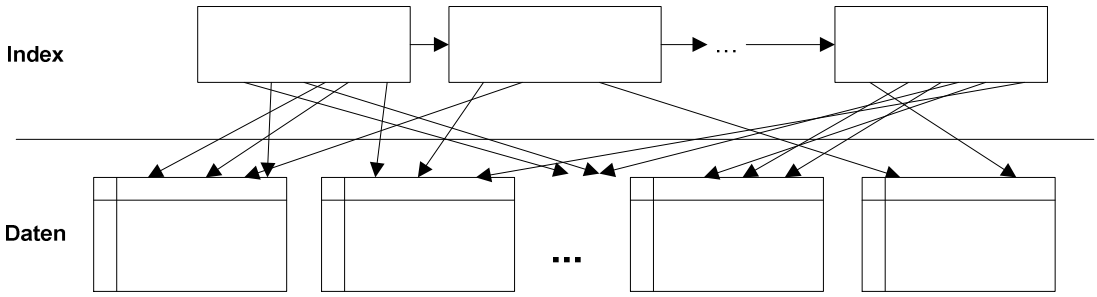


Abbildung 5: Nicht-geclusterter Index nach [SH99]

Durch einen geclusterten Index kann unter Ausnutzung der Organisationsform eine Abfrage nach einem bestimmten Datenbereich, z. B. Auftragsnummern von … bis, sehr effizient und schnell beantwortet werden. Ein geclusterter Index wird immer dann empfohlen, zu erwarten ist, dass eine entsprechend große Datenmenge nahezu ausschließlich mit Hilfe von Indexen gelesen wird. Als Beispiel wäre eine Schnittstellentabelle zu nennen, die einen großen Datenteil enthält und mit einer Sequenznummer gelesen wird, über die ein Primärindex gelegt wurde.

3.2.4. Weitere Klassifikationen

Ein-Attribut- und Mehr-Attribut-Index

Besteht ein Index aus nur einem Zugriffsattribut, so handelt es sich um einen Ein-Attribut-Index (engl. non-composite index). Umfasst ein Index mehr als ein Zugriffsattribut, wird dies Mehr-Atrribut-Index (engl. composite index) genannt. Würde anhand des obigen Beispiels ein Index über Kundenname und Standort gebildet, so ist es sinnvoll hier einen Mehr-Attribut-Index zu benutzen. Der Vorteil gegenüber zwei Ein-Attribut-Indexen offenbart sich bei einer Exact-Match-Selektion, etwa nach dem Kunden VW am Standort Salzgitter.

Statische und dynamische Zugriffsstrukturen

Statische Zugriffsstrukturen eignen sich nur für Daten, die nahezu keiner Veränderung unterliegen. Im typischen Beispiel einer Adresstransformation, z. B. der Kundennummer durch die Restdivision (*KundNr* mod *n*), ergeben sich *n* Speicherseiten, die die Kundennummern aufnehmen. Gewinnt die Firma viele Neukunden hinzu, reicht diese Struktur nicht mehr. Konzentriert sich eine Firma mit sehr vielen Kunden und einem entsprechend großen *n* auf einige wenige Stammkunden, wird eine derartige statische Struktur überflüssig und hindert den Zugriff mehr als sie ihm nützt.
Durch die Wahl einer Zugriffsstruktur, die sich an die Anzahl der Datensätze anpasst, wird das Problem gelöst, da eine optimale Verwaltung der Daten stets gegeben ist. Von den meisten DBMS werden dynamische Verfahren eingesetzt, da stets mit wachsenden oder schrumpfenden Datenbeständen gerechnet wird. [SH99]

3.3. Dateiorganisationsformen

Für die verschiedenen Anwendungsgebiete existiert eine Reihe von verschiedenen Speicherstrukturen. Je einfacher das Einfügen eines neuen Wertes in die Struktur ist, desto schwieriger wird es, diesen Wert zu einem späteren Zeitpunkt wieder aufzufinden – etwa bei Änderungsoperation auf diesem Wert. Die einfachste Klasse stellen die ungeordneten und sequentiellen Organisationsformen dar. Die Einträge werden gänzlich ohne Metainformation gespeichert und sind daher vor allem dort geeignet, wo Daten schnell und in großen Mengen gesichert werden müssen.

3.3.1. Heap

Bei einer Heap-Organisation einer Datei wird, wie aus der Übersetzung aus dem Englischen „Haufen“ bereits zu schließen ist, ein einzufügender Datensatz einfach an das Ende gestellt. Eine Heap-Datei bildet in der physischen Reihenfolge immer die zeitliche Reihenfolge des Einfügens der Datensätze wieder. Speichert das Data Dictionary zumindest die letzte Seite einer Datei, reicht einer Einfügeoperation ein einziger Seitenzugriff, falls noch genügend Speicherplatz vorhanden ist. Ansonsten müsste die Folgeseite geholt werden und der Eintrag im Data Dictionary aktualisiert werden.

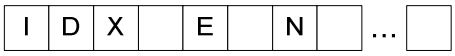


Abbildung 6: Heap-Struktur

Eine Suche bzw. das Löschen eines Datensatzes erfordert das sequentielle Lesen der Heap-Datei vom ersten Datensatz bis zum Treffer und bedeutet deshalb die maximale Komplexität *O(n)*. Im Gegensatz dazu steht das Einfügen mit dem bestmöglichen Aufwand *O(1)*. [SH99] Abhängig von der Zahl der Löschoperationen, muss eine Heap-Struktur reorganisiert werden, um die durch das Löschen entstehenden Lücken wieder aufzufüllen. Ansonsten nimmt eine Heap-Organisiation mehr Speicherplatz in Anspruch als sie eigentlich an Daten vorhält.

3.3.2. Sequentiell

Bei der sequentiellen Speicherung (sortierte Liste) werden die Datensätze sortiert abgelegt. Das Einfügen eines Datensatzes erfordert hier einen höheren Aufwand. Es muss zunächst die Stelle gesucht werden, in die der Datensatz gemäß der Sortierungsvorgabe eingefügt werden

kann. Dabei kann es passieren, dass nachfolgende Datensätze verschoben werden müssen. Solange es dabei nicht zu einem Überlauf auf einer Seite kommt, sind die Kosten dafür nicht sehr hoch. Es ist daher zu empfehlenswert, eine Seite nie komplett zu füllen, sondern etwas Speicherplatz für spätere Einfügeoperationen zu reservieren.

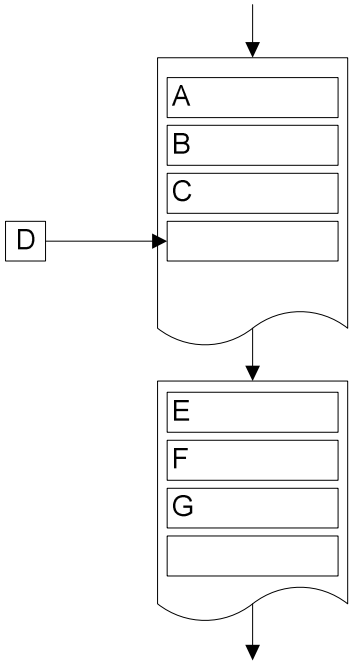


Abbildung 7: Sequentielle Speicherung

Das Suchen oder Löschen eines Datensatzes wird durch die sequentielle Speicherung beschleunigt – die Frage, wo mit der Suche begonnen werden soll, bleibt dennoch offen.
Eine mögliche Lösung wäre die binäre Suche, wobei zunächst das mittlere Element m auf der Speicherseite ausgewählt wird. Dieses wird mit dem gesuchten Element k verglichen. Ist er Schlüssel des gesuchte Datensatzes $k < m$, so wird mit einem rekursiven Algorithmus der vordere Bereich der Speicherseite ebenso zerlegt und durchsucht. Eine weitere Möglichkeit der Effizienzsteigerung beim Durchsuchen von sequentiellen Dateien bieten Verteilungsfunktionen, über die eine ungefähre Position auf einer Speicherseite interpoliert werden kann.

3.3.3. Index-sequentiell

Wird eine sequentielle Dateiorganisation mit einer separaten Indexdatei kombiniert, die auf Bereiche der sequentiellen Datei verweist, wird von index-sequentieller Dateiorgansisation gesprochen. Dabei stimmt die Sortierung der Datendatei mit der Sortierung der Indexdatei überein. Wird nur das jeweils erste Element auf einer Seite im Speicher und den Verweis auf

diese Seite für den Index herangezogen, entsteht ein dünnbesetzter und geclusterter Index, wie die folgende **Abbildung 8:** Index-sequentielle Speicherung illustriert:

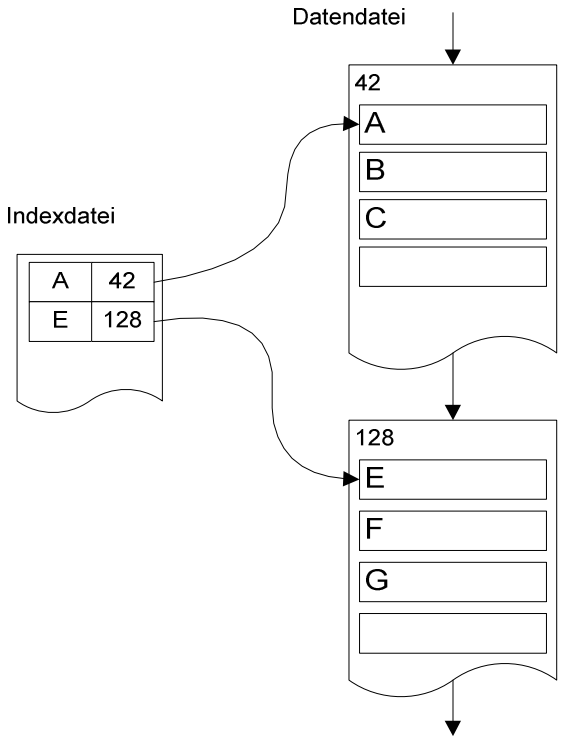


Abbildung 8: Index-sequentielle Speicherung, einstufig

Die Datendatei (oder Hauptdatei) ist nach einem Primärschlüssel, bestehend aus einer geeigneten Attributmenge, sortiert. Die Indexdatei enthält für jede Speicherseite, die von der Datendatei belegt wird, einen Eintrag. Diese Einträge sind ebenso sortiert wie die Hauptdatei.
Bei einer entsprechend großen Hauptdatei wird auch die Indexdatei linear wachsen und kann – wie die Hauptdatei – mehrere untereinander verkettete Seiten umfassen. Diese Seiten müssten ebenso sequentiell durchsucht werden, was den ursprünglichen Vorteil der index-sequentiellen Speicherung teilweise wieder aufheben kann.
Eine Lösung dieses Problems kann Indexieren der Indexdatei sein, das Erzeugen eines mehrstufigen Index. Dabei ist es erstrebenswert die Indexdatei der höchsten Stufe nicht über eine Speicherseite hinaus wachsen zu lassen, sondern in diesem Falle eine höhere Stufe einzuführen.
Eine Suchoperation (lookup) auf einer index-sequentiellen Datei zielt nun zunächst auf die Indexdatei. Das Prinzip für eine index-sequentielle Dateiorganisation mit einem mehrstufigen Index ist das gleiche: die Suchoperation wird nach dem gleichen Algorithmus für jede Stufe neu durchlaufen. Wird ein Datensatz zu dem Zugriffsattributwert k gesucht, muss der Index sequentiell nach einem Satz (k_1, s) mit $k_1 \leq k$ durchsucht werden. Wenn (k_1, s) der letzte

Eintrag auf der Indexseite ist, kann k nur auf der letzten Speicherseite s liegen oder der Attributwert k ist nicht in der Datendatei enthalten. Trifft der Algorithmus beim Durchsuchen des Index auf den Satz (k_2, s´) und gilt $k_2 > k$, dann ist der Datensatz zu k – falls er überhaupt vorhanden ist – auf der Seite s abgelegt. Dabei wird die Ausprägung des Zugriffsattributwertes k vom Indexeintrag (k_1, s) überdeckt. Die Suche auf der Seite s wird dann sequentiell vorgenommen. [SH99]

Vor einer Einfügeoperation (insert) muss zunächst eine geeignete Stelle in der Datei gesucht werden, damit zum einen die Sortierung erhalten bleibt und zum anderen muss genügend freier Speicherplatz auf der Seite s vorhanden sein. Reicht der Platzbedarf. wird der neue Datensatz an die entsprechende Stelle auf der Seite gespeichert. Handelt es sich dabei um die erste Stelle auf der Seite muss auch der Index aktualisiert werden und k würde k_1 verdrängen. Reicht der Speicherplatz auf der gefundenen Seite nicht aus, kann entweder eine neue Seite angelegt werden, auf die die Elemente der vollen Seite verteilt werden oder es wird eine Überlaufseite für die gefundene Seite s angelegt. Wird eine neue Seite angelegt, muss auch der Index entsprechend aktualisiert werden.

Auch die Löschoperation (delete) führt zunächst ein lookup aus und löscht dann den gefundenen Satz k auf der Seite s. Dies geschieht üblicherweise durch Setzen eines Löschbits. Ist der durch k identifizierte Satz der Seitenanführer der Seite s, muss der Index angepasst werden. [SH99]

Die Vorteile der index-sequentiellen Speicherung liegen im guten Antwortzeitverhalten bei exakten Abfragen und Bereichsabfragen, bei Pattern-Matching[15] und bei Abfragen nach Teilschlüsseln. Demgegenüber stehen die Nachteile einer statischen Organisation von Index und Hauptdatei. Wenn neu hinzukommende Sätze in Überlaufseiten gespeichert werden, werden die Antwortzeiten bei lookup-Operationen schlechter und zeitaufwändige Reorganisationsmaßnahmen müssen gestartet werden. Bei wachsenden Datenmengen kommt es auf der Indexdatei zu linear verknüpften Seiten, die auf der Suche nach einem Attributwert k sequentiell durchsucht werden müssen. Im entgegengesetzten Fall der stark schrumpfenden Datenmenge sinken die Seitenanzahlen in Index- und Hauptdatei nur langsam. Die Folge sind viele Seiten im Speicher, die nur ungenügend gefüllt sind, da nur komplett leere Seiten gelöscht werden. In der Praxis gibt es eine Vielzahl an Beispielen für plötzlich schrumpfende Datenbestände. So werden in vielen Unternehmen jeweils zum Monats- und besonders zum Jahresende verschiedene Bereinigungsläufe auf den Datenbeständen

[15] Musterabgleich, Abfragen mit Operatoren wie LIKE oder SIMILAR TO.

durchgeführt. Diese Probleme können nur von einer dynamischen Struktur ausgeglichen werden, die die Organisation der Daten anhand verschiedener Faktoren neu ausbalanciert.

3.3.4. Indexiert-nichtsequentiell

Da die Werte für Sekundärschlüssel in der internen Relation nicht sortiert sind, wird ein indexiert-nicht-sequentieller Zugriffspfad benötigt. Die Organisationsform von Haupt- und Indexdatei für einen Sekundärschlüssel unterscheiden sich, da für einen Sekundärschlüssel keine Sortierung der Hauptdatei erreicht werden kann. Die Indexdatei wird hingegen sehr wohl nach dem Attribut sortiert, jedoch kann es durch die Charakteristika eines Sekundärschlüssel (vgl. 3.1.3) dazu kommen, dass ein Attributwert mehrmals in die Indexdatei aufgenommen werden muss, weil er auch mehrfach in der Hauptdatei vorkommt. Als Alternative dazu, kann dem mehrfach auftretenden Attributwert auch eine Liste mit den Verweisen auf die entsprechenden Speicherseiten zugeordnet werden. Da es zu jedem Satz in der Hauptdatei auch mindestens einen Satz in der Indexdatei gibt, handelt es sich um einen dichtbesetzten Index, der aufgrund der unterschiedlichen Sortierung nicht geclustert sein kann.
Auf der Suche nach einem sekundären Zugriffsattribut müssen dann alle der Ausprägung k zugeordneten Seiten s sequentiell durchsucht werden. Da die Attribute mehrfach vorkommen können und unsortiert vorliegen, muss eine im Index verknüpfte Seite im Gegensatz zur index-sequentiellen Datei auch vollständig durchsucht werden. [SH99]

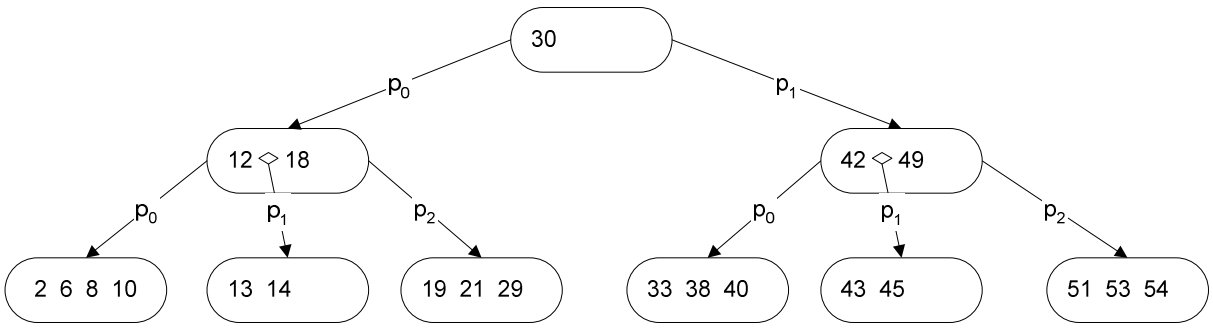
3.4. Baumstrukturen

Die Nachteile der bis hier genannten Verfahren zu Dateiorganisation äußern sich vor allem bei einem sich änderndem Datenvolumen. Wenn die Hauptdatei stark wächst oder stark schrumpft, kann eine statische Organisationsform, wie z. B. eine index-sequentielle Speicherung mit einer festgelegten Stufenzahl, die Zugriffe nicht mehr optimal abwickeln und muss reorganisiert werden. Die sogenannten Baumverfahren wurden entwickelt, um eine große Datenmenge effizient verwalten zu können und die Zugriffszeit auf die Daten so gering wie möglich zu halten.
Nach [SS01] ist ein Baum ein gerichteter Graph, der aus einer Menge von Knoten und Kanten besteht. Dabei verbinden die Kanten die Knoten von der Wurzel zu den Blättern gerichtet. Mit der Wurzel existiert genau ein spezieller Knoten, der keinen Vorgänger besitzt. Mit den

Blättern existieren ein oder mehrere Knoten, die keinen Nachfolger besitzen. Ein Baum ist stets zusammenhängend und frei von zyklischen Beziehungen.

3.4.1. B-Baum

Das Konzept eines B-Baums wurde im Jahre 1972 von R. Bayer und E. M. McCreight entwickelt. Es erwies sich als ideale Ordnungsstruktur für das relationale Datenmodell und wird heute in allen Datenbanksystemen als Grundtechnik eingesetzt. Ein B-Baum eignet sich prinzipiell für Primär- als auch für Sekundärindexe. [Eh05]
Im Unterschied zu einem binären Baum sind die Mehrwegbäume, zu denen auch der B-Baum gehört, in der Verzweigung breiter aufgestellt. Bei einem Binärbaum folgen einem Knoten immer genau zwei Knoten, bei einem B-Baum können grundsätzlich beliebige viele Folgeknoten definiert werden.
Ein Mehrwegbaum gelte aus völlig ausgeglichen, wenn alle Kanten zwischen den Knoten gleich lang sind und jeder Knoten die gleiche Anzahl an Einträgen beherbergt. Da bei Erfüllung dieser Forderung jedes Löschen, Einfügen oder Ändern eines Wertes umfangreiche Reorganisationsoperationen nach sich ziehen würde, legten Bayer und McCreight für den B-Baum fest, dass auf jeder Seite, außer der Wurzel[16], zwischen m und 2m Daten enthalten sein müssen. Die Wurzel muss mindestens m Elemente enthalten. Diese Konstante m wird als die Ordnung eines B-Baums bezeichnet. Ein Knoten ist entweder ein Blattknoten[17] oder hat $k + 1$ Nachfolger, wobei k die Zahl der im Baum abgelegten Elemente ist. Die letzte Forderung verlangt, dass alle Blattknoten auf der gleichen Stuf e zu liegen haben. Dadurch ist ein B-Baum immer nahe an der vollständigen Ausgeglichenheit und nutzt mindestens 50% des zur Verfügung stehenden Speicherplatzes. Die Komplexität eines Zugriffs liegt bei $O(\log_m(n))$.



