Beleg
–
Maschinennahe Programmierung C

Thema:
Datenbankindeximplementierung
auf Basis eines B(ayer)-Baumes

Student : Stefan Pohl
Professor : Prof. J. Geiler
Datum : 18.03.2001

Stefan Pohl
spohl1@htwm.de

Diese Seite wird nicht mitgedruckt

Inhaltsverzeichnis

Stefan Pohl
spohl1@htwm.de

1. Einleitung

Heutzutage ist die Informationsflut so groß geworden, daß es ohne entsprechenden Mehraufwand nicht mehr möglich ist, gezielt Informationen zu suchen und vor allem zu finden, ein Tribut der Informationsgesellschaft.
Es gilt, geeignete Techniken und Hilfsmittel zu finden, um den Daten Herr zu werden.
Um die Realisierung von effektiven, zukünftig erweiterbaren Algorithmen zu gewährleisten, kapselt man die gesamte Funktionälität in Datenbanken. Diese bestehen einerseits aus den ursprünglichen Daten und andererseits aus diversen redundanten Datenstrukturen. Die Schnittstelle, die Datenbanken bereitstellen, beschränkt sich zumindest auf die primitivsten Funktionen zur Arbeit mit Daten: Hinzufügen, Ändern, Löschen, Abfragen. Das Ziel ist es, diese Funktionen möglichst schnell auszuführen, wie das geschieht, kann und soll dem Benutzer egal sein, dem interessierten Informatiker, vielleicht dem Programmierer eines solchen Retrieval-Systems, aber nicht.
Man bedient sich der Indexe, um die Daten oder Verweise auf sie nach verschiedenen Gesichtspunkten zu sortieren, um ein effizientes Suchen nach ihnen zu ermöglichen. Gängige Datenbanken benutzen hierfür die Datenstruktur des B-Baumes, der nach seinem Erfinder Rudolf Bayer benannt ist. Dieser Baum, zusammen mit den auf ihn ausführbaren Funktionen, hat alle Eigenschaften, die Datenbanken verlangen: Schneller, zeitlich nach oben hin abgrenzbarer, Zugriff auf alle Elemente der Datenbasis.

2. Antrieb

Eine effektive Implementierung kann nur erreicht werden, wenn die verwendete Programmiersprache möglichst hardwarenah ist und einen optimierenden Compiler besitzt. C ist deshalb prädestiniert als Programmiersprache für die Implementierung eines in der Ausführung aufwendigeren Algorithmus.
Deshalb war es mir ein Anliegen, statt eines Programms mit viel Oberfläche und wenig Know-How, mich an der Implementierung eines Datenbankindex zu versuchen. Oberflächen sind in anderen Umgebungen leichter und besser zu programmieren, welche ja meistens auch wieder Schnittstellen zu „shared objects" bzw. „dynamic link libraries" zur Verfügung stellen.
Im Sinne von dem unter Unix bekannten Baukastensystem, soll das Ziel dieses Beleges ein kommandozeilenorientiertes Programm sein, daß lediglich das Management eines Index erlaubt, da dieser trotz logischer Redundanz nebst den Daten der essentielle Bestandteil einer Datenbank ist, und das Programm somit unabhängig von Formaten der jeweiligen Datenbasis bleibt. Um dieses Ziel zu erreichen, war die Schaffung weiterer Programme zum Test der hier vorgestellten B-Baum-Implementierung notwendig.
Die Wahl des Betriebssystems fiel auf Linux, da hier über die Makefiles ein effektiver Apparat existiert, um auf die Kompilierung Einfluß zu nehmen. Hinzu kommt, daß die sofortige Kompilierung auf jedem Unix möglich ist, da überall der Gnu-C-Compiler vorhanden ist.