[16] Der Knoten eines Baums, der keinen Vorgänger hat – der oberste Knoten.

[17] Die Knoten eines Baums, die keine Nachfolger haben – die untersten Knoten.

Abbildung 9: B-Baum der Ordnung 2

Die Suche nach einem Schlüssel k in einem B-Baum beginnt in der Wurzelseite. Wird der Schlüssel dort nicht gefunden, muss das kleinste Element k_i in diesem Knoten ausgewählt werden, das größer als k ist. Wird ein solches Element k_i gefunden, setzt sich die Suche in dem Kindknoten p_{i-1} fort. Gibt es kein solches Element, wird der Suchalgorithmus beim letzten Kindknoten p_n fortgesetzt. Eine erfolglose Suche endet erst in einem Blatt.
Beim Einfügen wird zunächst eine Suche durchgeführt, die zum Ergebnis „nicht gefunden“ kommen soll. Wenn nämlich ein Wert gefunden wird, muss er nicht erst eingefügt werden. Auf dem letzten Blatt der erfolglosen Suche wird der neue Wert eingefügt. Wenn das Blatt mit dem neuen Wert überfüllt werden würde, muss der Knoten aufgeteilt werden. Dazu wird die Mittelwert m des Knotens bestimmt. Die Elemente größer m werden in einen neuen Knoten neben dem bisherigen ausgelagert. Der Mittelwert m wandert eine Ebene höher in den Vaterknoten. Wenn der Vaterknoten jetzt überfüllt ist, wird der gleiche Vorgang wiederholt. Beim Löschen eines Wertes muss analog der Fall der Unterfüllung (underflow) behandelt werden. Dabei gibt es die Lösungsvarianten Ausgleichen und Verschmelzen.

3.4.2. B+-Baum

Während bei einem B-Baum auch die inneren Knoten Informationen tragen, ist ein B+-Baum ein hohler Baum. Die Einträge in den Knoten, die keine Blätter sind, enthalten keine Verweise auf die Daten, sie dienen lediglich dem Zugriffspfad. Sie müssen nicht einmal einer vorhandenem Schlüsselausprägung entsprechen, sondern lediglich einer lookup-Operation ermöglichen zu den Blattseiten zu gelangen, wo eine Entscheidung „found“ oder „not found“ getroffen werden kann. In den Blattknoten befindet sich dann das Tupel aus Schlüssel und Zeiger auf die Hauptdatei.

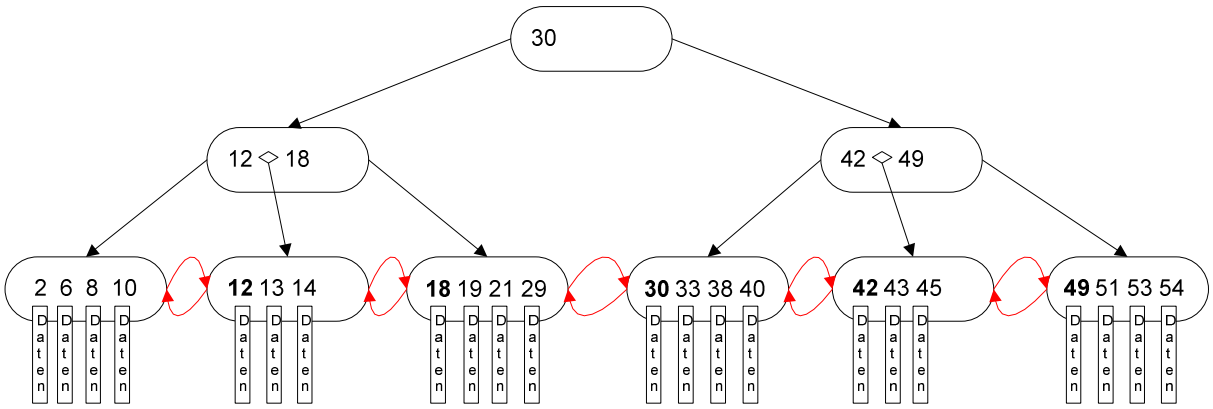


Abbildung 10: B+-Baum der Ordnung (1,2)

In **Abbildung 10** ist der B-Baum aus **Abbildung 9** als B+-Baum dargestellt. Die gegenüber dem B-Baum fett gedruckten Werte stellen die Änderungen dar, die einen B+-Baum u. a. ausmachen. Diese Zugriffsattributwerte sind in die Blattknoten gerutscht und dort mit den Daten verknüpft, obwohl sie schon auf den Indexseiten vorhanden sind.
Ein weiterer Vorteil sind die, im Bild rot markierten, lineare Verkettungen der Blätter. Dadurch ist es einem next-Operator möglich, eine Bereichssuche effizient abzuhandeln. Ein Sprung über die Ebenen eines Baums zurück zum Vaterknoten, um zum nächsten Wert zu gelangen, ist bei einem B+-Baum nicht notwendig. Die next-Operation kann sich am Ende eines Blattes zum nächsten hangeln.
Während die insert-Operation mit dem B-Baum vergleichbar ist, kann bei einer Löschoperation i. d. R. darauf verzichtet werden, Knoten miteinander zu verschmelzen oder auszugleichen. Wie bereits erwähnt, können die Zugriffsattribute in den Knoten stehen bleiben, selbst wenn der verknüpfte Wert aus einem Blattknoten entfernt wurde. Lediglich die Unterfüllung muss behandelt werden.

3.4.3. Weitere Baumvarianten

Mehr in der wissenschaftlichen Literatur oder anderen Anwendungsgebieten denn in tatsächlich eingesetzten DBMS finden sich weitere Baumvarianten, die für eine eng umrissene Aufgabenstellung geeignet sind bzw. einen Teilbereich verbessern. Beispielhaft seien hier B*-Bäume genannt, die sich von B-Bäumen lediglich durch das mögliche Aufteilen eines Knotens in drei Kindknoten bei einem Überlauf unterscheiden. Dadurch wird allerdings die Speicherplatzausnutzung eines B-Baum von nur ca. 50% auf 66% verbessert. Unter dem Oberbegriff der Digitalen Bäume werden Konzepte für Tries, Patricia-Tries, Suffix- und Präfix-Bäume zusammengefasst.
Aufgrund ihrer hohen Spezialisierung, der laut [SH99] teilweisen fehlenden Adaptivität an ungünstige Datenverteilungen und der derzeit fehlenden Relevanz für heutige DBMS werden weitere Baumverfahren an dieser Stelle nicht betrachtet.

3.5. Hash-Verfahren

Der Begriff „hash“ im Sinne der Informatik lässt sich am besten mit „streuen“ übersetzen. Mittels einer Hash-Funktion wird aus einer großen Menge an Quellwerten durch eine

Schlüsseltransformation eine kleinere Zielmenge erzeugt. Die Anzahl der dabei auftretenden Kollisionen soll dabei möglichst gering bleiben. Eine Kollision bezeichnet das Errechnen des gleichen Hash-Werts für unterschiedliche Originalwerte. Geeignete Hash-Funktionen transformieren daher die Quelldaten in eine Gleichverteilung und überprüfen das Ergebnis auf Kollisionen. Ab einer bestimmten Größe des Definitions- und des Wertebereichs sind Kollisionen unvermeidlich. Verbreitete Hash-Funktionen sind z. B. die Modulo-Division oder die multiplikative Methode, bei der das Produkt des Schlüssels mit einer Zahl (i. d. R. handelt es sich um eine lange irrationale Zahl) gebildet wird, von dem der ganzzahlige Anteil abgeschnitten wird, so dass ein Wertebereich von null bis eins belegt wird.
Bei statischen Hash-Verfahren ergeben sich die gleichen Nachteile wie bei den eingangs erwähnten statischen Dateiorganisationsformen. Bei stark wachsenden oder schrumpfenden Datenbeständen häufen sich die Kollisionen oder die Ausnutzung des Speicherplatzes sinkt unter einen akzeptablen Wert. Dennoch werden im Bereich der Datenbanken auch statische Hash-Verfahren eingesetzt.
Mit dynamischen Hash-Verfahren wird versucht die unzureichende Auslastung der Speicherseiten und die Bildung von Überlaufseiten zu verhindern oder zumindest einzuschränken. Das erweiterbare Hashen nach [FNP79] ist ein dynamisches Hash-Verfahren, bei dem eine zusätzliche Indexstruktur eingeführt wird. Jeder Eintrag in diesem Index zeigt genau auf ein Blatt. Auf ein Blatt können jedoch mehrere Indexeinträge verweisen. Tritt dann der Fall eines Überlaufs ein, wird der Index vergrößert und die überlaufende Seite wird gesplittet. Dabei werden die restlichen Blätter nun von doppelt so vielen Indexeinträgen verknüpft.
Trotz weiterer viel versprechender Ansätze existieren noch einige grundsätzliche Schwächen, wie zum Beispiel die Anfälligkeit beim Auftreten von nicht-gleichverteilten Hash-Werten oder der Auslagerung von Indexstrukturen auf Hintergrundspeicher. [SH99]

3.6. Mehrdimensionale Verfahren

Die bisher vorgestellten Techniken, wie zum Beispiel Baumverfahren haben Nachteile bei Daten, die nicht ohne weiteres nach einer linearen Ordnung sortiert werden können. Bei Hash-Verfahren können Schwierigkeiten bei Ungleichverteilung und Bereichsanfragen auftreten. Für Partial-Match-Abfragen oder bei geografischen Daten gibt es Zugriffsstrukturen, die als besser geeignet eingestuft und als „mehrdimensional“ bezeichnet werden.
Ein kd-Baum ist ein binärer Suchbaum mit einem k-dimensionalem Sortierschlüssel. In so einer Struktur wird eine Menge von mehrdimensionalen Datenpunkten verwaltet. Bei der

homogenen Variante enthält jeder Knoten des Baums die Ausprägung eines mehrdimensionalen Zugriffsattributes. Von jedem Knoten verweist jeweils ein Zeiger auf den linken und ein Zeiger auf den rechten Kindknoten. Die Werte der einzelnen Attribute werden abwechselnd pro Ebene zur Unterscheidung für die Folgeknoten benutzt. Bei einem zweidimensionalen Baum wird auf einer geraden Ebene nach Attribut 1 und auf einer ungeraden Ebene nach Attribut 2 entschieden. Bei der inhomogenen Variante werden die Schlüsselinformationen der entsprechenden Dimension je nach Ebene auf den inneren Knoten gespeichert, die auf weitere innere Knoten verweisen. Der Verweis auf die Datenblöcke der Hauptdatei erfolgt ausschließlich durch die Blattknoten. [VM01]

Die mehrdimensionalen Baumverfahren KDB-Baum von Robinson und KdB-Baum von Kuchen kombinieren das Prinzip eines kd-Baums mit einem B-Baum. Der verfeinerte Ansatz eines KdB-Baums stellt auf jeder Indexseite einen Teilbaum dar, bei dem hintereinander nach mehreren Attributen verzeigt wird. [SH99]

Weitere mehrdimensionale Ansätze sind Schlüsseltranformationsverfahren, wie z. B. das Bit Interleaving und das daraus abgeleitete mehrdimensionale Hashen MDH. Dieses Verfahren erzeugt zunächst mit einem linearen Hash Bit-Folgen, die in einem weiteren Schritt zyklisch via Bit-Interleaving abgearbeitet werden und so den Hash-Wert zu erzeugen. [SH99]

Eine der bekanntesten Technologien für mehrdimensionale Dateiorgansisationformen ist das Grid-File, auch Gitterdatei genannt. Es handelt sich bei einem Grid-File um eine mehrdimensionale Punktstruktur, die sich aus „Buckets“ und „Grid Directory“ zusammensetzt. Die Buckets sind Lagereinheiten mit einer festen Kapazität, die die Daten von 1-n Grid-Blöcken aufnehmen. Die Daten eines Grid-Blocks werden aber immer in ein und demselben Bucket gespeichert. Das Grid-Directory hat die Aufgabe die Buckets zu verwalten und vor allem auf die Ereignisse Overflow und Underflow eines Buckets entsprechend zu reagieren.

In der folgenden **Abbildung 11:** Grid-File (2-dimensional) wird ein Beispiel für ein Grid-File veranschaulicht. Gesucht seien alle Aufträge eines Unternehmen, die im Jahr 2007 angelegt wurden und deren Status mittlerweile „erledigt“ ist. Bei einer herkömmlichen Dateiorganisationform der Zugriffsstrukuren auf der Beispieltabelle AUFTRAG würden, je nach Optimizer, zunächst alle Aufträge herausgesucht, deren Zugriffsattribut „angelegt“ dem Wert „2007“ entspricht. Die gefundenen Sätze müssten daraufhin einzeln auf ihren Status untersucht werden, was höchstwahrscheinlich mehr als zwei Plattenzugriffe bedeuten würde. Eine der Zielsetzungen bei der Entwicklung von Grid-Files war die Minimierung der Zugriffe auf Sekundärspeicher, wie z. B. Festplatten, da diese Zugriffszeiten zu einem großen Anteil

von Geschwindigkeit und Anzahl der Plattenzugriffe abhängen („Two-disk-access principle“). Die Zugriffsattribute werden so transformiert, dass über das Grid-Directory die richtigen Speicherseiten („Buckets“) angesprochen werden, auf denen die Datensätze zu finden sind – oder es existieren keine passenden Datensätze.

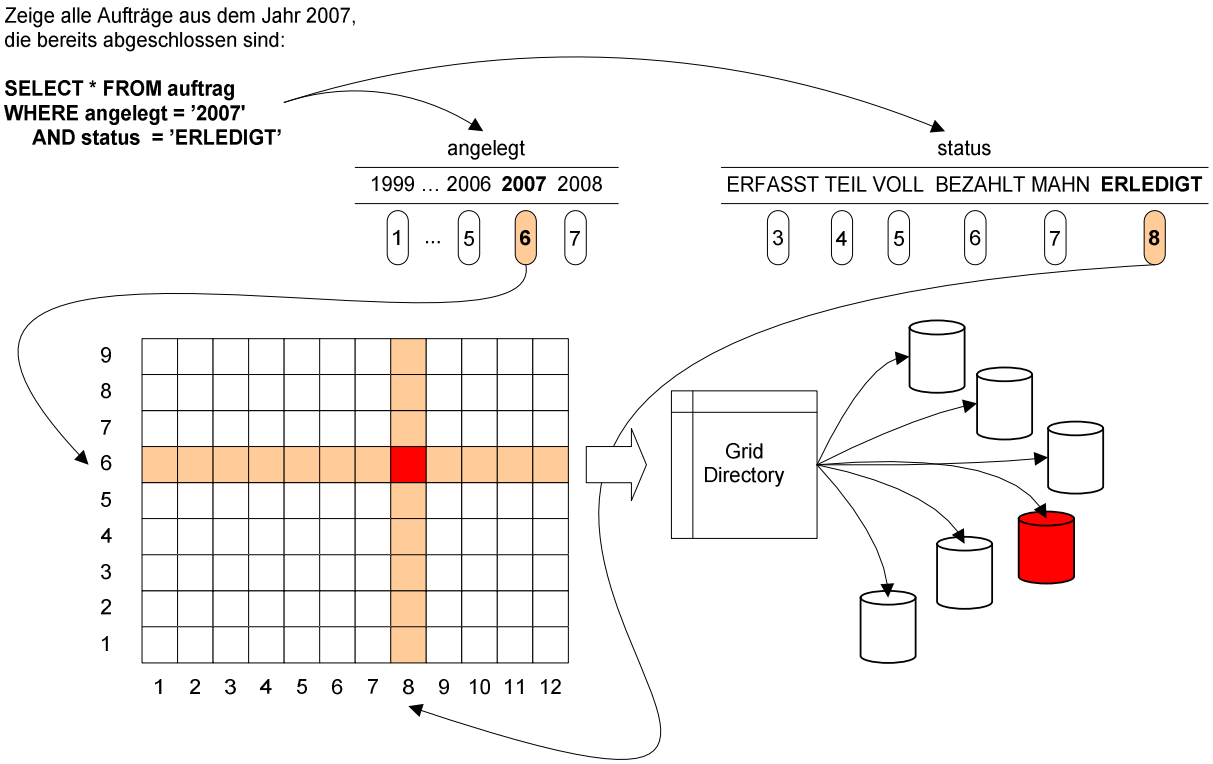


Abbildung 11: Grid-File (2-dimensional)

Ein Grid-File kann seine Stärken bei Range-Queries und Nachbarschafts-Anfragen ausspielen. Bei den Range-Queries werden die linearen Achsen genutzt, um die relevanten Bereiche zu finden. Bei Nachbarschaftsanfragen zu einem bestimmten Punkt wird einfach der Bucket eines gefundenen Punktes bzw. es werden die über die angrenzenden Bereiche auf den Achsen verknüpften Speicherseiten durchsucht. Bei ungleich verteilten oder korrelierten Daten weist das Grid-File Schwächen auf.

4. Automatisiertes Datenbanktuning

Die bisher vorgestellten Dateiorgansisationsformen für Zugriffsstrukturen haben in ihrem jeweiligen Anwendungsbereich zweifellos ihre Stärken. Doch sehen sich Anwendungsentwickler und Datenbankadministratoren eher aus dem Grund mit langen Laufzeiten der Datenbankabfragen konfrontiert, weil es überhaupt keine zu den Anfragen passenden Zugriffsstrukturen gibt. Das Datenbankmanagesystem hat gar keine andere Wahl, als direkt und sequentiell auf der Hauptdatei selbst zu operieren (FULL TABLE SCAN). Ein naiver Lösungsansatz wäre das Anlegen von Indexen für alle diejenigen Felder, die in einer Anwendung in einer WHERE-Bedingung auftauchen. Die Vorteile eines Index erkauft sich jedes Datenbanksystem jedoch mit Nachteilen auf anderem Gebiet. So muss für einen Index natürlich Speicherplatz vorgehalten werden. Das Anlegen eines Index verbraucht Rechenzeit, ebenso das Pflegen eines Index bei ändernden Transaktionen. Werden Datensätze eingefügt, geändert oder gelöscht, muss auch der Index dem Rechnung tragen und gemäß des ACID-Prinzips[18] geändert werden. Ansonsten könnte ein DBMS, das für den Zugriff einen Pfad über den Index gewählt hat, u. U. fehlerhafte Informationen liefern. Viele Zugriffsstrukturen (z. B. Bäume) bedürfen von Zeit zu Zeit einer Reorganisation, was ebenfalls Rechenzeit und Speicherplatz verbraucht.

Weiterhin ist anzumerken, dass bestenfalls die Anwendungsentwickler die logische Struktur und die Zusammenhänge in dem Maße beurteilen könnten, dass sinnhafte Entscheidungen für Zugriffsstrukturen getroffen werden können. Oft sind Anwendungsentwickler damit jedoch überfordert, da es ihnen an Einblick und Verständnis für die Mechanismen moderner DBMS mangelt.

Dabei muss insbesondere bedacht werden, dass sich die Menge der Daten in einem typischen Anwendungsfall über mehrere Jahre hinweg eher vergrößert als verkleinert. So betrug die Größe der Datenbank auf einer Großrechnerarchitektur für den Bereich „Kaltwalzwerk“ eines großen deutschen Stahlproduzenten bis zum Jahr 1999 nie mehr als 50 MB. Mit dem Einsatz von UNIX-Servern und dem DBMS *Oracle8i* wuchs die Größe der Datenbank auf fast 1 GB, ohne dass sich die Funktion der Applikation nennenswert geändert hätte. Lediglich durch das Aufheben der Beschränkungen (z. B. bei der Eingabe von Hinweistexten), Inkorporieren von

[18] ACID steht für Atomarität (atomicity), Konsistenz (consistency), Isoliertheit (isolation) und Dauerhaftigkeit (durability), die Grundanforderungen an eine Datenbanktransaktion.

Datenbeständen aus VSAM[19]-Dateien und durch das Erweitern des zeitlichen Horizonts für Archivierungsläufe vervielfachte sich die Datenmenge. Nach der Migration auf das Betriebssystem Linux und die Datenbankversion *Oracle 9i* hat sich diese Zahl seit 2005 nochmals verzehnfacht.

Moderne DBMS bieten eine Vielzahl an Stellrädchen, die dafür sorgen sollen, dass die Performance der Datenbank auch bei steigenden Kennzahlen stets gleich hoch bleibt. Für nahezu jedes große DBMS existieren Werkzeuge von Fremdanbietern, die den Datenbankadministrator bei seinen Tuning-Maßnahmen unterstützen sollen. Die Kosten für die Lizenzen des DBMS und der eventuell anzuschaffenden Zusatzsoftware werden von den Beträgen bei weitem übertroffen, die ein Unternehmen ausgeben muss, um die Mitarbeiter zu bezahlen, die die „schwarze Kunst des Datenbanktunings“ [WMH02] beherrschen.

Die Höhe dieser Total Cost of Ownership (TCO)[20], die steigende Komplexität von mehrschichtigen Softwarearchitekturen und die Vielzahl der bereits heute in DBMS existierenden Parameter lassen den Bedarf nach einem automatischen Tuning entstehen. [CW05]

Eine beliebte Lösungsstrategie war die Beschaffung und der Einsatz neuerer und besserer Hardware (KIWI-Ansatz[21]). Bei einer entsprechenden Ausgangslage ist dies durchaus Erfolg versprechend, doch zum einen erstrecken sich die üblichen Abschreibungszeiträume für Hardware über mindestens drei Jahre und zum anderen ist der tatsächliche Engpass beim „langsamen Datenbankzugriff“ nicht bekannt. So kann es sein, dass die Hardware durchaus schnell genug ist. Der Datenbank fehlen nur die geeigneten Zugriffsstrukturen, um das Potential der Hardware auch auszunutzen. [CW05]

Eine Daumenregel reicht für eine entsprechend große und komplexe Datenbank nicht mehr aus. Vor etwas zehn Jahren konnte ein Datenbankadministrator die zehn häufigsten Statements analysieren und den Anwendungsentwicklern einige entscheidende Hinweise geben. In Zeiten von dynamischem SQL und Cost-Based-Optimizern ist das Verbesserungspotential durchaus noch vorhanden, aber es ist schwieriger aufzuspüren und bringt nicht in jedem Fall einen durchschlagenden Erfolg. Die einfachen Lösungen reichen nicht mehr aus. [CW05]

---

[19] VSAM steht für Virtual Storage Access Method, eine von IBM entwickelte Zugriffsmethode für Dateien

[20] Unter TCO wird die Gesamtheit aller Kosten von der Anschaffung über den Erhalt bis zur Abschaffung über eine bestimmte Laufzeit zusammengefasst.

[21] Kill It With Iron

In einigen Fällen lässt sich ein Performance-Engpass sicherlich durch die Methode „Versuch und Irrtum“ (engl. trial and error) ermitteln und beheben, bei produktionskritischen Anwendungen dürfte sich diese Arbeitsweise eher als Kündigungsgrund für den irrenden Mitarbeiter erweisen. Durch den Einsatz geeigneter Simulationswerkzeuge kann jedoch auch die Methode„trial and error“ Problemlösungen eröffnen. [WMH02]
Um ein automatisches Tuning auf eine solide Grundlage zu stellen, sind die Auswirkungen verschiedener, teilweise mit wechselseitigen Zielkonflikten behafteter Einstellungsmöglichkeiten zu prüfen. Die Auslastung der Datenbank wird in aller Regel einer zeitlichen Variabilität unterliegen. So sind Spitzenzeiten morgens und nachmittags, von Montag bis Freitag, zum Monatsende, zum Jahresende, generell ansteigende Gesamtlast und viele weitere Lastprofile bekannt, so dass eine Momentaufnahme nicht weiter hilft. Vielmehr muss die Datenbank über einen signifikant längeren Zeitraum vermessen werden – Idealerweise permanent im Hintergrund. Weiterhin ist die Frage zu stellen, was unter Last verstanden werden soll und was gemessen werden muss. Es existieren mehrere Kennzahlen, um die Last einer Datenbank zu messen, wie z. B. der CPU- oder Hauptspeicherverbrauch, die Antwortzeit, der Durchsatz an Daten. Diese Kennzahlen müssen mittels geeigneter mathematischer Methoden (Durchschnitt, Verteilung, Ausreißer) bewertet werden. Ein weiterer wichtiger Gesichtspunkt sind die Parameter einer Datenbank, die einen großen Einfluss auf die Performance der Datenbank ausüben. [CW05]

4.1. Fünf Paradigmen des Auto-Tuning

Chaudhuri und Weikum formulierten in [CW05] die Grundlagen des Auto-Tunings für Datenbanksysteme und fassten diese in den so genannten fünf Paradigmen zusammen.

4.1.1. Tradeoff Elimination

Am Beispiel der beiden Standardtechniken für das Cache-Management illustrieren Chaudhuri und Weikum in [CW05] ein klassisches Self-Tuning-Problem. Während LRU[22] die Seiten zuerst verwirft, deren letzter Zugriff am längsten zurück liegt, wird bei LFU[23] die Seite aus dem Cache verdrängt, auf die am wenigstens zugegriffen wird. Somit beinhaltet die Alternative LRU eine zeitliche Komponente, aber keine Information über die absolute Zahl der Zugriffe, während LFU das genaue Gegenteil darstellt. Beide Prinzipien können ihre

[22] least recently used

[23] least frequently used

Vorteile auf gegensätzlichen Bereichen ausspielen. So wird LFU bei statischen Zugriffswahrscheinlichkeiten bessere Ergebnisse liefern und demgegenüber liegt LRU bei solchen Anwendungsprofilen vorn, bei denen der letzte Zugriff auf ein Objekt auch die Wahrscheinlichkeit erhöht, dass an dieser Stelle einer erneuter Zugriff erfolgen wird. Die Wahl eines Algorithmus aus diesen beiden Alternativen stellt eine „Entweder-Oder“ (trade off) Entscheidung zwischen Zugriffshäufigkeit und Zugriffsaktualität dar, bis ein Kompromiss gefunden wird, der beide Anforderung „adäquat“ bedient.
Der Lösungsvorschlag für viele Probleme ist daher die Suche nach einem Kompromiss, der sich in einer gegebenen Umgebung als akzeptabel erweist, statt eine Lösung zu entwickeln, die durch das Ermitteln des aktuellen Zustands zwischen den verschiedenen Modellen hin- und her schaltet. Überdies wäre eine solche Lösung äußerst fehleranfällig. [Li06]

4.1.2. Static Optimization

Techniken zur statischen Optimierung werden seit einigen Jahren im Zusammenhang mit Autonomem Computing für Kapazitätsplanung, physischer Systemaufbau und Konfiguration von Ressourcen eingesetzt. Dabei werden mathematische Optimierungen ebenso eingesetzt wie “Was-wäre-wenn”-Analysen und KI-verwandte Algorithmen wie Greedy, Zufallssuche (Monte-Carlo-Simulation), Branch-and-Bound, Genetische Algorithmen, Neuronale Netze, Simulierte Abkühlung und andere [Li06]. Das Problem wird von [CW05] mit „Real Life Queries are Complex!“ treffend beschrieben. Ausgehend von einem Arbeitsvorrat an Abfragen und Änderungen wird unter restriktiven Rahmenbedingungen (z. B. Speicherplatz) versucht, das Problem durch Auswahl der Konfiguration (Indexe, Materialized Views oder Partitionen) zu lösen, die die geringsten Kosten[24] verspricht.
Über Manipulationen der Metadaten eines DBMS kann der Optimizer veranlasst werden, verschiedene, derzeit nicht existierende, physische Designschemata („What-If-Indexe“) vor der Ausführung der Query zu berücksichtigen, um daraus einen Rückschluss auf die optimale Konfiguration zu ziehen. Nach Ermittlung der jeweils besten zu erstellenden Konfiguration für eine Query müssen diese lokalen Optima für eine Gruppe von Abfragen zusammengefasst werden, um den besten Kandidaten insgesamt zu ermitteln („Merging“). [CW05]

[24] die vom Optimizer jeweils geschätzten Ausführungskosten

4.1.3. Stochastic Prediction

Bei einer stochastischen Optimierung wird davon ausgegangen, dass die Auslastung eines Datenbanksystems einer statistischen Fluktuation unterliegt. Für eine Kapazitätsplanung (z. B. Aufrüstung des Hauptspeichers) oder eine optimale Systemkonfiguration (z. B. Größes des Cache) bieten stochastische Modelle Vorhersagen für eine zukünftig mögliche Auslastung. Dabei sind sogar grobe Aussagen und Modelle besser als das Fehlen jeglichen Modells. Diese Modelle können um Daten aus Simulations- und Messläufen ergänzt werden, um eine genauere Aussage treffen zu können. [CW05]

4.1.4. Online Optimization

Bei der Optimierung zur Laufzeit steht für den Algorithmus eine Reihe von Ereignissen im Vordergrund, die nacheinander abgearbeitet werden müssen. Die zukünftigen Ereignisse sind dabei unbekannt. Die Qualität wird dabei an einem Offline-Algorithmus gemessen, dem der komplette Umfang des Inputs bekannt ist, der so genannten „competitive ratio“.
Für Online-Optimization mit Hilfe von Histogrammen sind zwei Szenarien zu unterscheiden. Zum einen die „Online incremental maintenance“, bei der auch ändernde Operationen (Einfügen, Ändern, Löschen) betrachtet werden und die „online incremental correction“, bei der Daten über die Queries gesammelt werden, ohne dass es zu Änderungen kommt. Für das erste Szenario muss die Frage beantwortet werden, wie das komplette Neuerstellen des Histogramms nach dem Einfügen, Ändern oder Löschen der Daten umgangen werden kann. Im zweiten Szenario werden die Histogramme lediglich durch das Auswerten der Anfragen auf die Relation geschrieben. Dabei geht es um die Frage, wie das Histogramm verändert werden muss, wenn zusätzliche Erkenntnisse vorliegen. Da die Daten selbst in der Relation für die Erstellung eines Histogramms nicht interessant sind, wird von Gleichverteilung und Unabhängigkeit ausgegangen. Mit jeder Query wird das Histogramm angepasst und verfeinert. [CW05]

4.1.5. Feedback Control Loop

Diese Technik basiert auf der Kontrolltheorie, nach der ein bestimmter Soll-Zustand definiert wird, der aus einem gegebenen Ist-Zustand heraus möglich schnell und effizient über einen sich ständig wiederholenden Regelkreis aus Beobachtung, Vorhersage und Reaktion erreicht werden soll. Dabei sind restriktiv wirkende Nebenbedingungen zu beachten. Im COMFORT-

Projekt wird durch [WMH02] und [CW05] unter anderem ein Mechanismus zur Kontrolle von Datenbanktransaktionen und deren Locking-Verhalten beschrieben, dessen Ziel eine optimale Parallelität der Transaktionen ist. Dabei gilt es zu verhindern, dass ein „Trashing[25]“ genanntes Verhalten auftritt, bei dem sich die Transaktionen gegenseitig durch Sperren und Waits behindern.

Ein von [WMH02] betrachtetes Stellrädchen ist der „Multiprogramming Level“ (MPL). Das ist die Zahl mit der die Höchstgrenze für die Anzahl der gleichzeitig abzuarbeitenden Transaktionen (TA) in einem Datenbanksystem festgelegt wird. Wird diese Zahl zu hoch gewählt, vergrößert sich das Risiko des Auftretens von Thrashing. Bei einer zu niedrigen Einstellung warten unnötig viele Transaktionen auf den Beginn ihrer Abarbeitung und erhöhen gleichzeitig die Antwortzeit. Da eine im Workload gegebene Menge von Transaktion in der Realität nicht gleichförmig sein wird, wäre die Unterteilung in Transaktionsklassen wünschenswert, für die jeweils ein eigener MPL einzustellen ist. Eine solche Option wird von [WMH02] allerdings als Tuning-Albtraum bezeichnet. Stattdessen wird der MPL in einer Feedback Control Loop (FCL) kurzfristig an den Workload angepasst. Die Beobachtungsphase wird durch eine einfach zu berechnende Konfliktrate gestaltet, bei der einfach die Locks gezählt werden:

$$Konfliktrate = \frac{Anzahl\ der\ Locks\ aller\ Transaktionen}{Anzahl\ der\ Locks\ der\ nicht-blockierten\ Transaktionen}$$

Experimentell wurde ein Wert von 1,3 ermittelt, der bereits eine sehr hohe Thrashing-Gefahr indiziert. Bei der Beobachtung wird für jede ankommende Transaktion eine Wahrscheinlichkeit berechnet, nach der diese Transaktion blockiert werden könnte und basierend darauf die neue Konfliktrate. Die Reaktionsphase verfügt mit der Zugangs-, Abbruch- und Neustartkontrolle über drei Mittel zur Steuerung der MPL von Transaktionen. Abgebrochene Transaktionen werden in eine Restart-Queue eingefügt und mit bestimmten Regeln belegt, die bei einem erneuten Start zu beachten sind, beispielsweise die ID einer konkurrierenden Transaktion, die das System verlassen haben muss, bevor ein Restart erfolgen kann. [WMH02]

[25] engl. Dreschen, von „leeres Stroh dreschen“, zunehmender Ressourcenverbrauch bei abnehmender Abarbeitungsgeschwindigkeit, z. B. Speichermanagement, das viele Seitenfehler erhält und dadurch ständig Speicherseiten auslagern und nachladen muss.

4.2. MAPE

Der von der IBM entwickelte MAPE-Ansatz weist Ähnlichkeiten mit der FCL von [WMH02] auf. MAPE steht dabei für Monitor, Analyze, Plan und Execute und bezeichnet die vier Phasen des Regelkreises. Nach [Te04] werden dabei jedoch nicht nur Parameter als einstellbare Kenngröße für Software betrachtet, sondern MAPE soll jedes diskrete Element in einem Workflow, auch Hardware, überwachen und regeln.

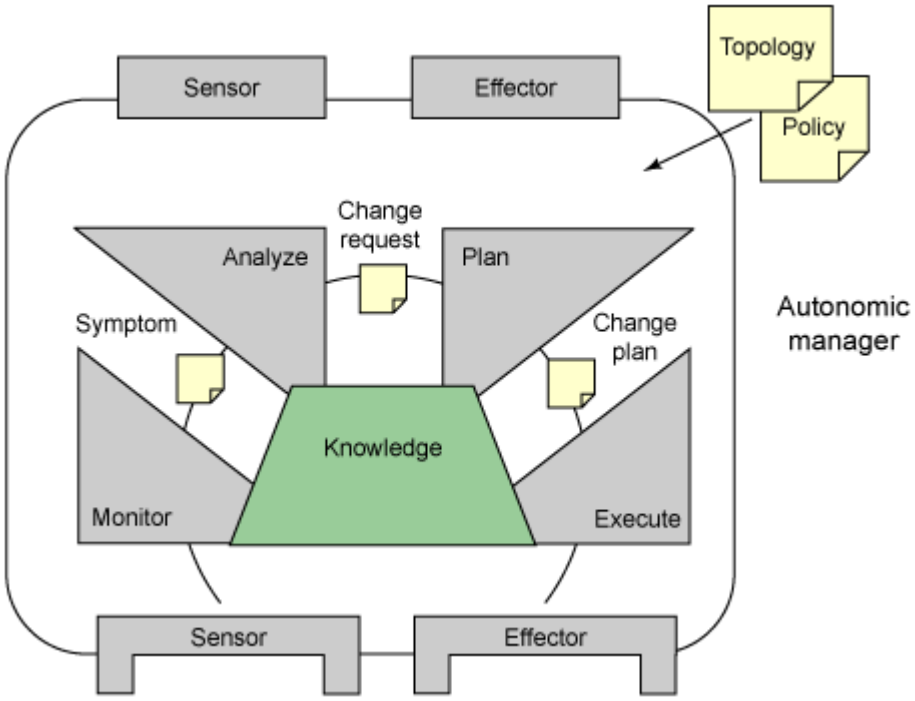


Abbildung 12: MAPE-Regelkreis [Mi05]

In der Phase 1 (Monitor) werden kontinuierlich Daten gesammelt und gefiltert. Dies entspricht in etwa der Observation-Phase der FCL. Wird dabei ein Symptom erkannt, das einer genaueren Analyse bedarf, erfolgt eine Übergabe des Objekts an die zweite Phase (Analyze). In dieser Stufe wird entschieden, ob ein Parameter einer Anpassung bedarf. Ist dies der Fall, wird ein „Change Request“ generiert und an die Phase 3 (Plan) übergeben. Hier wird entschieden, welche Operationen durchzuführen sind. Dies kann z. B. ein Abbrechen einer Transaktion sein oder eine ganze Reihe von verketteten Operationen. Der dabei erzeugte Change Plan wird an die Phase 4 (Execute) übergeben, die wiederum autonom die geplanten Änderungen durchführt. In jeder Phase wird dabei auf eine Wissensbasis (Knowledge) zurückgegriffen, die Informationen über Symptome, Lösungen und Rahmenbedingungen bereitstellt. Diese Wissensbasis kann bei Bedarf in jeder Phase aktualisiert werden.