3. Installation

Die Benutzung der in diesem Beleg vorgestellten Programme setzt eine kurze Installation voraus. Die beiliegende CD enthält folgende Verzeichnisse und Dateien:

Verzeichnis/Datei	Beschreibung
db/	Beinhaltet Datenbasis und einen bereits erstellten Index über Nachnamen
doc/	Beinhaltet Dokumentation

Verzeichnis/Datei	Beschreibung
runimage/	Beinhaltet sofort lauffähige Skripte zur Benutzung des Index und der Datenbasis in db/
runimage.tgz	Komprimierte Version des gleichnamigen Verzeichnisses
src/	Beinhaltet die Quelltexte der C-Programme
src.tgz	Komprimierte Version des gleichnamigen Verzeichnisses

Die Datei "runimage.tgz" beinhaltet sämtliche zum Beleg gehörenden ausführbaren Dateien und kann leicht mit "tar –xvzf /cdrom/runimage.tgz" (evtl. Pfad anpassen) ins aktuelle Verzeichnis ausgepackt werden. Diese Vorgehensweise ist anzuraten, da hierdurch die Ausführungsrechte automatisch richtig gesetzt werden.
Die Benutzung der auf der CD im Verzeichnis "runimage" untergebrachten direkt lauffähigen Version setzt das Ausführungsrecht für alle Dateien der gemounteten CD voraus.
Nach erfolgreicher Installation liegen alle Programme bzw. Skripte für die weitere Verwendung vor. Je nach Plattform, auf der das Programm ausgeführt werden soll, ist eine Neukompilierung aus den Quellen vorzunehmen.
Da der Funktionsumfang des im Beleg vorgestellten Programms "index" begrenzt ist, war es unerläßlich, eine geeignete Umgebung zu schaffen, die das Programm durch geeignete Kapselung benutzt. Diese beinhaltet einerseits eine Datenbasis, die als Beispiel herhalten soll. Um die Alltagstauglichkeit zu zeigen, wurde dabei auf eine veraltete Version eines Telefonbuchs von Sachsen zurückgegriffen. Es steht in einer kommaseparierten Datei namens "telsachsen.csv". Andererseits wurden eine Vielzahl Shellskripts benutzt, die die Eingaben für das Programm vorformatieren und die Ausgaben auch wieder auf die Datenbasis anwenden.

4. Bedienungsanleitung:

Der Beleg besteht aus dem Hauptprogramm "index", mehreren Testprogrammen und Shellskripts.
Um die Funktionalität anhand eines Beispiels zu sehen, braucht man nur die Skripte beginnend mit "idx_" auszuführen.
Alle ausführbaren Programme sollen im folgenden besprochen werden.

4.1. index

Das Programm ist als Unix-Baukasten-Tool konzipiert und arbeitet somit wie ein Filter. Es bietet seine Dienste über Parameter dem Benutzer an.
"Index –h" liefert die Hilfe, welche alle relevanten Informationen zur Benutzung des Programms beinhalten sollte:

```
index - Verwaltung eines Index zu einer Datenbasis

Das Programm arbeitet auf Basis von Stringvergleichen zwischen den Schlüsseln.
Dadurch ist es gleichzeitig möglich nach Strings, Werten sowie binären Werten zu
sortieren.

Syntax:
index <action> [options]

Aktionen:
-h, --help      Gibt eine kurze Beschreibung des Programms aus
-c, --create    Erstellt initial einen Index aus Schlüssel-Werte-Paaren an der
                Standardeingabe
```

```
-a, --add       Fügt dem Index Schlüssel-Werte-Paare an der Standardeingabe
                hinzu
-d, --delete    Löscht Schlüssel aus dem Index, die über die Standardeingabe
                übergeben werden. Hier dürfen nur die Schlüssel übergeben werden
-r, --range     Führt eine Bereichsabfrage über den Index aus. Dazu werden
                zwei Schlüssel der Standardeingabe entnommen.
-l, --list      Gibt die Eigenschaften des Index aus. Dazu gehören die
                Definition der Schlüssel- und Wertlängen, die Ordnung und die
                Anzahl der Schlüssel.

Optionen:
-f, --file      Gibt die Datei des zu verwaltenden Indexes an
-p, --parse     Über diese Option werden die Schlüssel- und Wertelängen
                angegeben, die über die Standardeingabe eingelesen werden
                sollen. Diese Option wird nur für -c benötigt, da bei einem
                einmal erstellten Index der Aufbau der Schlüssel nicht mehr
                geändert werden kann.
                z.B. -p 5,10 für 5 Byte Schlüssel und 10 Byte Wert
-o, --order     Gibt die Ordnung an, in der der Indexbaum aufgebaut werden soll
                Diese Option hat ebenfalls nur beim Erstellen des Index eine
                Wirkung. Default: 16
-n, --newline   Gibt an, wieviel Byte zwischen den Schlüssel-Werte-Paaren im
                Eingabestrom überlesen werden sollen. Default: 0
-v, --verbose   Ausführliche Ausgabe (Abschließende Statusmeldung)
-vv, --verbosev Ausführlichere Ausgabe (zusätzlich Ausgabe des akt. Schlüssels)
```