4.3. Partial Indexes

In [St89] schlug Michael Stonebraker die Erstellung von Indexen vor, die nicht eine gesamte Spalte (oder mehrere Spalten) einer Relation zum Ziel haben, sondern eine durch eine WHERE-Bedingung qualifizierbare Untermenge. Dazu wird das bekannte CREATE INDEX-Kommando erweitert:

```
CREATE INDEX <index_name>
ON <table_name>
(column_name)
WHERE <column_name> <operator> <constant>
```

Damit deckt ein partieller Index einen genau bestimmten Wertebereich ab. Stonebraker schloss die Erstellung von partiellen Indexen über zusätzliche Verbundoperationen aus. Denkbar ist zumindest auch die Form:

```
CREATE INDEX <index_name>
ON <table_name>
(column_name)
WHERE <column_name> <operator> <sub_select>
```

Der Einsatz von partiellen Indexen erfordert einige Anpassungen am Optimizer. Wird eine Anfrage an das DBMS gerichtet, wählt der Optimizer – im besten Falle – einen optimalen Zugriffspfad aus. Nachdem ein Index für eine Abfrage als benutzbar aufgrund des Spaltennamens qualifiziert wurde, muss nun eine zusätzliche Prüfung auf den im Index enthaltenen Wertebereich erfolgen.
In [St89] werden mehrere Einsatzgebiete für partielle Indexe beschrieben. So können Spalten, die mit unbekannten (NULL) oder Default-Werte belegt sind, vom Index ausgeschlossen werden. Ausgehend von Spalten, deren Wertebereich ungleich verteilt ist, sind Aufgabenstellungen denkbar, die einige wenige Werte über einer bestimmten Schwelle mit einer höheren Abfragewahrscheinlichkeit klassifizieren als die Masse der restlichen Ausprägungen. So wird sich das Controlling eines Unternehmens auf diejenigen (wenigen) Ausgaben konzentrieren, deren Betrag eine bestimmte Höhe überschreitet. Eine Anwendung für das Controlling eines Betriebes würde also von einem partiellen Index profitieren, da dabei erstens weniger Speicherplatz verbraucht wird, als bei voller Indexierung der Spalte

„Betrag“ und zweitens wesentlich effektiver auf die wenigen hohen Werte zugegriffen werden kann, wenn die vielen niedrigen Beträge nicht den Index belasten.
Ein drittes Einsatzgebiet sind Felder mit einem begrenzten diskriminierenden Charakter, typischerweise J/N-Felder oder Returncodes. Hier können nur die Werte indexiert werden, die vom Normalzustand (‚J’ oder ‚N’ bzw. 0 oder 1) abweichen.
Durch die aktuelle Geschwindigkeit von Datenbanken auf heutiger Hardware hat die Bedeutung von partiellen Indexen bei einem inkrementellen Indexaufbau abgenommen. Dabei wird der Index anhand eines Primärschlüssels aufgeteilt und schrittweise erzeugt, um die negativen Seiteneffekte des Indexaufbaus („lock table“) zu umgehen. Dennoch gibt auch heute noch (z. B. in Enterprise Ressource Planning (ERP) Systemen) Anwendungen, auf deren mehrere Millionen Einträge umfassenden Tabellen nahezu ständig DML-Operation durchgeführt werden. Indexe auf diesen Tabellen werden üblicherweise bei geplanten Unterbrechungen erzeugt bzw. reorganisiert.
Mit partiellen Indexen eröffnet sich ebenso Möglichkeiten für das query-driven index tuning. Dabei analysiert die Datenbank die Abfragen auf den Spalten einer Tabelle und erzeugt bei der Erkennung von Häufigkeitsmustern einen partiellen Index auf den betreffenden Tupeln. Wird dieser Ansatz mit einer FCL kombiniert, die den Erfolg einer solchen Zugriffsstruktur misst und gegebenenfalls immer wieder an die Erfordernisse anpasst, ergibt sich ein interessantes Feld für das automatische Datenbanktuning [SSG05].
Viele Abfragen an Datenbanksysteme sind durch eine Anwendung vorgefertigt und werden vom Anwender zu bestimmten Zeiten ausgelöst. Beispiele sind der Meister, der nach Ende seiner Schicht die Leistung (produzierte Menge in einem bestimmten Zeitraum) kontrollieren will oder ein Produktionsplaner, der sich jeden Morgen über die zu verplanenden Mengen informiert. Oft sind solche Abfragen in einem System auf Knopfdruck verfügbar. Durch die Erzeugung bzw. den Rebuild eines partiellen Index, der genau auf diese Abfragen zugeschnitten ist, lässt sich die Erzeugung von Systemlast durch diese Abfragen zeitlich entzerren. Die Abfrage wurde durch den partiellen Index im Voraus berechnet und belastet das System nicht zusätzlich zu Zeiten, zu denen ohnehin eine hohe Last herrscht (z: B. morgens). [St89]
Im DBMS PostgreSQL (vgl. 2.2.3) können partielle Indexe erstellt und genutzt werden.

4.4. QUIET

Das von Sattler, Geist und Schallehn 2003 in [SGS03] und [SSG04] vorgestellte QUIET Tool ist eine in Java programmierte Middleware, die auf dem DB2 DBMS aufsetzt und im

Gegensatz zum IBM DB2 Index Advisor online die Abfragen sammelt und ein anfragegetriebenes (query-driven) Index Tuning bereit stellt. Dazu werden die Abfragen zunächst an das QUIET-System gerichtet und von dort aus via JDBC an DB2 weiter gegeben. QUIET analysiert die Anfrage, sammelt statistische Informationen und ermittelt für verschiedene Indexkonfigurationen die Kosten- und Profitinformationen mit Hilfe des DB2 Optimizers. Die vom System bevorzugte Indexkonfiguration kann noch während der Ausführung der Abfrage in einem Speicherplatz limitierten Pool („on the fly“) materialisiert oder einem DBA zur Sichtung und Entscheidung vorgelegt werden. [SGS03]
QUIET wurde durch die im folgenden Absatz beschriebenen „Soft Indexe“ weiterentwickelt, auch um eine engere Integration mit dem DBMS zu erreichen, da der Ansatz als externe Komponente mit JDBC-Interface einen Zeitverlust bedeutet.

4.5. Soft Indexe

Die in einem DBMS vom Anwendungsentwickler oder vom Datenbankadministrator angelegten Indexe bezeichnen Lühring, Sattler, Schallehn, und Schmidt in [LSS07] als *Hard Indexe* und grenzen sie so von denen als *Soft Indexe* bezeichneten Zugriffsstrukturen ab, die autonom vom DBMS angelegt und verwaltet werden sollen.
Die Soft Indexe erhalten drei zusätzliche Merkmale. Dazu zählen „managed“ mit der Aussage, ob der Index autonom vom System verwaltet wird, „virtual“ für den Status der Materialisierung und „deferred“ für einen leer angelegten Index.
Wie in einem Regelkreis (z. B. FCL) wird das Tuning der Soft Indexe in drei Phasen vorgenommen. In der ersten Phase (**Monitoring**) wird eine Anfrage zunächst auf konventionelle Art mit Hilfe des Optimizers mit Kosten bewertet. Anschließend wird die Anfrage analysiert und versucht verschiedene mögliche Indexkandidaten zu extrahieren. Für jeden dieser Kandidaten wird ein virtueller Index angelegt und die Kosten der Abfrage werden neu ermittelt. Im Vergleich zu den ursprünglich ermittelten Kosten ohne Soft Index ergeben sich für die Indexkandidaten Einzelprofite, die unter Berücksichtigung der Größe eines Index und der Behandlung von Updates zu einem Gesamtprofit *benefit(I)* für eine Indexkonfiguration berechnet werden. Da diese Berechnungen die Antwortzeit nachteilig beeinflussen, werden Anfragen in Epochen gesammelt. Dadurch ist es auch möglich eine Priorisierung über die Zeit zu realisieren, indem z. B. zurückliegende Epochen gedämpft werden.
In der zweiten Phase (**Decision**) wird unter den Nebenbedingungen des aktuellen Workloads und des begrenzten Speicherplatzes das Rucksackproblem der optimalen Indexkonfiguration

mit dem Greedy-Algorithmus gelöst. Die ermittelte Konfiguration wird jedoch nur realisiert, wenn sie besser als die aktuelle Konfiguration zuzüglich eines empirisch ermittelten Schwellwertes ist (Vermeidung von Thrashing).
In der dritten und letzten Phase (**Action**) werden die Indexe materialisiert. Bei der Wahl des Zeitpunktes muss zwischen geringster Verzögerung der Abfrageausführung und Aktualität gewichtet werden. Der in [LSS07] vorgeschlagene Kompromiss ist das Erzeugen von Deferred Indexen. Das heißt, die erzeugten Indexe sind zunächst leer und werden während einer Anfrage mit einem IndexBuildScan erzeugt. Über einen SwitchPlan-Operator kann sogar während der Abarbeitung einer Verbundanfrage von Table-Scan auf Index-Scan umgeschaltet werden. Laut Schmidt in [Sc06] konnte dieser Operator Nested-Loops sogar schneller bearbeiten als eine Anfrage mit vorher erstelltem Index. Der Unterschied fällt bei schnellerer Hardware sogar noch größer zugunsten des SwitchPlan aus.
Trotz einiger Schwierigkeiten bei der Verlässlichkeit des PostgreSQL-Optimierers präsentieren sich Soft Indexe als probates Mittel für ein autonomes Tuning, insbesondere bei variablem Workload. Eine Kombination mit den Partial Indexes von Stonebraker aus [St89] erscheint vorteilhaft, ebenso eine Verflechtung mit selbst-tunenden Indexen aus [SSG05] oder dieser Arbeit.

4.6. State of the Art

Die Hersteller von DBMS haben den Bedarf an Werkzeugen zur Unterstützung des Systemmanagements erkannt und bieten entsprechende Erweiterungen an. Die Konfiguration einer Datenbank, insbesondere das Setzen von Parametern, wird bei allen großen Datenbanken durch so genannte Assistenten oder Advisors (Ratgeber) getragen. Auch für den Entwurf, insbesondere von Indexen, stehen Tools in recht unterschiedlicher Tiefe bereit. An dieser Stelle sei der Microsoft Database Tuning Advisor hervorzuheben, der von einem Workload unter Nutzung des Optimizer-Kostenmodells verschiedene Indexe als What-If-Kandidaten untersucht und eine Empfehlung als Ergebnis zurückgibt.
Bei allen Fortschritten auf diesem Gebiet bleibt derzeit noch der Mensch, besser gesagt, der Datenbankadministrator, als letzte Instanz übrig, um eine Entscheidung zu treffen, ob und welche Tuningmaßnahme jeweils durchgeführt wird. Keines dieser Werkzeuge versieht selbständig im Hintergrund seinen Dienst. Echtes Auto-Tuning muss mehr leisten. Unter den vom Menschen gesetzten Rahmenbedingungen sind durch das DBMS nicht nur einzelne Anfragen, sondern auch die Speicher- und Zugriffsstruktur zu optimieren.

5. Last-balancierte Indexstrukturen

Das Problem vieler Performanceengpässe beim Einsatz von klassischen DBMS ist die zunehmende Schwierigkeit Abfragen vorherzusehen. In Zeiten von klassischem Batch-SQL, zum Beispiel SQL/DS, wurde eine Datenbankanfrage wohlüberlegt in ein Skript gebettet, dass vom Betriebssystem zur Ausführung gebracht wurde. Heutzutage existiert eine Vielzahl von Werkzeugen, mit denen SQL ad hoc ausgeführt werden kann. Viele dieser Werkzeuge bieten bereits eingebaute Reports, die mit wenigen Mausklicks konfigurierbar sind und sofort an das DBMS zur Ausführung gegeben werden können. Viele Datenbanken werden vom Anwender mit Hilfe von Front-Office-Werkzeugen, insbesondere Microsoft Access, konnektiert, wobei eine nicht vorhersehbare Last erzeugt wird. Hinzu kommen die Möglichkeiten, die dem Anwendungsentwickler durch dynamisches SQL zur Verfügung stehen. Der Kunde kann sich über eine Konfigurationsmaske genau die Daten aussuchen und mit den Bedingungen verknüpfen, die zur Stillung seines Informationsbedürfnisses gerade jetzt notwendig sind. Die Analyse der Top-SQL Statements, sortiert z. B. nach Ausführungszeit, CPU-Verbauch oder Anzahl Ausführungen fördert in einer gut gepflegten Datenbank zunächst nur die Statements hervor, die zwar wegen ihrer hohen Zahl der Ausführungen die Liste anführen, aber in allen relevanten Belangen optimiert wurden. Das viel zitierte Index Selection Problem (ISP) stellt sich oftmals nicht, da überhaupt kein passender Index für diese Abfrage vorhanden ist. Ein naiver Ansatz könnte darauf abzielen, alle die Spalten mit Indexen zu belegen, die in einer Abfrage vorkommen. Doch Datenbankindexe sind mit einem Preis versehen: sie verbrauchen Speicherplatz, besonders bei dichtbesetzten Sekundärindexe ist genauestens abzuwiegen. Weiterhin kostet das Aktualisieren bei jedem schreibendem Zugriff Rechenzeit, um die in der internen Relation geänderten Daten auch im Index entsprechend zu ändern. Nach dem Löschen oder Ändern wird es passieren, dass der Index reorganisiert, d. h. neu ausbalanciert werden muss. Dies alles muss notwendigerweise innerhalb der gleichen Transaktion erfolgen und somit persistent sein. Ein Index, der die Daten der internen Relation falsch abbildet und bei einem Zugriff über einen „Index Scan“ ohne „Table Access“ etwa ein NOTFOUND liefert, ist entschieden schlechter als gar kein Index. Einige der hier bereits beschriebenen Lösungsvorschläge zum Auto-Tuning und auch der hier aufzuzeigende Ansatz suchen nach dem Kompromiss zwischen schnellerer Zugriffszeit und Beanspruchung von weniger Ressourcen durch Aufbau und Pflege eines Index.

5.1. Ausgangsituation und Vorarbeiten

Hauptziel des Datenbanktuning, ob autonom oder manuell, ist die Reduzierung der Ausführungszeit von derzeitigen und zukünftigen Abfragen.
Die bisher erhältlichen und hier genannten Möglichkeiten des Auto-Tunings befassen sich oft mit einer gegebenen Auswahl an Indexen. Das heißt, von einer bestimmten Indexkonfiguration eines DBMS ausgehend, wird untersucht, ob weitere oder andere Indexe die Menge an zu analysierenden Abfragen, den Workload, entscheidend in Bezug auf Antwortzeit und Ressourcenverbrauch verbessern. Dabei können die Indexe zur Auswertung des Kosten-Nutzen-Verhältnisses entweder tatsächlich erzeugt werden oder das DBMS wird so verändert, dass der Optimizer davon ausgeht, diese nicht-materialisierten Indexe wären vorhanden und benutzbar. Auf diese Art und Weise werden unterschiedliche Indexkonfigurationen untersucht und ausgewertet und als Lösung des Rucksackproblems wird eine optimale Indexkonfiguration ermittelt und dem Datenbankadministrator vorgeschlagen. Durch bestimmte Erweiterungen kann das DBMS die gefundene Indexkonfiguration auch selbständig materialisieren. Dennoch bleibt der einzelne Index in seinem Aufbau gegeben und wird nicht betrachtet: entweder wird ein Index auf der Spalte erzeugt oder nicht. Die Idee von Sattler, Schallehn und Geist in [SSG05] ist die Verlagerung des Auto-Tunings um eine Ebene nach unten. Die Indexe werden als sich selbst organisierende Einheiten betrachtet, die sich dynamisch an die aktuellen Erfordernisse anpassen. Dies ermöglicht eine viel feinere Kontrolle als die Alternativen „Index materialisieren“, „Index nicht materialisieren“ und „Index löschen“.
Insbesondere bei hoch-komplexen Zugriffswerkzeugen, wie zum Beispiel beim Online-Analytical-Processing (OLAP), stößt die Untersuchung statischer Workloads an ihre Grenzen. Wenn innerhalb einer OLAP-Anfrage temporär Tabellen angelegt, gefüllt und mit Indexen versehen werden, können durch die bisher genannten Möglichkeiten des Auto-Tunings kaum brauchbare Aussagen für eine optimale Indexkonfiguration getroffen werden.
Das Tuning durch einen innerhalb eines bestimmen Zeitraums erzeugten statischen Workload ist insbesondere dann nur begrenzt aussagekräftig, wenn bestimmte Zyklen im generellen Lastverhalten einer Datenbank außer Acht gelassen werden. So gibt es je nach Anwendung Schwankungen an den verschiedenen Wochentagen, in ERP-Systemen wird nach Ende einer Abrechnungsperiode, z. B. Monats- oder Quartalsende, Jahresabschluss, außergewöhnlich viel Last erzeugt, die entweder gar nicht in den erstellten Workload einfließen oder selbigen verfälschen. Fehlen die Daten dieser Spitzenzeiten, wird sich die Performance im Zeitraum

dieser besonderen Situationen auch entsprechend verschlechtern. Dies kann u. U. sogar dazu führen, dass die Performance nach dem Tuning innerhalb dieser Zeiten sogar schlechter ist als vor den Maßnahmen, weil z. B. speziell für diese Zwecke erstellte Indexe durch das Auto-Tuning gelöscht wurden, da sie auf Basis des statischen Workloads als unnötig eingestuft wurden. Fließen die Daten der Sondersituation hingegen mit ein, werden Indexkonfigurationen als optimal ermittelt und eingestellt, die es im Normalzustand, außerhalb dieser Zeiten, gar nicht sind.

Von Sattler, Schallehn und Geist wurde in [SSG05] ein Ansatz beschrieben, bei dem ein binärer Suchbaum (vgl. 5.1.2) um eine Komponente zur Reorganisation nach dem Kriterium der Häufigkeit eines Zugriffs auf einen Knoten erweitert wurde.

Dazu wird es notwendig werden einen gewissen Overhead[26] in Kauf zu nehmen, um Statistiken führen zu können, über welchen Knoten im Baum wie häufig oder wenig zugegriffen wird.

5.1.1. Problem und Lösungsansatz

Für keine der in Kapitel 4 vorgestellten Tuning-Methoden spielen die benutzten Indexstrukuren selbst eine Rolle. Die existierenden Ansätze sind immer datenbasiert, d. h. der Index wird über die alle Ausprägungen der Zielspalte oder Zielspalten erzeugt, ohne deren Relevanz für spätere Abfragen zu berücksichtigen. Dabei wird versucht, für eine bestimmte Gruppe von Abfragen (z. B. Punkt- oder Bereichsabfragen) eine möglichst niedrige Komplexität zu erzielen.

Struktur	Punktabfrage	Bereichsabfrage	Einfügen	Löschen	Balancierung
Heap	*O(n)*	*O(n)*	*O(1)*	*O(n)*	
Liste	*O(log n)*	*O(log n)*	*O(log n)*	*O(log n)*	
Hash	*O(1)*		*O(1)*	*O(1)*	
Binärbaum	*O(log n)*	*O(log n + r[27])*	*O(log n)*	*O(log n)*	teuer
B-Baum	*O(log n)*	*O(log n + r)*	*O(log n)*	*O(log n)*	geringe Kosten
B+-Baum	*O(log n)*	*O(log n + r)*	*O(log n)*	*O(log n)*	geringe Kosten

Tabelle 14: Komplexität verschiedener Zugriffsstrukturen

[26] gemeint ist die Speicherung von zusätzlichen Metadaten im Index

[27] r steht für die Anzahl der Elemente in der Bereichsabfrage

In der Tabelle werden die Komplexitäten zur Übersicht dargestellt und es wird deutlich, dass die Verfahren je nach Anwendungsbereich ihre Stärken und Schwächen haben. Vergleiche dazu auch Kapitel 3. Keine Rolle spielt dabei die Zugriffshäufigkeit. Werden die durch einen Indexeintrag referenzierten Tupel häufig abgefragt, werden sie selten oder gar nicht benötigt? Diese Information bieten klassische Ansätze nicht, obwohl ein Index nicht weniger Speicherplatz benötigt, wenn er wenig benutzt wird oder wenn er nur benutzt wird um z. B. 10% der referenzierten Tupel auszulesen. Der Nutzen oder Gewinn eines Index wird durch seine Kosten, in diesem Falle der für den Index verbrauchte Platz, geschmälert. In Index, der nur für 10% aller Tupel benötigt wird, verursacht dennoch 100% Kosten.
Einzig die Erzeugung von Teilindexen (vgl. 4.3 Partial Indexes) kann dazu benutzt werden, wobei jedoch im Vorhinein bekannt sein muss, welche Zeilen wenig oder gar nicht angesprochen werden.
Die Differenzierung in häufig und nicht häufig ist in jedem Falle praxisrelevant. Im klassischen Beispiel einer Auftragstabelle, die alle Kundenaufträge enthält, werden die aktuell bearbeiteten Aufträge häufig referenziert, während alte, abgearbeitete Aufträge oder neu geladene kaum oder gar nicht mehr abgefragt werden. Im Gegensatz zum „Alles-oder-Nichts“ Tuning[28] kann mit einem last-basierten Index viel feiner gesteuert werden, wie effizient auf die Daten zugegriffen werden kann und andererseits, wie viel Platz von diesem Index verbraucht wird.

5.1.2. Last-balancierter Binärbaum

In [SSG05] beschreiben Sattler, Schallehn und Geist einen Binärbaum, der durch die Häufigkeit der Zugriffe auf einen Knoten balanciert wird. Weiterhin ist es möglich die Zahl der erlaubten Knoten zur Laufzeit zu erhöhen bzw. zu verringern, was eine Reorganisation des Baums auslöst. Übersteigt die Zahl der Einträge im Baum die Zahl der Knoten werden die Einträge in Page Container ausgelagert. Diese Page Container sind die Blätter eines inneren Knotens und arbeiten wie eine Liste. Überschüssige Einträge, für die kein Knoten mehr erstellt werden kann, werden dort gespeichert.
Wird die Zahl der Knoten verringert werden jeweils zwei Knoten aufgelöst und ihre jeweiligen Page Container fallen an den Vaterknoten. Steigt die Zahl der erlaubten Knoten, kann der Baum wieder dichter gespeichert werden und ein Knoten kann in zwei Folgeknoten aufgespalten werden.

[28] Index anlegen oder löschen

Folgende Abbildung stellt einen klassischen Binärbaum mit sieben Knoten dar.

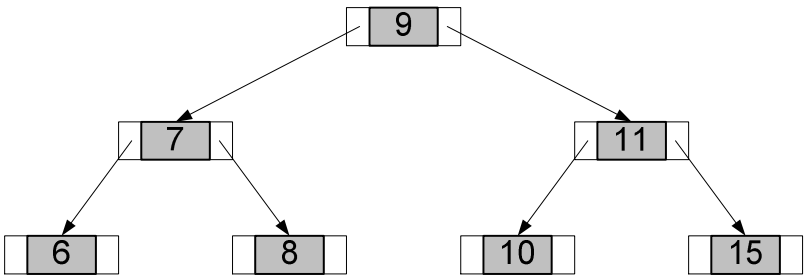


Abbildung 13: Binärbaum mit sieben Knoten

Nimm man jetzt an, dass diese Knotenzahl auch das derzeitige Limit ist, wäre das Einfügen eines weiteren Elements nicht mehr möglich. Im Bezug auf DBMS bedeutet das, dass entweder der Wert nicht in die indexierte Tabelle eingefügt werden kann oder dass der Index nicht mehr benutzt werden darf. Aus diesem Grunde wurde im beschriebenen Ansatz das Konstrukt „Page Container“ eingeführt. Wie eine Art Überlaufseite nimmt es diejenigen Elemente auf, die aufgrund der limitierten Knotenanzahl keinen eigenen Knoten mehr erhalten können.

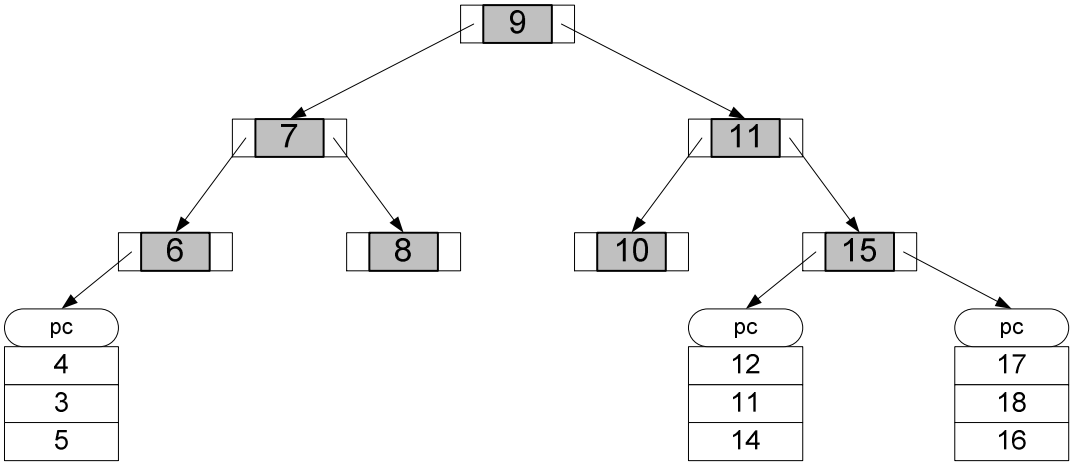


Abbildung 14: Binärbaum mit sieben Knoten und drei Page Containern (pc)

Bei einer Erweiterung der Knotenzahl können die Elemente in den Page Container „aufsteigen“ und einen eigenen Knoten bilden.

Um eine Balancierung nach der Zugriffshäufigkeit zu gewährleisten, werden an jedem Page Container die Anzahl der Zugriffe *pageAccesses* gezählt und gespeichert. Die Anzahl der Zugriffe an der Wurzel entspricht der Zahl aller Zugriffe. Somit ergibt sich die Balance aus dem Quotienen aller Zugriffe durch die Anzahl der im Baum enthaltenen Page Container.

$$balance = \frac{\sum pageAccesses}{Anzahl\ PageContainer}$$

Wenn zwei Page Container pc_1 und pc_2 eines Knoten im Baum die Bedingung

$$pageAccesses(pc_1) + pageAccesses(pc_2) < balance$$

erfüllt, werden pc_1 und pc_2 zusammen gelegt und mit dem Vaterknoten verknüpft. Träfe dies im obigen Beispiel für die Page Container unter dem Knoten „15“ zu, ergäbe sich folgender Baum:

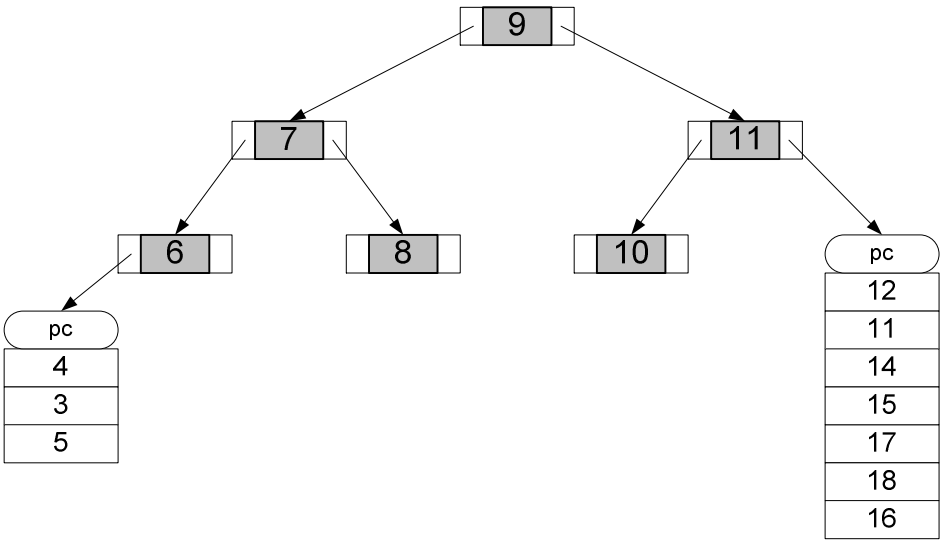


Abbildung 15: Binärbaum mit sechs Knoten und zwei Page Containern (pc)

Für den Baum steht nun wieder ein Knoten zur Verfügung, der verwendet werden kann, um einen Page Container, für den gilt

$$pageAccesses(pc_1) < 2 \times balance$$

in zwei neue Page Container zu spalten, wobei der Median aus diesem Container den neuen Knoten bildet.

Die experimentelle Evaluierung mit 100.000 Tupeln, 100 Tupeln pro Page und 1000 Nodes zeigte, dass der zugriffsbalancierte Baum der Variante ohne Zugriffsbalancierung um einen linearen Faktor überlegen ist. Nach anfänglich rasant steigenden Reorganisationen stabilisiert sich der Baum ab etwa 100.000 Zugriffen und muss nur noch selten reorganisiert werden.

Der Ansatz ist nicht unmittelbar auf DBMS übertragbar, da ein binärer Baum keine optimale Zugriffsstruktur darstellt. Dieser Ansatz klärt natürlich auch nicht die Probleme, die bei gleichzeitigem Zugriff und folgender Reorganisation auftreten oder wie mehrere

Zugriffspfade unterstützt werden können. Er zeigt aber den Weg auf, den ein feiner granuliertes Datenbank-Tuning nehmen muss.

6. Entwurf eines Last-balancierten B-Baums

Ausgehend vom skizzierten Ansatz des zugriffsbalancierten Binärbaums wird in diesem Kapitel das Konzept eines B-Baums (vgl. Kapitel 3.4.1) entwickelt, der seine Knoten nach der Zugriffshäufigkeit anordnet.

Die Knotenanzahl soll explizit vorgegeben werden können und sie soll auch bei einem bereits erstellten B-Baum verändert werden können. Die Anzahl der Knoten simuliert eine Speicherplatzgrenze, die einem Index vor der Erstellung zugewiesen werden kann. Ändert sich diese Grenze zur Laufzeit, wird darauf mittels einer Reorganisation zu reagieren sein. Mittels einer festen Knotengröße soll ferner eine Speicherplatzgrenze gezogen werden können, die auch zur Laufzeit änderbar sein soll.

Je häufiger auf einen Knoten zugegriffen wird, desto tiefer müssen die Knoten gestaffelt werden. Damit dies im Hinblick auf die maximale Knotenanzahl geschehen kann, müssen in anderen Bereichen des Baums, die einer geringeren Zugriffshäufigkeit unterliegen, die Knoten aufgelöst werden.

Das Ziel des im Rahmen dieser Diplomarbeit entwickelten Last-balancierten B-Baums ist zum einen die Verringerung des Speicherplatzverbrauchs bei annährend gleicher Performance und zum anderen der effizientere Zugriff auf bestimmte Teilbereiche im Vergleich zum Daten-balancierten B-Baum. Beide Ziele sind mittels eines autonomen Tunings zu erreichen. Das heißt, dass sich die Balancierung innerhalb des Baums nicht allein nach der Verteilung der Daten ausrichten kann, so wie der in Kapitel 3.4.1 bereits beschrieben B-Baum, sondern zusätzlich nach der Häufigkeit der Zugriffe auf bestimmte Knoten.

Der im Folgenden entwickelte Last-balancierte B-Baum wurde anschließend prototypisch in der Programmiersprache „Java“ implementiert. Anhand dieser Implementierung werden die hier entwickelten Thesen überprüft und mit einem ausschließlich Daten-balancierten B-Baum verglichen.

6.1. Die Seitenstruktur Bucket

Ein B-Baum, wie in 3.4.1 beschrieben, bietet keine Möglichkeit der Einschränkung des Speicherplatzbedarfs. Dieser hängt immer von der Anzahl und Größe aller abgebildeten Elemente ab. Um die Forderung nach einer Speicherplatzschranke zu erfüllen, wird vorgeschlagen, den B-Baum um eine Seitenstruktur zu erweitern, die im folgenden „Bucket“ genannt wird. Im Unterschied zu einem normalen Knoten (hier „Node“ genannt) wird ein Bucket typischerweise wesentlich mehr Elemente aufnehmen als ein Node. Die Elemente, hier ein Schlüssel und ein damit verknüpftes Objekt, werden in einem Node sortiert gespeichert. Dies wäre für ein Bucket ebenfalls möglich. So könnte ein Bucket eine verkettete Liste darstellen oder über eine Hash-Funktion befüllt werden. Da in einem Bucket, wie noch gezeigt wird, lediglich die Elemente gesammelt werden, die einer niedrigeren Zugriffshäufigkeit unterliegen, wurde im hier skizzierten Ansatz die Organisationsform einer unsortierte Liste gewählt. Für bestimmte Anwendungen kann es jedoch erforderlich sein, ein Bucket zu sortieren. Soll ein Bucket z. B. geteilt werden, empfiehlt es sich, den Median heraus zu suchen. Dieser ist aufgrund der fehlenden Sortierung nicht das Tupel in der Mitte der Seite, sondern muss durch Untersuchung jeden einzelnen Elementes aufwändig ermittelt werden. Aus dem gleichen Grund ist auch die Suche nach einem beliebigen Tupel in einem Bucket aufwändiger. Da ein Bucket vornehmlich dazu gedacht, die Werte aufzunehmen, auf die wenig oder gar nicht zugegriffen wird, muss bewiesen werden, ob sich eine andere Organisationsform aus Performancegründen lohnt. Da für die Suche in einem Bucket durch die Struktur des B-Baums selbst schon ein Großteil der Sortierung – im Vergleich zur Suche auf der unsortierten Hauptdatei – erledigt ist, wird die erhöhte Komplexität hier akzeptiert.

Entsteht beim Befüllen eines Buckets ein Überlauf wird nicht, wie beim Node, der Median in den Vaterknoten verschoben und das Node geteilt, sondern es wird ein Überlauf-Bucket erzeugt, der mit dem ursprünglichen Bucket linear verknüpft wird. Ein Bucket kann also immer nur einen Nachfolger haben und nicht 2*k+1 wie bei einem Node, wobei k für die Ordnung des Baums steht. Daraus wird ersichtlich, dass eine weitere Beschränkung, denen die Nodes unterliegen, nicht für einen Bucket gilt. Die Ordnung eines B-Baums gibt an, wie viele

Elemente mindestens auf einem Node zu sein haben.[29] Läuft ein Bucket über und wird ein Überlauf-Bucket erstellt, so findet sich darauf nur ein Element.

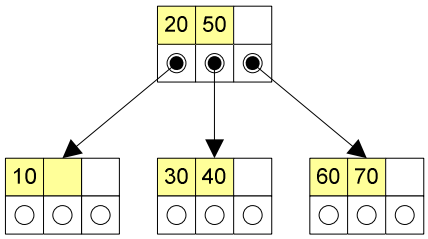


Abbildung 16: B-Baum der Ordnung k=1

In der obigen Abbildung ist ein B-Baum dargestellt, der fast vollständig gefüllt ist. Lediglich ein Wert n < 20 hätte darauf noch Platz, ohne die Spaltung eines Knotens zu erzeugen. Unter der Annahme, dass die Knoten auf 4 limitiert wurden, könnten jetzt keine weiteren Werte bis auf den genannten Bereich eingefügt werden. Denn dies hätte die Spaltung eines Knotens zur Folge, dessen Median in Richtung der Wurzel wandert und würde – da die Wurzel ebenfalls mit 2*k Werten belegt ist – eine erneute Spaltung mit der Erstellung einer neuen Wurzel nach sich ziehen. Das Einfügen eines Wertes führt, in diesem Falle, zum Verbrauch von 2 Knoten. Ohne die Grenze von 4 Knoten (Nodes) zu überschreiten, kann der Baum den Wert nicht aufnehmen. Wäre dies die Grundlage eines Index, müsste das Einfügen eines Wertes z. B. „80“ abgelehnt werden oder der Index würde die Relation nur noch teilweise abbilden. Beides sind Ausschlusskriterien für den Einsatz in DBMS.

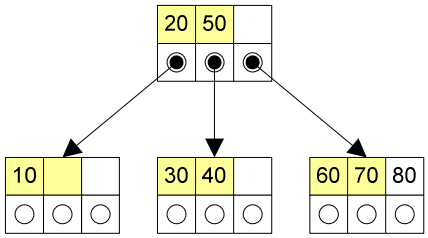


Abbildung 17: Einfügen des Schlüssels „80“

Der erzeugte Überlauf auf dem unteren rechten Knoten muss nun also anderes behandelt werden. In diesem Ansatz würden daher nun die eingehend beschriebenen Buckets

[29] Nach einer anderen Definition bezeichnet die Ordnung die maximale Anzahl der Elemente auf einem Knoten, hier jedoch wird die o. g. Definition verwendet.

herangezogen. Da die maximale Zahl der Knoten erreicht ist, bleibt nur noch die Alternative ein Bucket zu erzeugen und den Wert dort zu speichern.

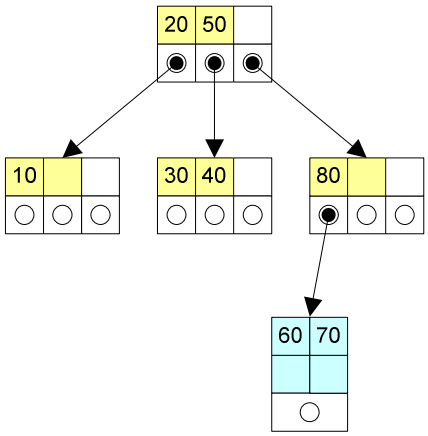


Abbildung 18: Einfügen des Schlüssels „80“ und Erzeugen eines Buckets

In diesem Ansatz wird in einem solchen Fall zunächst versucht, das Bucket unter dem niedrigsten (d. h. der am weitesten links liegende) Pointer zu erzeugen.

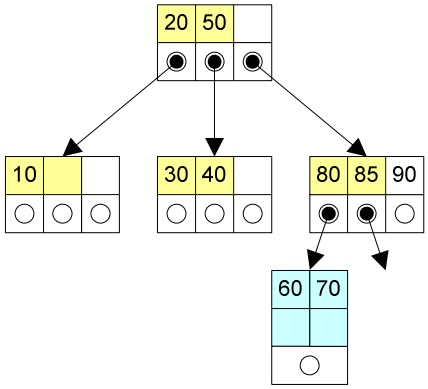


Abbildung 19: Einfügen des Schlüssels „85“

Existiert dort schon ein Bucket, wird nach rechts ausgewichen und dieser Pointer geprüft. Im Beispiel wird der Schlüssel „85“ eingefügt. Dabei wird er – trotz der Überlaufs – dort eingefügt, wo er im Sinne der Sortierung zu stehen hätte. Der „alte“ Schlüssel „90“ befindet sich auf der Überlaufposition. Da die Pointer-Position 1 im Knoten besetzt ist, könnte der Wert „85“ in das neu erzeugte Bucket „hinunter gestoßen“ werden. Bis auf den Wert auf der Überlaufposition (hier „90“) würden alle Schlüssel in das neu erzeugte Bucket gestellt und auf dem Ursprungsknoten gelöscht. Lediglich der Wert auf der Überlaufposition verbleibt und wird nach links bewegt bis an die Position des neuen Buckets.