Diese Hilfe sollte ausreichen, um mit dem Programm arbeiten zu können. Bei Unklarheiten verweise ich auf die technische Referenz, die die inhaltlichen Zusammenhänge zusammenfäßt.

4.2. Die Testprogramme

Die Testprogramme dienen der Validitätsprüfung einer B-Baum-Implementierung und arbeiten sowohl im Speicher wie auch in einer Datei. Das ist der Grund, warum beim Kompilieren mehr als 2 ausführbare Dateien entstehen. Den jeweiligen Programmen, die in einer Datei arbeiten, kann beim Start als Parameter die zu erstellende Datei übergeben werden. Ansonsten wird eine Datei im aktuellen Verzeichnis erstellt.

4.2.1. randombtreefile/-mem

Dieses Programm dient dem automatischen Hinzufügen und darauffolgenden Löschen aller Schlüssel in zufälliger Reihenfolge. Als Schlüssel werden Integer verwendet.
Nach dem Start werden Sie zunächst zur Eingabe der Ordnung aufgefordert, in der der Baum aufgebaut werden soll. Danach können Sie die Anzahl der Elemente spezifizieren, die hinzugefügt werden soll.
Sie können nun den Fortschritt der Operationen verfolgen und sollten am Ende wieder einen leeren Baum mit der Höhe –1 erhalten. Ist das nicht der Fall, wird zum Debugging eine proprietäre Ausgabe des nun noch vorhandenen Baumes vorgenommen.
Sie können das Programm beliebig oft wiederholen.

4.2.2. testbtreefile/-mem

Dieses Programm dient dem manuellen, inkrementellen Hinzufügen und Löschen von beliebigen Elementen. Als Elemente werden wieder Integer verwendet.
Die Ordnung des B-Baumes wurde hier direkt im Programm auf 1 festgelegt, da eine Überwachung der Aktionen sowieso nur sinnvoll mit kleinsten Ordnungen möglich ist.
Das Programm fragt sie ständig, was es mit dem Baum tun soll. Hierzu können Sie ‘a’ (Hinzufügen), ‘d’ (löschen) oder ‘q’ (Abfragen) eingeben. Direkt danach geben Sie nun den für den Schlüssel gewünschten Wert ein. Da für einen Abfragebereich noch ein 2. Wert erforderlich

ist, wird dieser gesondert nachgefragt. Nach jeder Aktion gibt das Programm den momentanen Zustand des Baumes in proprietärer Form aus.

4.3. Shellskripte

Diese Skripte dienen der einfachen Benutzung des Programms “index”. Dazu sind die meisten Parameter standardmäßig auf die Benutzung der auf der CD vorhandenen Beispieldatenbasis zugeschnitten. Diese können aber jederzeit per Kommandozeile “überschrieben” werden.

4.3.1. idx_create

Dieses Skript erstellt einen Index unter Zuhilfenahme des in diesem Beleg vorgestellten Programms “index”. Dazu selektiert es eine Spalte einer kommaseparierten Datei, bringt sie auf eine gewisse feste Länge, benutzt sie als Schlüssel für einen Index und übergibt als Wert des Schlüssels die Position des Datensatzes in der CSV(Comma-Separated-Values)-Datei an. Der Wert wird dabei nicht als String übergeben, sondern binär.
Bis auf die Angabe der CSV-Datei können alle weiteren Parameter entfallen, da sie im Skript auf Standardwerte gesetzt wurden. Hier ist unter anderem festgelegt, daß der auf der CD vorhandene Index über die 2. Spalte der kommaseparierten Datei, den Nachnamen, erstellt wurde. Das ist natürlich jederzeit im Skript auf neue Umgebungen zuschneidbar, ein Grund für die Benutzung von Shellskripts.