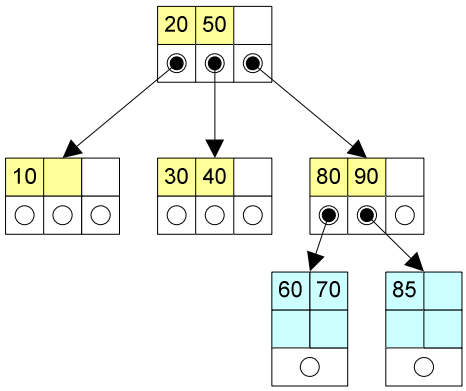


Abbildung 20: Einfügen des Schlüssels „85“ und Erzeugen eines zweiten Buckets

Für den Fall, dass alle Pointer belegt sind, kann mittels eines geeigneten Algorithmus vor dem Verschieben in das der Sortierung nach richtige Bucket eine andere Alternative ausgewählt werden. Dieser Algorithmus kann leicht auf die Erfordernisse einer Lastbalancierung angepasst werden. Dabei wäre zu definieren, wie mit neu eingefügten Werten in einem Last-balancierten Baum umgegangen werden soll. Dabei sind mehrere Varianten denkbar:

- der Wert wird unabhängig von der Last-Balancierung eingefügt
- der Wert wird dort eingefügt, wo die Belegung am geringsten ist
- der Wert wird dort eingefügt, wo die wenigsten Zugriffe erfolgen
- der Wert wird dort eingefügt, wo die meisten Zugriffe erfolgen

Für die beiden letzten Varianten müsste eine Strategie a priori festgelegt und ausgewählt werden. Da zum Zeitpunkt des Einfügens eines Tupels nicht bekannt ist, wie viele Zugriffe es erhält, wird eine Parametrisierung vorgeschlagen, über die der Anwendungsentwickler oder der Datenbankadministrator die Befüllung steuern kann. Dies erfordert jedoch Kenntnisse der logischen Zusammenhänge der Relation. So könnte bei einer Auftragstabelle z. B. davon ausgegangen werden, dass neu eingefügte Werte, das sind frisch eröffnete Aufträge, zunächst wenig abgefragt werden. Daher würde eine Einsortierung des Elements sortiert nach der Anzahl der geringsten Zugriffshäufigkeit die Strategie, die den größten Nutzen verspricht. In der prototypischen Implementierung wurde darauf verzichtet, da das Einfügen eines Wertes von der Last-Balancierung abhängig zu machen, da erst die Lookup-Operationen die Statistiken dafür liefern.

Im Falle eines Bucketüberlaufs wird einfach ein Folgebucket erzeugt, statt wie üblich das überlaufende Objekt zu splitten. Ein Split wäre sehr aufwändig, da, wie bereits bekannt, zum Auffinden des Medians das Bucket zunächst sortiert werden müsste. Ab einer gewissen

Größe, würde dies viel Zeit in Anspruch nehmen und ggf. sogar eine Reihe von Festplattenzugriffen erforderlich machen.

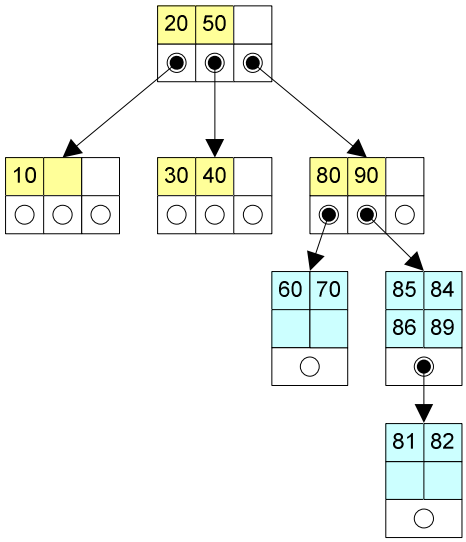


Abbildung 21: Verkettung von Folgebuckets

Diese Art der Verkettung hat den Nachteil, dass Schlüssel, die – geht man von der Sortierung aus – eigentlich bereits auf dem ersten Bucket zu finden sein müssten, aber auf ein Folgebucket ausgelagert wurden (z. B. „81“ und „82“). Daher muss bei der Suche nach einem Tupel in einem Bucket bei Existenz eines Folgebuckets auch dieses durchsucht werden.

6.2. Erheben und Speichern der Statistiken

Um eine Aussage über die Zugriffshäufigkeit treffen zu können, müssen die einzelnen Nodes und Buckets Daten über die Anzahl der aktuellen Zugriffe protokollieren. In diesem Ansatz wird zwischen vier verschiedenen Arten von lesenden Zugriffen unterschieden, die jeweils nach einem bestimmten Wert im Baum suchen.

- Selektion – Suchen eines Wertes
- Insert – Suchen eines Wertes zum Auffinden der Einfügeposition
- Delete – Suchen eines Wertes zur Löschung
- Reorganisation – Suchen eines Wertes zu Reorganisationszwecken, z. B. bei einem Split

Alle diese Zugriffe werden für jedes Objekt einzeln gespeichert. Dies ermöglicht eine differenzierte Aussage über die Benutzung der Nodes und Buckets. So lassen sich beispielsweise genau die Bereiche eines Index ausmachen, in denen verstärkt gelöscht wird. Kombiniert mit einer verringerten Zugriffsanzahl für die Selektion kann davon ausgegangen

werden, dass es sich um obsolete Bereiche in der Tabelle handelt. Eine Zusammenfassung dieser Werte in Buckets ist zu empfehlen, da beim Löschen eines Wertes aus einem Bucket keine Unterlaufsituation behandelt werden muss. Umgekehrt kann bei Häufung der Zugriffe für Inserts in einem Teilbaum proaktiv das Freigeben von Nodes in einem anderen Bereich angestoßen werden, um Reorganisationsaufwand beim Insert selbst klein zu halten.
In diesem Zusammenhang ist die Frage zu klären, ob das Speichern der Statistiken Teil der Transaktion ist oder ob davon abgelöst werden muss. Da in diesem Ansatz die Statistiken nicht an der Wurzel, sondern dezentral in den jeweiligen Knoten gespeichert werden, würde ein Einbinden in die Transaktion zu teuer werden. Auf der anderen Seite ist die Priorität der Statistiken denkbar gering. Mögliche geeignete Alternativen sind zeitverzögerte Speicherung und/oder gesonderte Behandlung außerhalb des Transaktionskonzepts.

6.3. Last-Balancierung

Nimmt man für den Beispielbaum 100 Zugriffe an, die sich wie in **Abbildung 22:** Zugriffstatistiken verteilen, wird ersichtlich, dass im linken Teilbaum die Zugriffe deutlich geringer ausfallen als in der rechten Hälfte.

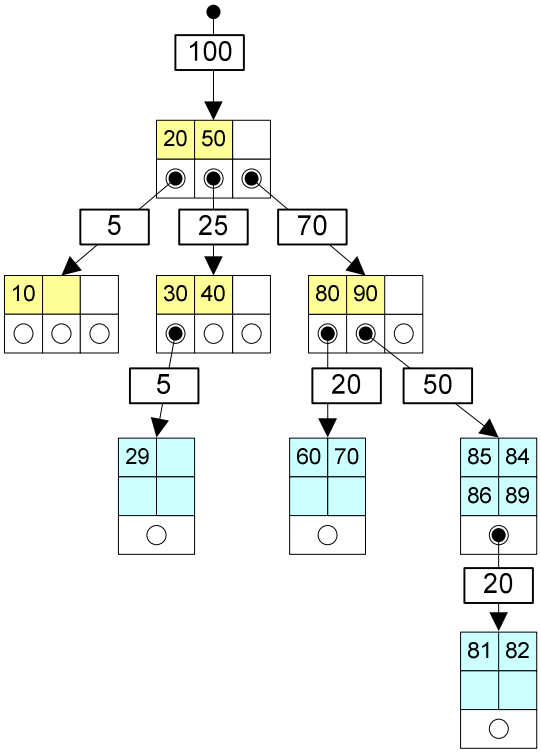


Abbildung 22: Zugriffstatistiken

Ist die Ebene 0 die Wurzel mit 100 Zugriffen, so wäre der Baum ausgeglichen, wenn in

Ebene 1 unterhalb der Wurzel die Gruppe aus drei Knoten für jedes Element genau 100/3 = 33,3 Zugriffe aufwiese. Diese Zahl wird im Folgenden als Balance bezeichnet.

$$balance = \frac{\sum Knotenzugriffe\ \ pro\ Gruppe}{Anzahl\ \ Knoten\ \ pro\ Gruppe}$$

Bei Bedarf kann diese Zahl über das Wurzelelement und gegebener Gruppe ermittelt werden. Eine Gruppe definiert sich immer über den Vaterknoten. Das bedeutet, dass auf jeder Stufe des Baums für jeden Vaterknoten eine Gruppe existiert.
Bei der Last-Balancierung werden grundsätzlich zwei Verfahren anwendet. Knoten, die in ihrer Zugriffshäufigkeit deutlich hinter dem Durchschnitt zurück fallen, müssen aufgelöst oder zusammengelegt werden. Das Auswahlkriterium dafür bietet die bereits besprochene Balance einer Gruppe. Es gilt eine Gruppe zu finden, in der zwei benachbarte Knoten kn_1 und kn_2 zusammen weniger Zugriffe aufweisen als der Durchschnitt dieser Gruppe (Balance).

$$\sum Zugriffe\ kn_1 + \sum Zugriffe\ kn_2 < Balance$$

Im Regelfall kann bei der Balancierung nach Zugriffshäufigkeit davon ausgegangen werden, dass die Obergrenze für neue Nodes bereits erreicht wurde. Somit würden die Knoten kn_1 und kn_2 in ein neues Bucket integriert werden. Dies ist jedoch nur möglich, wenn beide Knoten nicht mehr als einen Verweis (Pointer) auf eine tiefere Stufe beinhalten. Ist dies dennoch der Fall, müssten zunächst die Knoten dieser unteren Gruppe zusammengefasst werden. Es ist daher angebracht, von der Wurzel aus zunächst auf der tiefsten Stufe des Baums die Gruppen zu untersuchen.

In der folgenden **Abbildung 23:** Gruppen mit Optimierungskandidaten sind die Bereiche hinterlegt, in denen der Algorithmus nach der oben genannten Formel nach Kandidaten zum Zusammenfassen suchen würde. Dabei wird ersichtlich, dass es in der Gruppe 1 zwar zwei Kandidaten gäbe, diese aber nicht zu einem Bucket zusammengefasst werden können, weil drei Folgepointer existieren. In der Gruppe 2 müssen zwei Buckets zusammengefasst werden, da die Summe ihrer Zugriffe (4 + 1 = 5) unter der Balance (25 / 3 = 8,3) für diese Gruppe liegt. Für die Gruppe 3 kann lediglich die Aussage getroffen werden, dass der Pointer mit den geringeren Zugriffen im Hinblick auf das Balance-Kriterium immer noch zu viel und der

Pointer mit den meisten Zugriffen noch zu wenige Hits ausweist. Ziel dieser Maßnahme ist es, auf der darüber liegenden Stufe Kandidaten für eine Vereinigung zu schaffen.

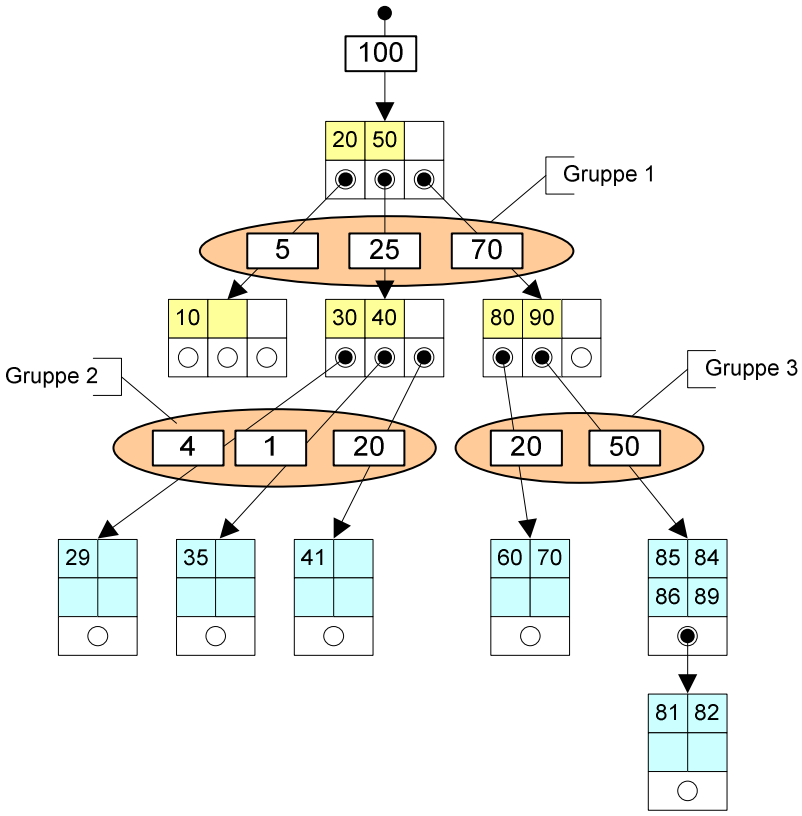


Abbildung 23: Gruppen mit Optimierungskandidaten

Bei der Zusammenfassung werden zwei Varianten unterschieden. Zum einen kann es auf der zu vereinigenden Seite zu einem Überlauf kommen. Daher wird vorher geprüft, ob dies passieren wird. Der erste Schritt des Zusammenfassens ist das Verschieben des Schlüssels an der Position des Pointers zur linken Seite (dem Bucket mit dem Schlüssel 29 im obigen Beispiel). Anschließend werden die Werte von rechten auf die linke Seite verschoben.

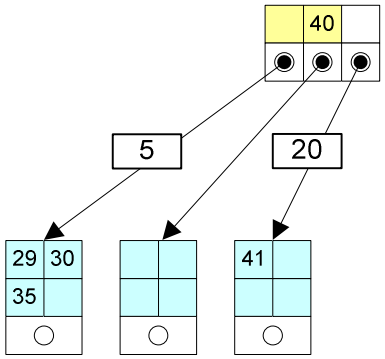


Abbildung 24: Zwischenstand des Zusammenfassens

Die nun leere rechte Seite wird gelöscht, indem ihr Zeiger auf Position 2 im Vaterknoten entfernt wird. Die Schlüssel im Vaterknoten werden um eine Stelle nach links bewegt und ebenso die übrigen Pointer. Beim Zusammenfassen von zwei Knoten werden die Statistiken addiert.
In der Variante, in der es zu einem Überlauf kommt, würden so viele Werte wie möglich vom rechten auf den linken Knoten übertragen und der letzte Schritt bestünde statt des Löschens des Pointers im Vaterknoten im Anhängen des rechten an den linken Knoten.

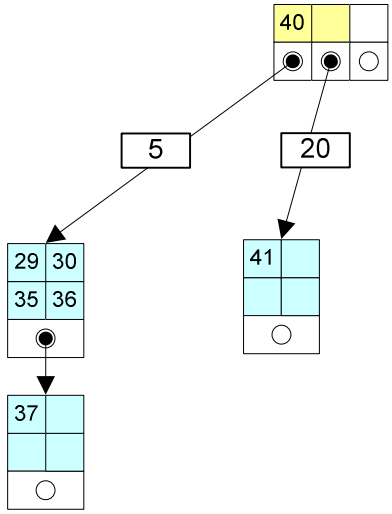


Abbildung 25: Variante mit Überlauf

Zur Vermeidung von Thrashing wird vorgeschlagen, die Last-Balancierung nicht mit jedem Zugriff auf die Struktur neu auszurechnen. Stattdessen sollte je nach Größe des Baums ein Intervall festgelegt werden, nach dem Baum seine Balancierung erneut überprüft und gegebenenfalls korrigiert. Weiterhin kann eine Schwelle definiert werden, um die die Balance überschritten werden muss, bevor eine Reorganisation gestartet wird. Schließlich kann sich ein Knoten ebenfalls merken, ob und wie er während der letzten Reorganisation verändert wurde. Eine solche Information kann mit dem Trashing-Schwellwert kombiniert werden, der z. B. verdoppelt wird, wenn nach einem Split jetzt ein Merge oder umgekehrt durchgeführt werden soll.

Die zweite Form der Reorganisation ist das Aufspalten eines Knotens. Für das Aufspalten gelte folgende auslösende Bedingung

$$\sum Zugriffe\ kn_1 > 2 \times balance$$

Da sich ein Split bis zur Wurzel kaskadierend fortsetzen kann, muss im Vorhinein geprüft werden, ob dafür genügend Nodes zur Verfügung stehen. Ist dies nicht der Fall, darf auch die Aufteilung nicht passieren, da sie den Baum in einem ungültigen Stand zurück lassen würde. Daher wird eine weitere Form der Reorganisation vorgeschlagen: das Transformieren eines Nodes in ein Bucket. Sie sollte vor der Phase „Split“ erfolgen, damit sich die Chance erhöht, dass genügend Buckets zur Verfügung stehen.

1. Transform – Umwandlung von Nodes in Buckets
2. Merge – Vereinigen von zwei Knoten
3. Split – Aufspalten eines Knotens

Bei der Transformierung werden diejenigen Nodes identifiziert, die in ein Bucket umgewandelt werden sollten. Dafür wird die Bedingung vorgeschlagen, dass ein Knoten kn_1 vom Objekttyp „Node“ dann in ein Bucket umgewandelt wird, wenn die Anzahl seiner Zugriffe kleiner ist als die Hälfte der Balance:

$$\sum Zugriffe\ kn_1 < \frac{balance}{2}$$

Die Umwandlung ist genau dann trivial, wenn es 0-1 Nachfolger des Kandidaten kn_1 gibt, die ebenfalls vom Objekttyp „Bucket“ sind. Sind diese Nachfolger vom Typ „Node“, müssen sie zunächst in Buckets überführt werden. Bei mehr als einem Nachfolgeknoten werden die Buckets linear verknüpft. Da bei der erfolglosen Suche in einem Bucket immer an einem Nachfolger fortgesetzt wird, kann auf eine Sortierung verzichtet werden.

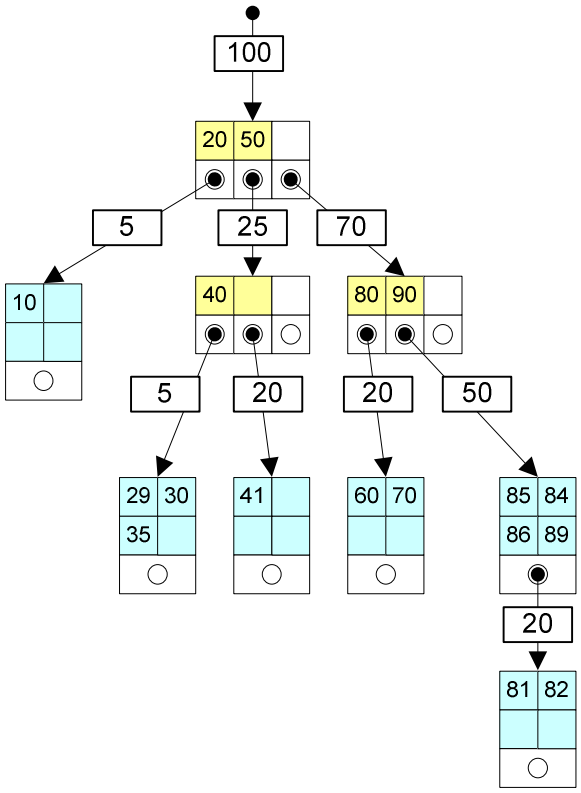


Abbildung 26: Umwandlung in ein Bucket

Die Abbildung zeigt den Beispielbaum nach erfolgter Umwandlung des linken Nodes in ein Bucket. Nach einer solchen Transformation besteht die Möglichkeit, dass genügend Nodes für einen kaskadierenden Split zur Verfügung stehen. In diesem Beispiel bedürfte es noch einer weiteren Umwandlung – obwohl hier die Zugriffsuntergrenze balance/2 nicht erreicht wurde. Ein Split wäre auch dann nicht möglich, weil bei einer Ordnung von k=1 kein Median mehr zur Verfügung stünde. Eine Alternative ist das Heranziehen von Werten aus dem verknüpften Bucket. In diesem Beispiel wird der erste Wert des Buckets („85“) in das Node mit den Schlüsseln „80“ und „90“ verschoben, das dann gesplittet werden kann.