4.3.2. idx_query

Dieses Skript fragt einen Schlüssel von einem Index ab. Dazu muß mindestens dieser Schlüssel per Standardeingabe übergeben werden. Wichtig dabei ist, daß der übergebene Schlüssel die gleiche Länge hat, wie in dem erstellten Index (Über index –l –f /cdrom/db/telsachsen.csv.idx leicht in Erfahrung zu bringen.).

4.3.3. idx_queryrange

Dieses Skript fragt einen Schlüsselbereich von einem Index ab. Es ist lediglich die verallgemeinerte Version von “idx_query”. Dazu müssen mindestens zwei Schlüssel per Standardeingabe übergeben werden. Wichtig dabei: Beide übergebene Schlüssel haben die gleiche Länge, wie in dem erstellten Index und folgen direkt aufeinander (Leicht durch Änderung des Skripts änderbar).

4.3.4. Sonstige

Alle weiteren vorhandenen Skripte werden entweder von den für einen Anwender interessanten Skripte beginnend mit “idx_” benutzt oder können für eine Migration auf eine andere Datenbasis von Nutzen sein.
Alle weiteren ausführbaren Programme/Skripte sind durch Hilfeausgaben selbsterklärend und im Quelltext dokumentiert.

5. Technische Referenz

5.1. Einstieg

Das Programm “index” besteht einerseits aus den Algorithmen zur Verwaltung eines B-Baumes und andererseits aus einem Rahmen, der die gewünschten Aktionen, die auf der Kommandozeile übergeben wurden, auf den Baum anwendet.
Hier wird schon klar, daß die Implementierung eines B-Baumes als zweckmäßigerweise wiederverwendbare Bibliothek vorzunehmen ist. Diese Bibliothek kann somit als Unterprojekt aufgefaßt werden, weshalb sie auch ihr eigenes Verzeichnis spendiert bekommt.
Die Entwicklung und die Endprüfung der verschiedenen Funktionen erforderte die Schaffung verschiedener Testwerkzeuge zum manuellen und automatischen Aufruf der Bibliotheksfunktionen. Als solche nicht direkt zur Bibliothek gehörende Quellcodes werden sie in ein Unterverzeichnis dieser Bibliothek verbannt.
Die für einen Benutzer relevanten Funktionen exportiert „btree.c” bzw. „btree.h”.
Schnell erschloß sich die Notwendigkeit der Trennung von Inhalt und Form, also den Algorithmen zur Verwendung des Baumes und der physikalischen Repräsentation desselben. Die physikalische Darstellung des Baumes soll im folgenden auch mit Underlying bezeichnet werden. Diese strikte Trennung findet ihren Ausdruck in der Schnittstelle „node.h”. Hier werden alle für ein entsprechendes Underlying zu implementierende Funktionen aufgeführt.
Aufgrund der Einfachheit stellt die erste Implementierung dynamische Speicherallokation („nodemem.c”, „nodemem.h”) dar, da man sich hier um keine Serialisierung des Baumes kümmern muß. Die Geschwindigkeit erlaubt Tests mit Millionen von Schlüsseln in Minuten.
Da ein Speicherabbild aber irgendwann serialisiert werden muß, kommt man auf den Gedanken, den Baum in einer Datei aufzubauen, da hier auch kein Einlesen des gesamten Index vor seiner Benutzung notwendig ist. „nodefile.c” bzw. „nodefile.h” stellen ein ensprechendes Underlying bereit.
Hier stellt sich nun das Problem der Serialisierung. Um das Projekt voranzubringen, ist die Entscheidung „zugunsten” von Einfachheit (jeder neue Knoten ans Ende der Datei) und Fragmentation (nur logisches Löschen von Knoten) gefallen, da eine spätere Verbesserung dieser Algorithmik jederzeit möglich ist. Dieses „inkrementelle” Programmieren wurde in dem gesamten Projekt angewandt, z.B. auch bei der Wahl des Suchalgorithmus von Schlüsseln innerhalb von Knoten. Hier kann die Linearsuche bei kleinen Ordnungen der Binärsuche einiges voraus haben, genauso wie Versuche gezeigt haben, daß Binärsuche ohne Abbruch der Suchschleife schneller waren als bei Abbruch bei gefundenem Schlüssel (!). Das gilt erstaunlicherweise bis zu gebräuchlichen Ordnungen wie 32.
Die Entscheidung, mit welcher physikalischen Repräsentation der Baum und damit der Index letztlich arbeitet, nimmt man einfach durch entsprechendes Linken und Setzen bzw. nicht-Setzen der Präprozessorvariablen „_NODEFILE” vor.
Die Testprogramme verwenden diesen Ansatz und arbeiten so in beiden Varianten. Die Makefiles erstellen aus den beiden Testprogramm-Quelltexten dadurch 4 verschiedene Programme.

Alle weiteren Internas sind, denke ich, durch die Quelldateien erklärt.

5.2. Der B-Baum

Ein B-Baum besteht wie alle Bäume aus miteinander unzyklisch verzeigerten Knoten gleicher Größe. Anders als bei binären Bäumen, kann ein Knoten mehrere Schlüssel beinhalten und somit mehr als 2 Kinder haben. Als Maß hierfür definiert man die Ordnung eines Baumes.
An die Struktur eines B-Baum wird nur eine Bedingung geknüpft: Jeder Knoten muß mindestens halb voll sein, von der Wurzel einmal abgesehen.
Der kleinste B-Baum hat die Ordnung O=1, somit beinhaltet jeder Knoten mindestens O=1 Schlüssel. Maximal kann ein Knoten 2*O Schlüssel beinhalten. Zeiger bzw. Verweise geben Knoten an, deren Schlüssel zwischen den benachbarten Schlüsseln liegen. Somit hat ein Knoten mindestens O+1 und maximal 2*O+1 Verweise.
Soviel zum Aufbau, interessant wird aber erst die Wahl der Verfahren, um Schlüssel zu finden, hinzuzufügen und zu löschen. Um einen Schlüssel zu finden, bedient man sich der Verfahren für binäre Bäume.
Da eine Datenbasis normalerweise mehr wächst als sie kleiner wird, ist es sinnvoll, für das Hinzufügen einen schnellen Algorithmus zu benutzen, der auf Kosten der Ausführungszeit zum Entfernen arbeitet. Hierzu wandert man im Baum zu dem Blatt hinab, in das der Schlüssel eingefügt werden muß. Dort fügt man es an der richtigen Stelle ein. Ist das Blatt bereits voll, fügt man ein, indem man das Blatt in 2 halbvolle Blätter aufteilt und den mittleren Schlüssel aufsteigen läßt und dort versucht, diesen Schlüssel einzufügen. Dieser Vorgang setzt sich schlimmstenfalls bis zur Wurzel fort, wo bei Bedarf eine neue Wurzel entsteht. Die Höhe dieses B-Baumes wächst also generell an der Wurzel, wodurch bereits automatisch eine vollständige Balanciertheit sowie ein vollständiger Baum entsteht.
Beim Löschen wird es schwieriger, da zu löschende Schlüssel sich nicht nur in Blättern befinden können, da sie inzwischen aufgestiegen sein können. Man sucht den zu löschenden Schlüssel, ersetzt ihn mit dem am weitesten rechts liegenden Schlüssel unterhalb des linken Sohnes. Hierdurch wird ein Schlüssel eines Blattes entfernt, wodurch die B-Baum-Eigenschaft zerstört werden kann. Deshalb wird von diesem Blatt aufsteigend die Eigenschaft geprüft, ggf. Knoten ausgeglichen oder Knoten zusammengefügt, was zum Verlust eines Schlüssel des darüberliegenden Knotens führt. Dieser Vorgang setzt sich auch im schlimmsten Fall bis zur Wurzel fort, weshalb der Baum in seiner Höhe auch wieder nur an seiner Wurzel schrumpfen kann.
Diese Vorgänge finden in ihren Funktionen ihre Entsprechung.
Höhere Ordnungen bedeuten flachere Bäume, somit „logarithmischeres" Zeitverhalten bei der Traversierung im Baum, aber auch breitere Knoten und langsameres Suchen innerhalb dieser. Man muß also, wie so oft, einen Kompromiß schließen. In der Praxis erschwert die Wahl des Underlying: Die Indexdatei auf der Festplatte besitzt zwar sogenannten „direkten" Zugriff, die Suchzeit beträgt aber ein Vielfaches des Lesens von sequentiellen Daten. Hierdurch sind große Ordnungen in der Geschwindigkeit begünstigt. Aber auch nur je nachdem, wie groß die einzelnen Schlüssel und Schlüsselinhalte (Werte) sind: Wird der Index z.B. als Primärindex genutzt und alle Daten anstelle von Verweisen auf die Daten der Datenbank in diesem Index gespeichert, vermindert sich der Vorteil von sequentiellem Lesen.
Aus diesen Gründen muß effektiverweise die Wahl der Ordnung, in der der Baum bzw. der Index aufgebaut werden soll, beim Benutzer liegen.