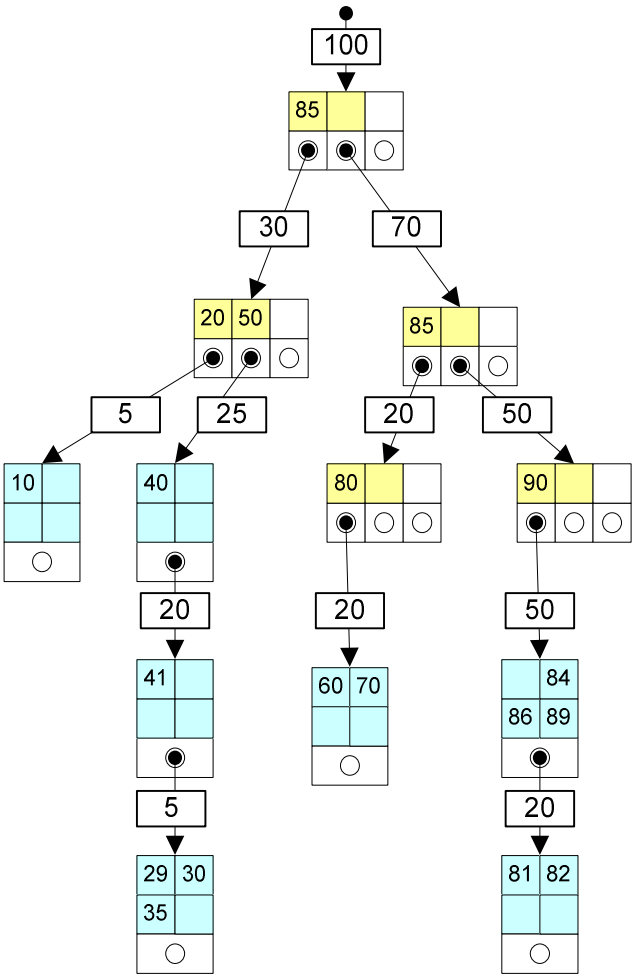


Abbildung 27: Split (kaskadierend)

Dieses Beispiel – trotz der Einschränkungen auf k=1 – veranschaulicht, wie der Baum aufgrund der unterschiedlichen Zugriffshäufigkeit gestaffelt ist. Im rechten Teilbaum sind die Werte tiefer gestaffelt, da auf dieser Stufe noch Nodes verwendet werden. Würde sich die Zugriffshäufigkeit derart verändern, dass der linke Teilbaum bevorzugt werden müsste, würde dies auf die gleiche Weise erfolgen: Transform und Split und ggf. Merge.

6.4. Kostenmodell

6.4.1. Index-Selection-Problem

Wie bereits im Kapitel 3 besprochen, wird ein Index angelegt, um eine möglichst optimale Performance beim Zugriff auf die Daten zu ermöglichen. Dabei steht zum einen die Frage im Mittelpunkt, ob überhaupt ein Index angelegt werden soll und zum anderen welcher Index der geeignetste sei. Es ist davon auszugehen, dass sowohl im manuellen als auch im autonomen Tuning eine Vielzahl an Indextypen zur Verfügung steht, die ihrerseits auch diverse

Konfigurationen bieten. Um eine Entscheidung unter Kosten-/Nutzengesichtspunkten treffen zu können, ist es notwendig einen Indexkandidaten daraufhin zu bewerten. Dabei ist der Nutzen eines Index seinen Kosten gegenüber zu stellen, um den Profit zu erhalten.
Für eine Menge von Anfragen $Q_1,\ldots,Q_m$ existieren die möglichen Indexkandidaten $I_1,\ldots,I_n$. Der Profit eines Indexkandidaten I_k für eine Abfrage Q_i ergibt sich aus der Differenz der Kosten für die Ausführung der Abfrage mit und ohne einen Index I_k:

$$profit(Q_i, I_k) = max\{0, cost(Q_i) - cost(Q_i, I_k)\}$$

Der verbrauchten Speicherplatz $size(I_k)$ als auch die Rechenzeit zur Aktualisierung des Index $mcost(I_k)$ müssen vom Profit noch abgezogen werden. Für die möglichen Indexkonfigurationen $C \subseteq \{I_1 \ldots I_n\}$ ergibt sich das ISP

$$\sum_{i=1}^{m} \max\left\{ profit(Q_i, I_j) : I_j \in C \right\} - \sum_{I_j \in C} mcost(I_j)$$

unter Vermeidung der Überschreitung der Speicherplatzschranke S

$$\sum_{I_j \in C} size(I_j) \leq S$$

Ein Ansatz zur Lösung dieses NP-Problems[30] ist z. B. der sogenannte Greedy-Algorithmus, weiterhin existieren approximative Vorschläge. [LSS07] [Lü07]

6.4.2. Kosten des Last-balancierten B-Baums

Das autonome Self-Tuning für einen Last-balancierten B-Baum verfolgt einen feiner granulierten Ansatz. Das Problem sind hier nicht eine Menge von Indexen über n Relationen mit m Attributen, sondern die Wahl der richtigen Parameter.
Der Last-balancierte B-Baum würde hauptsächlich dort zum Einsatz kommen, wo zum einen Gewissheit über die Notwendigkeit eines Index auf einem Attribut besteht, aber die Kosten für einen vollständigen Index im Vergleich zum erzielbaren Nutzen zu hoch sind. Ein Vergleich des hier skizzierten Konzepts auf der Basis von Zugriffszeiten mit einem

[30] Variante des Rucksackproblem, siehe auch [KPP04]

vollständig höhenbalancierten Baum ergibt wenig Sinn. Bei einer Gleichverteilung der Abfragen über den gesamten Wertebereich der zu indexierenden Spalte, wird der hier beschriebene Ansatz ebenfalls keinem Vergleich standhalten.
Ein Last-balancierter B-Baum kann vom DBMS an eine im Rahmen der Lösung des ISP definierte Speicherplatzschranke S angepasst werden. Dazu können die Parameter Maximalzahl der zugelassenen Nodes und der Anzahl der Schlüssel pro Node justiert werden. Dabei wird davon ausgegangen, dass die in Buckets abgelegten Tupel keinen bzw. nur wenig Platz im Index beanspruchen. Dies ist dann der Fall, wenn man den Pointer auf ein Bucket als direkten Verweis auf die Daten in der Relation versteht.
Der Vorteil eines Last-balancierten Indexkandidaten als Menge der Parametereinstellungen maximale Nodeanzahl, maximale Schlüsselanzahl pro Node und der Reorganisationsfrequenz $I_{Last} = \{I_1,\ldots,I_k\}$ kann als analog zu [LSS07] wie folgt ausgedrückt werden:

$$profit(Q, I_{Last}) = cost(Q) - cost(Q, I_{Last})$$

wobei sich der tatsächliche *profit(Q, I)* eines solchen Index nach dem in [SSG04] empfohlenen Ansatz zur Aufteilung nach der Indexgröße annähren ließe:

$$profit(Q,I) = \frac{profit(Q,I_{Last}) \cdot size(I)}{\sum_{I_j \in I_{Last}} size(I_j)}$$

Wird der Gesamtprofit *benefit(I)* ermittelt, indem die Einzelprofite pro Anfrage kumuliert werden, lässt sich daraus ein in [SSG04] vorgestellter relativer Gewinn ermitteln:

$$relative\_benefit(I) = \frac{benefit(I)}{size(I)}$$

Dieser wird absteigend sortiert und über einen Greedy-Algorithmus werden solange Kandidaten für die Indexkonfiguration selektiert, bis der Speicherplatzschranke erreicht wird.

Eine entscheidende Rolle bei der Bewertung von Konfigurationen von Last-balancierten Indexen spielt die Varianz bzw. die Standardabweichung der Abfragen. Es gilt, je weiter die Abfragen über den Wertebereich des Schlüssels streuen, desto geringer ist der Nutzen eines solchen Indexes. Das Last-balancierte Tuning wird nur dort erfolgreich sein, wo die Abfragen

eng um einen bestimmten Wert streuen. Das DBMS muss aus den Abfragen $Q_1,\ldots,Q_m$ den Mittelwert μ berechnen, um daraus die Standardabweichung σ ermitteln zu können.

$$profit(Q,I) = \frac{profit(Q, I_{Last}) \cdot size(I)}{\sum_{I_j \in I_{Last}} size(I_j) \cdot \sigma(Q_1,...,Q_m)}$$

Auf diese Weise wird der Nutzen bei erhöhter Standardabweichung reduziert. Je geringer die Standardabweichung, desto geringer der schmälernde Effekt auf den Gewinn.

6.5. Mögliche Weiterentwicklungen

Ausgehend von diesem Ansatz sollte versucht werden, einen B+-Baum zu realisieren, da dies eine der am häufigsten verwendeten Indexstrukturen bei DBMS ist. Ein solcher Index könnte dann auch in ein bestehendes DBMS integriert werden. Hierfür bietet sich aufgrund seiner Quelloffenheit PostgreSQL an.

Um ein übergreifendes autonomes Self-Tuning zu bieten, können Last-balancierte Indexe in das Portfolio der Optimizer aufgenommen werden. Wird dieser mit den nötigen Informationen über die Abfragen, insbesondere Mittelwert und Standardabweichung, versorgt, können für häufig benutzte Bereiche Indexe erstellt werden, die sich variabel den Speicherplatzbedürfnissen anpassen lassen. Als Kombination mit Soft Indexen (vgl. 4.5) bieten sie ein viel feiner einstellbares Tuning-Werkzeug, als dies vollständige Indexe aufgrund ihres Ressourcenbedarf bieten können. Dem gegenüber stehen die erhöhten Aufwände bei der Reorganisation, da ein Last-balancierter Index nicht nur bei schreibenden Operation reorganisiert werden muss, sondern auch bei Select-Zugriffen. Dieser Aufwand kann jedoch durch die Wahl einer geeigneten Reorganisationsfrequenz minimiert werden.

Durch die Definition von Bedingungen durch einen Datenbankadministrator bzw. einen Anwendungsentwickler kann sich der Last-balancierte Index a priori dem sich ändernden Last-Schwerpunkt anpassen. So sind z. B. Felder wie ein Status oder ein Datum möglicherweise geeignet, eine Vorhersage zu treffen, welche Bereiche der Tabelle wie stark angesprochen werden. Ändert sich z. B. der Status in einer Tabelle „Auftrag“ auf „erledigt“, würden hier andere Balance-Werte errechnet werden.

In diesem Zusammenhang sei nochmals auf die Möglichkeit der partiellen Indexe hingewiesen, die in PostgeSQL schon verfügbar ist. Auf diese Art und Weise könnte sich der Baum beim Unterschreiten einer weiteren Balance-Schranke ganzer Teilbäume entledigen und auf diese Weise enorm Speicherplatz einsparen und mehr Tiefe für die häufiger benutzen Schlüssel bieten.

7. Implementierung in Java

Zur Implementierung wurden die von Rick Grehan [RG02] entwickelten Java-Klassen für einen B-Baum als Basis benutzt. Darauf aufbauend wurden Konzepte von Kai-Uwe Sattler, Eike Schallehn, Ingolf Geist aus [SSG05] eingebunden.

7.1. Package lbtree.pages

In diesem Paket wurden die Seitenstrukturen definiert. Dabei wurde ein Interface LBPage geschaffen, das vom Node LBNode und vom Bucket LBBucket implementiert wird.

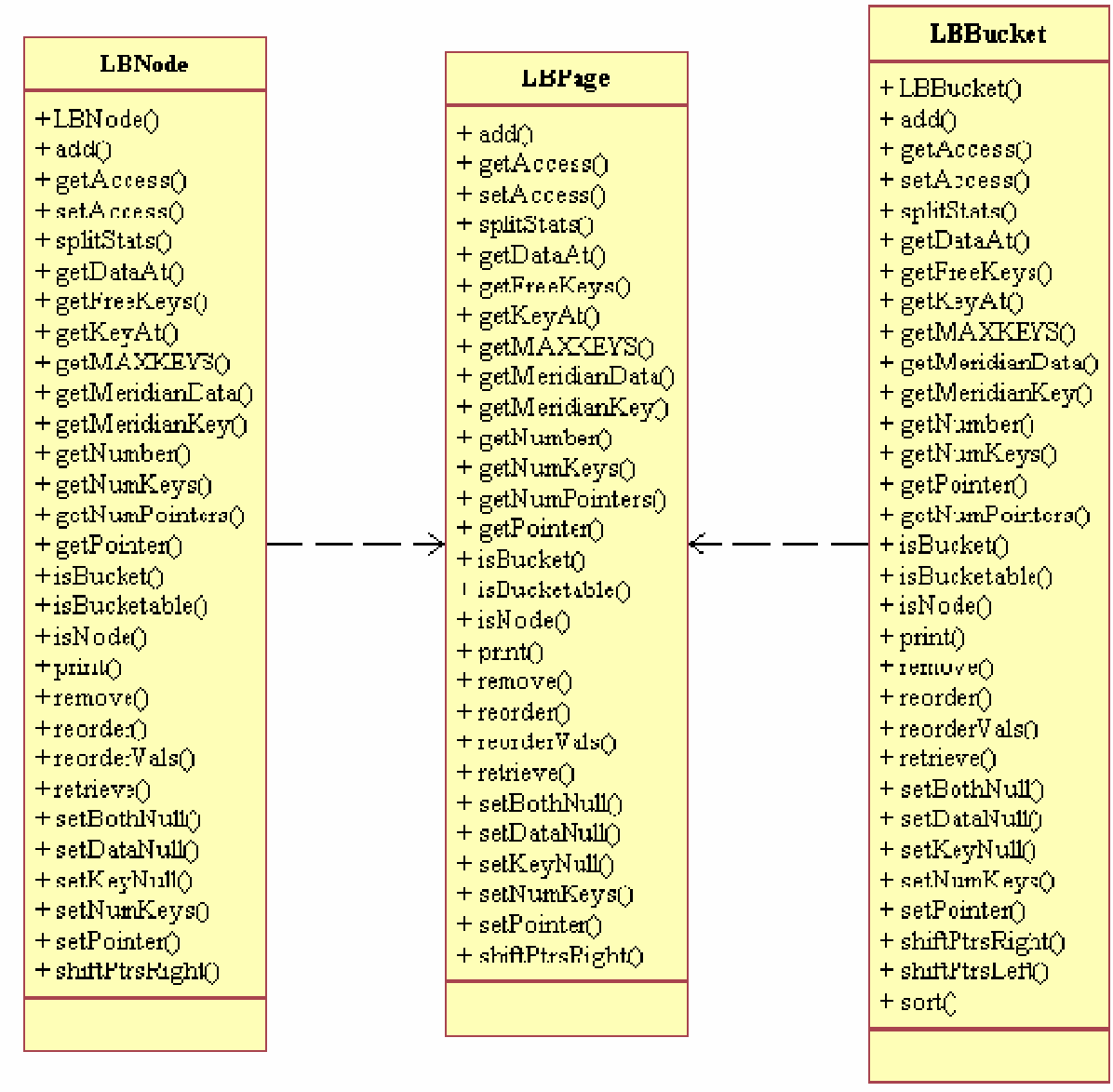


Abbildung 28: LBNode, LBBucket implements LBPage

Aus der Klasse LBNode, die ein Node implementiert, wurde eine LBBucket-Klasse erzeugt, um die Eigenschaften eines Buckets darzustellen. Um in der Implementierung nicht stets eine Unterscheidung zwischen Node und Bucket treffen zu müssen, wurde ein Interface LBPage geschaffen, das beide Knotentypenklassen implementieren.
Die Bucket-Klasse besitzt im Gegensatz zum Node eine Methode sort(). Damit können bei Bedarf die Keys innerhalb eines Buckets in eine korrekte Reihenfolge gebracht und Lücken geschlossen werden. Für die Sortierung wird auf die Funktion sort() aus der Klasse java.util.Arrays zurückgegriffen. Diese Methode wird z. B. benutzt, wenn ein Bucket geteilt wird.

7.2. Package lbtree.keys

Die Implementierung der Schlüssel wurde von Grehan übernommen und lediglich um einen Schlüsseltyp Integer erweitert:

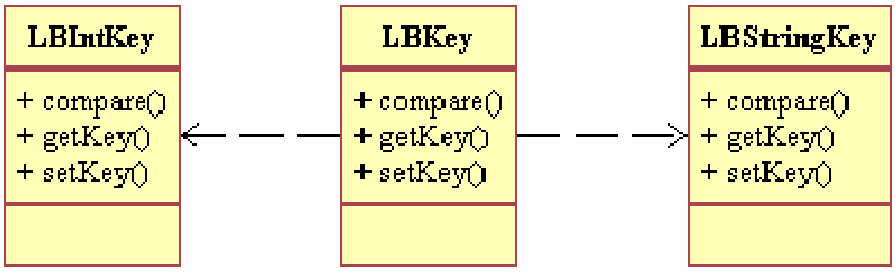


Abbildung 29: LBStringKey, LBIntKey implements LBKey

Die Schlüsselobjekte werden in Arrays der entsprechenden Seitengröße verwaltet. Sie bieten keine besonderen Methoden bis auf den Vergleich.

```
// Comparison method
public int compare (BTreeKey thatkey)     {
      int rcode=0;
      if(this.key > (Integer)thatkey.getKey())
            rcode= 1;
      if(this.key == (Integer)thatkey.getKey())
            rcode= 0;
      if(this.key < (Integer)thatkey.getKey())
            rcode= -1;
      return rcode;
}
```

Diese Methode vergleicht das Schlüsselobjekt mit einem Parameter und gibt die in der Suchmethoden von LBNode und LBBucket erwarteten Rückgabewerte (-1 = Schlüsselobjekt ist kleiner, 0 = gleich oder +1 = bei einem größeren Schlüssel) an die aufrufende retrieve-Methode zurück.

7.3. Package lbtree.tree

Im Paket lbtree.tree findet sich die Klasse LBTree, die den Baum erstellt.

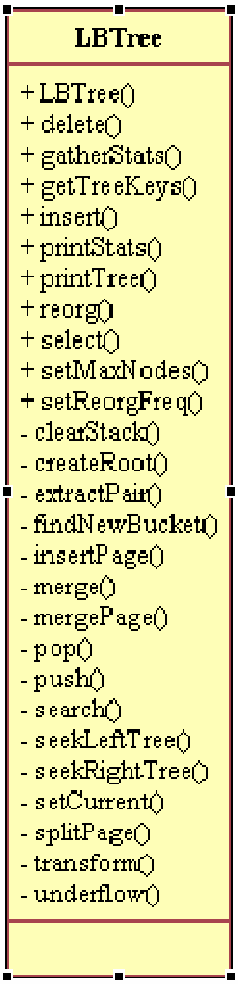


Abbildung 30: Klasse LBTree

Die wichtigsten Methoden der Klasse LBTree sind select(), insert() und delete(). Alle diesen Methoden benutzen die private Methode search() um ein Schlüsselobjekt in dem Baum zu finden. Während sich search() über die Kanten der Knoten innerhalb des Baums fortbewegt, wird gleichzeitig mit push() ein Stack gefüllt, der den Weg, den search() durch den Baum genommen hat, festhält. Dieser Stack besteht aus einem Array für Objekte der abstrakten Klasse LBPage zur Aufnahme von LBNodes und LBBuckets. Gleichzeitig wird ein Array vom Typ int gefüllt, das den besuchten Pfad beschreibt. Durch die Methode pop() kann eine Stufe zurückgegangen werden. Dies wird beispielsweise bei einem Split benutzt, um zu prüfen wie viele Nodes bei einem zur Wurzel kaskadierenden Split benötigt werden würden.

7.4. Erzeugen des Baums

Die Klasse LBDebug wurde zum Auffinden von Fehler erstellt. Mittels eines festen Arrays kann ein mit LBTest erkannter Fehler hier Schritt für Schritt nachvollzogen werden.
Zur Evaluierung wird die Klasse LBEval benutzt, der als Parameter die Reorganisationsfrequenz übergeben wird.