5.3. B-Baum-Transformationen

Folgende Illustrationen sollen die Baum-Transformationen verständlicher machen:

5.3.1. node_insertkey

Diese Funktion fügt einen Schlüssel in einen noch nicht vollen Knoten ein

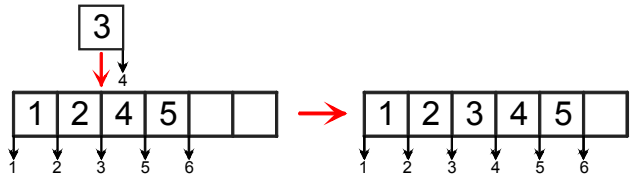

Abbildung 1

5.3.2. node_insertkey_split

Diese Funktion fügt durch Aufteilen der Schlüssel in 2 übergebene Knoten einen Schlüssel in einen vollen Knoten ein

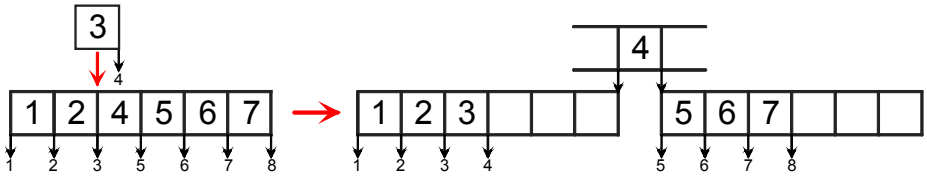

Abbildung 2

5.3.3. node_delkey

Diese Funktion löscht einen Schlüssel und sein rechtes Kind aus einem Knoten

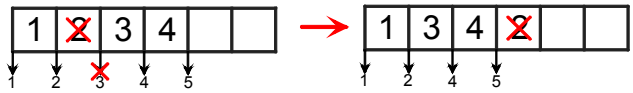

Abbildung 3

5.3.4. node_rotateleft

Diese Funktion rotiert einen Schlüssel über seinen Vaterschlüssel um eine Stelle nach links

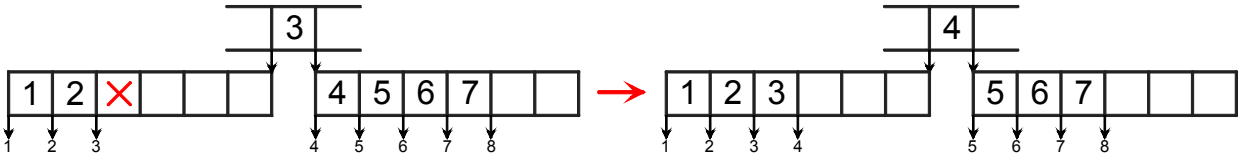

Abbildung 4

5.3.5. node_rotateright

Diese Funktion rotiert einen Schlüssel über seinen Vaterschlüssel um eine Stelle nach rechts

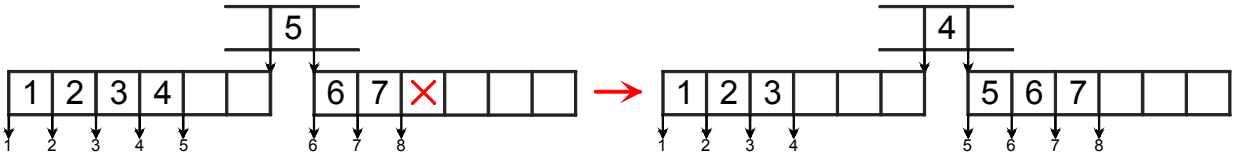

Abbildung 5

5.3.6. node_merge

Diese Funktion vereinigt zwei Knoten mit seinem Vaterschlüssel

Stefan Pohl
spohl1@htwm.de

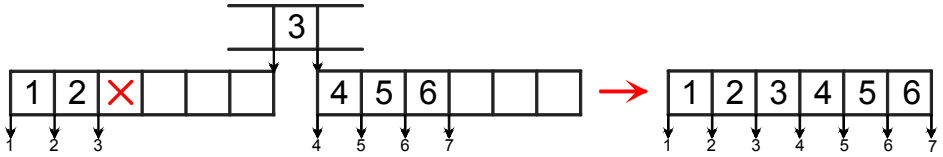

Abbildung 6

5.4. B-Baum-Schlüssel-Kapazitäten

B-Bäume sind zwar vollständig und balanciert, man bezahlt diesen Umstand aber durch eine vergleichsweise geringe Auslastung des zur Verfügung stehenden Platzes in jedem Knoten. Das liegt ganz daran, wie die hinzuzufügenden und zu löschenden Schlüssel verteilt sind.
Folgende Tabellen sollen auf diesen Umstand aufmerksam machen:

Ordnung: 1		
Höhe	minimal	maximal
0	1	2
1	3	8
2	7	26
3	15	80
4	31	242
5	63	728
6	127	2.186
7	255	6.560
8	511	19.682
9	1.023	59.048
10	2.047	177.146
11	4.095	531.440
12	8.191	1.594.322
13	16.383	4.782.968
14	32.767	14.348.906
15	65.535	43.046.720
16	131.071	129.140.162
17	262.143	387.420.488
18	524.287	1.162.261.466
19	1.048.575	3.486.784.400
20	2.097.151	10.460.353.202
21	4.194.303	31.381.059.608

Ordnung: 2		
Höhe	minimal	maximal
0	1	4
1	5	24
2	17	124
3	53	624
4	161	3.124
5	485	15.624
6	1.457	78.124
7	4.373	390.624
8	13.121	1.953.124
9	39.365	9.765.624
10	118.097	48.828.124
11	354.293	244.140.624
12	1.062.881	1.220.703.124
13	3.188.645	6.103.515.624
14	9.565.937	30.517.578.124

Ordnung: 3		
Höhe	minimal	maximal
0	1	6
1	7	48
2	31	342
3	127	2.400
4	511	16.806
5	2.047	117.648
6	8.191	823.542
7	32.767	5.764.800
8	131.071	40.353.606
9	524.287	282.475.248
10	2.097.151	1.977.326.742
11	8.388.607	13.841.287.200

Ordnung: 5		
Höhe	minimal	maximal
0	1	10
1	11	120
2	71	1.330
3	431	14.640
4	2.591	161.050
5	15.551	1.771.560
6	93.311	19.487.170
7	559.871	214.358.880
8	3.359.231	2.357.947.690
9	20.155.391	25.937.424.600

Ordnung: 10		
Höhe	minimal	maximal
0	1	20
1	21	440
2	241	9.260
3	2.661	194.480
4	29.281	4.084.100
5	322.101	85.766.120
6	3.543.121	1.801.088.540
7	38.974.341	37.822.859.360

Formeln:
minimal=2*(O+1)^h-1
maximal=(2*O+1)^(h+1)-1

Ordnung: 16		
Höhe	minimal	maximal
0	1	32
1	33	1.088
2	577	35.936
3	9.825	1.185.920
4	167.041	39.135.392
5	2.839.713	1.291.467.968

Ordnung: 32		
Höhe	minimal	maximal
0	1	64
1	65	4.224
2	2.177	274.624
3	71.873	17.850.624
4	2.371.841	1.160.290.624

Ordnung: 64		
Höhe	minimal	maximal
0	1	128
1	129	16.640
2	8.449	2.146.688
3	549.249	276.922.880
4	35.701.249	35.723.051.648

Ordnung: 128		
Höhe	minimal	maximal
0	1	256
1	257	66.048
2	33.281	16.974.592
3	4.293.377	4.362.470.400