7.5. Einschränkungen

Die Funktion „Löschen“ ist nicht abschließend implementiert, da sie für die Versuchsanordnung wenig relevant war. Die vorhandene Funktion stammt noch aus der originalen B-Baum-Implementierung von Grehan und müsste an das Arbeiten mit Buckets angepasst werden.

7.6. Quellcode

Dieser Diplomarbeit liegt eine CD bei, auf der sich der komplette Quelltext befindet.

8. Evaluierung

8.1. Testsystem und Java-Version

Die Evaluierung erfolgte auf einem Intel Pentium-4 PC mit 2,52 GHz und 1 GB Hauptspeicher und dem Betriebssystem Windows XP.
Als Laufzeitumgebung kam hauptsächlich die Java-Version 1.6.0_03 zum Einsatz.

8.2. Ausgangssituation

Da der Ansatz nicht mit einem dichtbesetzten B-Baum mit voller Knotenzahl verglichen werden kann, wird der Entwurf der Last-Balancierung an einem LBTree mit Nodes und Buckets und einer festen Knotenzahl gemessen, der keine Last-Balancierung erhält , sondern ausschließlich durch die Daten sortiert ist.
Zunächst wurde ein Baum mit maximal 200 Knoten erzeugt, der 10.000 Werte von 0 bis 10.000 aufzunehmen hatte. Ein Node fasst in diesem Beispiel 20 Einträge, während ein Bucket 400 Schlüssel aufnehmen kann. Die Abfragen sind mit einer Abweichung von 0,15 um den Meridian 75.000 verteilt.

8.3. Vergleich

In mehreren Serien wurden die Ergebnisse von jeweils 10 Testläufen hinsichtlich der Zugriffe auf Nodes und Buckets ausgewertet. Dabei wurden die Anzahl der Nodes und Buckets und die Zahl der Reorganisationen betrachtet. Die Versuchsreihen wurden zum einen ohne Last-balancierte Reorganisation durchgeführt und zum anderen mit einer Reorganisationsfrequenz von 1.000. Diese hat sich während der gesamten Testläufe als guter Kompromiss zwischen Last-Balancierung und Aufwandsminimierung heraus gestellt. Eine häufigere Reorganisation erhöht das Risiko, dass es zu einem Thrashing kommt. Bei einer höheren Frequenz können nicht genug Nodes freigegeben werden, um häufig benutzte Knoten zu splitten.

Die folgende Versuchsreihe zeigt die Verteilung der Zugriffe von 100.000 Lookup-Operationen auf die Pages, getrennt nach Nodes und Buckets. Nach dem 100.000. Lookup wurden die Statistiken ausgegeben. Aus 10 Versuchsreihen wurde der Mittelwert gebildet, um die durch die Zufallswerte entstehenden Abweichungen zu nivellieren.

	Zugriffe			Anzahl			
	Pages	Nodes	Buckets	Nodes	Buckets	Splits	Transforms
1	370.681	281.150	89.531	200	294	491	0
2	346.846	254.871	91.975	200	256	453	0
3	342.595	253.560	89.035	200	254	451	0
4	359.277	272.050	87.227	200	292	489	0
5	366.522	277.822	88.700	200	289	486	0
6	365.039	276.875	88.164	200	261	458	0
7	362.418	274.275	88.143	200	273	470	0
8	350.153	263.895	86.258	200	274	471	0
9	373.393	279.893	93.500	200	263	460	0
10	380.723	291.311	89.412	200	273	470	0
Ø	**361.765**	**272.570**	**89.195**	**200**	**273**	**470**	**-**
Δ	38.128	37.751	7.242	-	40	40	-

Tabelle 15: Zugriffszahlen ohne Reorganisation

In jedem dieser Testläufe wurden die 200 Nodes komplett verbraucht, die überzähligen Werte wurden in die Buckets verschoben.

Im Vergleich zu den Testläufen mit Reorganisation offenbart sich der Vorteil der Last-Balancierung:

	Zugriffe			Anzahl			
	Pages	Nodes	Buckets	Nodes	Buckets	Splits	Transforms
1	361.775	269.883	91.892	189	289	490	15
2	348.109	257.219	90.890	191	269	470	13
3	378.146	284.871	93.275	189	264	466	16
4	343.909	254.147	89.762	186	294	498	21
5	344.322	253.091	91.231	194	310	513	12
6	369.908	283.202	86.706	191	268	468	12
7	353.137	268.857	84.280	193	286	489	13
8	362.620	275.002	87.618	191	274	476	14
9	358.790	270.017	88.773	193	294	496	12
10	356.126	269.617	86.509	194	291	494	12
Ø	**357.684**	**268.591**	**89.094**	**191**	**284**	**486**	**14**
Δ	34.237	31.780	8.995	8	46	47	9

Tabelle 16: Zugriffszahlen mit Reorganisation

Die Zahlen für die Buckets sind, verglichen mit den Läufen ohne Reorganisation, ungefähr auf dem gleichen Niveau. Bei den Nodes benötigt der Last-balancierte Baum weniger Zugriffe, was dafür spricht, dass er seine Schlüssel schneller erreicht.

Hinzu kommt, dass der Last-balancierte Baum dafür weniger Knoten benötigt. Die werden beim Befüllen zwar zunächst alle verbraucht, aber mit den ersten Reorganisationen werden die wenig benutzten Nodes in Buckets umgewandelt (Transforms). Dabei werden teilweise ganze Teilbäume zu verketteten Buckets. Dabei werden die Buckets jedoch nicht einfach

hintereinander gehängt, sondern der im Stack höchste Bucket wird mit dem Inhalt der tiefer liegenden befüllt. Erst wenn ein Bucket komplett befüllt ist, werden die Folgebuckets erzeugt. Im Durchschnitt wurden 14 solche Transformationen durchgeführt, der Großteil der frei werdenden Nodes wird in einem anschließenden Split wieder verbraucht.

Bei einer weiteren Testreihe wurden 10.000 Schlüssel auf nur 100 Nodes verteilt, die jeweils maximal 20 Schlüssel fassen. Die Obergrenze für ein Buckets wurde auf jeweils 500 Schlüssel festgelegt.

	Zugriffe ohne Reorg			Zugriffe mit Reorg			
	Pages	Nodes	Buckets	Pages	Nodes	Buckets	Anzahl Nodes
1	360.639	269.888	90.751	365.598	275.025	90.573	91
2	371.011	280.319	90.692	351.351	255.434	95.917	88
3	370.743	280.185	90.558	354.805	258.441	96.364	85
4	349.639	253.573	96.066	377.460	286.536	90.924	90
5	365.324	274.949	90.375	365.157	274.827	90.330	90
6	351.936	255.363	96.573	362.694	267.721	94.973	84
7	357.343	266.413	90.930	353.825	257.540	96.285	79
8	363.273	272.150	91.123	351.012	254.779	96.233	89
9	359.793	269.105	90.688	361.887	265.826	96.061	88
10	365.186	269.421	95.765	359.566	267.425	92.141	81
Ø	**361.489**	**269.137**	**92.352**	**360.336**	**266.355**	**93.980**	**87**
Δ	21.372	26.746	6.198	26.448	31.757	6.034	12

Tabelle 17: Zugriffszahlen ohne und mit Reorganisation

Auch in diesem Beispiel zeigt sich eine beinahe ausgeglichene Anzahl der Zugriffe. Bei den Läufen ohne Reorganisation wurden alle Nodes verbraucht, mit Reorganisation blieben mehr als 10% der Nodes frei.

Die Reorganisationen sind bei allen Konfigurationen nach etwa 10% der Abfragen weitestgehend abgeschlossen. Ähnlich wie beim Binärbaum pendelt sich der Last-balancierte B-Baum innerhalb der ersten 10-15% der Zugriffe auf ein stabiles Niveau ein. Lediglich vereinzelte Reorganisationen erfolgen zu späteren Zeitpunkten.

8.4. Fazit

In mehreren Testläufen wurden die hier beispielhaft ausgewählten Ergebnisse bestätigt. Dabei begünstigt, wie zu erwarten war, eine höhere Node-Grenze oder eine höherer Wert für die Maximalzahl von Schlüsseln pro Node, den datenbalancierten Baum ohne Reorganisation.

Der Nutzen liegt klar in der Einsparung von Nodes und damit Speicherplatz. Versuche mit einer höheren Abweichung von der Normalverteilung brachten ebenfalls das erwartete Ergebnis, dass der datenbalancierte Baum eine bessere Performance zeigte.

9. Zusammenfassung und Fazit

Im abschließenden Kapitel werden die Erkenntnisse noch einmal zusammengefasst und kurz dargestellt und ein Ausblick auf die weitere Entwicklung gegeben.

Das Ziel der Arbeit war die Entwicklung eines Ansatzes für eine last-balancierte Baumstruktur. Dazu wurden im zweiten Kapitel zunächst die grundlegenden Begriffe dargelegt und der bisherige Stand in kommerziellen DBMS beleuchtet. Die verschiedenen Arten von Zugriffsstrukturen wurden im anschließenden Kapitel 3 eingehend beschrieben.
Das Kapitel 4 widmete sich dem aktuellen Stand des automatisierten Datenbanktunings und erläutert die verschiedenen Ansätze, die derzeit Gegenstand der Forschung sind.
Das folgende Kapitel 5 beschäftigt sich mit der Vorarbeit zum Last-balancierten B-Baum, dem Last-balancierten Binärbaum. Aufgrund der eingeschränkten Tauglichkeit eines Binärbaums für den Einsatz in DBMS wird in Kapitel 6 der Entwurf eines Last-balancierten B-Baums untersucht und beschrieben. Davon ausgehend wurde das Konzept einer solchen Zugriffsstruktur beschrieben und ihre Wirkungsweise erklärt. Anschließend wird das Kostenmodell erarbeitet und auf die Last-Balancierung übertragen. Zum Abschluss des Kapitels werden mögliche Weiterentwicklungen aufgeführt.
Kapitel 7 erläutert grob die prototypische Implementierung in Java, während in Kapitel 8 die Ergebnisse einiger Testläufe untersucht werden.

Das Feintuning von Indexen auf der Ebene des Index selbst ist ein viel versprechender Ansatz für ein automatisches Datenbanktuning. Der hier vorgestellte und teilweise implementierte Ansatz verfügt noch über Erweiterungspotential. So muss das „Merge“ noch in das Autotuning eingebunden werden. Dazu ist es notwendig jeweils vorausschauend zu agieren, d. h. bevor mit einem Tuning begonnen werden kann, muss geprüft werden, ob die Maßnahme gegebenenfalls auch bis zur Wurzel durchgeführt werden kann. Hier kann es sinnvoll sein, die Grenze der erlaubten Nodes kurzfristig zu überschreiten, um anschließend einen Kandidaten mit weiter gefassten Bedingungen oder als Ausnahmebehandlung in ein Bucket umzuwandeln.

Nur in Ansätzen implementiert ist der Weg aus einem Bucket zu einem Node. Sollte sich die Zugriffsverteilung während der Abfragen merklich ändern, müssten für in Buckets ausgelagerte Teilbäume Mechanismen gefunden werden, die es erlauben wieder einen

balancierten Baum zu erstellen. Hierfür ist jedoch ein höherer Aufwand anzusetzen, da teilweise mehrere hundert verkettete Buckets nach einem Median durchsucht werden müssten. Daher ist hier gleichzeitig die Frage zu untersuchen, ob ein solches Vorgehen im Hinblick auf die Performance tolerierbar ist.

Offen ist ebenfalls die Implementierung eines Prototyps in ein DBMS, um den Ansatz unter realen Umständen einer Prüfung unterziehen zu können. Im hier vorliegenden Java-Programm kann ein Festplattenzugriff nur über den Umweg des Schreibens in eine Datei simuliert werden. Die Relevanz und Aussagekraft eines solchen Versuchs ist aus meiner Sicht nicht gegeben.

Abschließend sollte darüber nachgedacht werden, bestimmte Bereiche ganz aus dem Index zu verbannen und diese Information in der Wurzel zu speichern. So könnten die Zugriffsstatistiken, die derzeit in jedem Knoten separat gespeichert werden, in der Wurzel abgelegt werden.

Weiterhin sind die Gründe zu analysieren, warum der Vorteil geringer ausfällt als erwartet. Wie im vorherigen Kapitel bereits angedeutet, kann es daran liegen, dass die Werte mit einer geringeren Zugriffshäufigkeit am linken und rechten Rand des Baums in lange Ketten von Buckets abgeschoben wurden. Erfolgt nun dennoch ein Zugriff, müssen u. U. alle Buckets nacheinander durchsucht werden, um festzustellen, ob ein Wert enthalten ist oder nicht.

Es bleibt zu prüfen, ob der hier entwickelte Ansatz in einem DBMS realisiert wird, um weitreichende Untersuchungen über sein Verhalten in verschiedenen praktischen Anwendungssituationen durchzuführen. Der Ansatz selbst birgt viel Potential und kann, wenn er auf den Datenbanktabellen mit einem passenden Nutzungsprofil angewendet wird, einen großen Beitrag zum autonomen Self-Tuning liefern.

Literaturverzeichnis

- [2007a] http://www.postgresql.org/about/history, PostgreSQL History, 16.06.2007
- [Au06] J. Auer, Dritte Normalform: Keine transitive Abhängigkeit, http://www.sql-und-xml.de/sql-tutorial/dritte-normalform-transitive-abhaengigkeit.html, 31.12.2006, abgerufen am 16.06.2007
- [BAC99] M. W. Blasgen, M. M. Astrahan, D. D. Chamberlin, J. N. Grey, W. F. King, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. Putzolu, M. Schkolnick, P. G. Selinger, D. R. Slutz, H. R. Strong, I. L. Traiger, B. W. Wade, R. A. Yost, System R: An architectural overview, IBM Systems Journal, Volume 38, Number 2/3, Page 375, 1999
- [Bo03] C. Bohn, Ein Vergleich zwischen PostreSQL und Oracle Database, Universität Mannheim, Lehrstuhl für Wirtschaftsinformatik III, 2003
- [Co90] E.F. Codd, The Relational Model for Data Base Management (Version 2), Addison-Wesley, 1990
- [CW05] S. Chaudhuri, G. Weikum, Foundations of Automated Database Tuning, SIGMOD, 2005
- [Eh05] H.-D. Ehrich, Entwurf von Datenbanken, Vorlesungsskript, http://www.ifis.cs.tu-bs.de/html_d/skripte/entw-a4.pdf, 26.06.2007
- [FNP79] R. Faging, J. Nievergelt, N. Pippenger, J. Strong, Extensible Hashing – A Fast Access Method for Dynamic Files, ACM Transactions on Database Systems, Bd 4, Nr. 3, S. 315–344, 1979
- [IBM04] An architectural blueprint for autonomic computing, WhitePaper 2004, http://www-03.ibm.com/autonomic/pdfs/ACBP2_2004-10-04.pdf, 21.06.2007
- [JH05] J. Hoffmeister, MaxDB – The Professional(’s) Database, Präsentation MySQL Users Conference, 2005
- [Ka04] F. Kalis, InsideSQL.de – Codds 12 Regeln: http://www.insidesql.de/beitraege/theorie_und_grundlagen/codds_12_regeln.html, 13.07.2004, abgerufen am 13.07.2007
- [Ke98] A. Kelz, Relationale Datenbanken, http://v.hdm-stuttgart.de/~riekert/lehre/db-kelz/, 16.11.1998, abgerufen am 16.06.2007

- [KPP04] H. Kellerer, U. Pferschy, D. Pisinger, Knapsack Problems, Springer-verlag Berlin, Heidelberg, 2004
- [Li06] S. Lightstone, Foundations of Autonomic Computing Development, IEEE TFAAS Newsletter – Issue 3, 2006
- [LSS07] M. Lühring, K. Sattler, E. Schallehn, K. Schmidt, Autonomes Index Tuning - DBMS-integrierte Verwaltung von Soft Indexen, BTW 2007 S. 152-171, 2007
- [Lu03] J. Lufter, Objektrelationale Datenbanksysteme, Informatiklexikon Gesellschaft für Informatik e.V. http://www.gi-ev.de/service/informatiklexikon/informatiklexikon-detailansicht/meldung/55/, abgerufen am 14.06.2007
- [Lü07] Andreas Lübcke, Self-Tuning-Konzepte zur Verwaltung von Bitmap-Index-Konfigurationen, Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Institut für technische und betriebliche Informationssysteme, 2007
- [Mi05] B. Miller, The autonomic computing edge: The role of knowledge in autonomic systems, IBM developerWorks 2005, http://www.ibm.com/developerworks/autonomic/library/ac-edge6, 21.06.2007
- [MKF03] J.-E. Michels, K. Kulkarni, C. M. Farrar, A. Eisenberg, N. Mattos, H. Darwen, The SQL Standard, it - Information Technology, 45(1), S. 30-38, 2003
- [MS04] R. Möller, J.W. Schmidt, Einführung in Datenbanksysteme http://www.sts.tu-harburg.de/~r.f.moeller/lectures/db-ws-04-05/31-RDM-2.pdf, abgerufen am 16.06.2007
- [MS05] http://www.microsoft.com/sql/prodinfo/compare/oracle/ss2005oracle10gnetdev.mspx, Comparing SQL Server 2005 and Oracle 10g as a Database Platform for Microsoft .NET Developers, Microsoft Technical White Paper, 27.07.2005
- [RG02] R. Grehan, How to Climb a B-tree, http://www.ftponline.com/javapro/2002_01/magazine/features/rgrehan/, JavaPro Magazine 01/2002
- [SAP05] http://dev.mysql.com/doc/maxdb/pdf/whitepaper.pdf, MaxDB Whitepaper, 01.02.2005, Version 1.0
- [Sc06] K. Schmidt, Automatische Erstellung von Soft-Indexen in PostgreSQL, Diplomarbeit an der Technische Universität Ilmenau, 2006
- [SGS03] K. Sattler, I. Geist, E.Schallehn, QUIET: Continuous Query-driven Index Tuning, Proceedings of the 29th VLDB Conference Berlin S. 1129-1132, 2003

- [SSG04] K. Sattler, E. Schallehn, I. Geist, Autonomous Query-driven Index Tuning, Proc. Int. Database Engineering and Applications Symposium (IDEAS 2004) in Coimbra, Portugal, S. 439–448, 2004
- [SH99] G. Saake, A. Heuer, Datenbanken: Implementierungstechniken, MITP-Verlag, 1999
- [SS01] G. Saake; K. Sattler, Algorithmen und Datenstrukturen – Eine Einführung mit Java,. dpunkt.verlag, 2001
- [SSG05] K. Sattler, E. Schallehn, I. Geist, Towards Indexing Schemes for Self-Tuning DBMS, ICDE Workshops 2005, S. 1216, 2005
- [St06] P. Stadler, Algorithmen und Datenstrukturen 1, Vorlesungsskript, http://www.bioinf.uni-leipzig.de/Leere/WS0607/ADS1/ADS1-10.pdf, 26.06.2007
- [St89] M. Stonebraker, The case for partial indexes, SIGMOD Rec. 18, 4, 1989
- [Te04] R. Telford, The State of Autonomic Computing Today, IBM developerWorks 2004, http://www.ibm.com/developerworks/autonomic/library/ac-telford, 21.06.2007
- [VM01] O. Vornberger, O. Müller, Datenbanksysteme - Vorlesung im SS 2001, http://www-lehre.inf.uos.de/~dbs/2001/Postscript/skript.pdf, 26.06.2007
- [WMH02] G. Weikum, A. Mönkeberg, C. Hasse, P. Zabback, Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering, Proceedings of the 28th VLDB Conference, 2002