Tabelle 1

5.5. Die C-Quell-Dateien

Datei	Beschreibung
Makefile	Makefile für das Hauptprogramm
index.c	Beinhaltet Hauptprogramm
index.h	Globale Vereinbarungen des Hauptprogramms
btree/btree.c	Beinhaltet die Funktionen zur Arbeit mit B-Bäumen
btree/btree.h	Prototypen und Strukturen zur Arbeit mit B-Bäumen
btree/node.h	Definiert Prototypen als Schnittstelle
btree/nodemem.c	Implementiert Schnittstelle als dynamische Speicherallokation
btree/nodemem.h	Underlying-spezifische Strukturen
Btree/nodefile.c	Implementiert Schnittstelle als Stream
Btree/nodefile.h	Underlying-spezifische Strukturen
Btree/testprog/makefile	Makefile für die 2/4 Programme des Verzeichnisses
Btree/testprog/rndbtree.c	Programm zum automatischen und zufälligen Hinzufügen, Löschen von Schlüsseln im B-Baum in beliebigem Underlying
Btree/testprog/tbtree.c	Programm zum manuellen Hinzufügen, Löschen von Schlüsseln im B-Baum in beliebigem Underlying

5.6. Optimierungsmöglichkeiten

Für eine gute Implementierung ist immer verwendete Algorithmik ausschlaggebend. Durch die allgemein anerkannte Verwendung des B-Baumes bzw. seiner Abwandlungen in Datenbanken bleibt für die Optimierung hauptsächlich nur noch die Minimierung der versteckten Konstante. Verschiedene Vorschläge und Ansätze hierzu werden weiter unten gemacht.
Der Rahmen um das Programm bietet verschiedene Optimierungsansätze. Zu jedem Schlüssel wurde als Wert der Offset innerhalb der Ausgangsdatei übergeben. Das geschieht über die Skripte binär, um nur 4 Byte je Zahl zu "verschwenden". Um die doch relativ groß werdenden B-Baum-Indexdateien zu schmälern, könnte dieser Ansatz auch auf die Schlüssel übertragen werden. So könnte man sich auf ein Alphabet einigen (Namen beinhalten fast nur Buchstaben), das man dann gepackt kodiert.
Im vorliegenden Beispiel wurde ebenfalls nicht die maximale Schlüssellänge zum Aufbau des Index benutzt, da auch die Möglichkeit besteht, eine Auswahlmenge auf Korrektheit nachzufiltern. Dies wären alles Verfahren, die auf dem hier vorgestellten Programm aufsetzen können.
Das Zeitverhalten der Skripte ist im Vergleich zum Index katastrophal, was hauptsächlich in der Implementierung, aber auch in deren Wesen zu begründen ist. Das wird in der Ausgabe einer Abfrage besonders deutlich, da hier für jede Zeile separat ein aufwendiges awk-Skript gestartet wird. Als erste Verbesserung zukünftiger Versionen dieses "Framework" stünde die Erstellung von (in C geschriebenen) Programmen, die die Funktionalität der Shellskripte ersetzen, damit dem Betriebssystem nur noch die Verwaltung der durch Pipes verbundenen Programme überlassen wird.

6. Zukünftige, mögliche Erweiterungen (To-Do-Liste)

- Automatische Aufnahme des Zeitverhaltens
- Codeoptimierungen, z.B. Rekursionen mittels eigener Stacks auflösen
- Zugriffszeitoptimierte Serialisierung der Knoten (Cluster)
- Transaktionen zur sicheren Ausführung/Wiederaufnahme von Operationen
- Integration von optimierter, mehrdimensionaler Erstellung bzw. Abfragen mittels UB-Baum: mittels eines U(niversal)B-Baumes wird ein Punkt im mehrdimensionalen Raum auf eine Dimension transformiert. Dies geschieht mittels einer Funktion, die räumlich nahe beieinanderliegende Punkte auf nahe beieinanderliegende Werte abbildet. Eine einfache Funktion stellt hier das Bit-Interleaving dar, die Punkte fraktal z-förmig miteinander verbindet. Diese Technik ist im Grunde nur einem Index, wie dem hier Vorgestellten, vorgeschaltet.

7. Quellenangabe

- c't 1/2001 174pp „Datenbanktechnik – Datenbankindexe in mehreren Dimensionen"
- Peter van der Linden – Expert C Programming – Deep C Secrets
- www-info.mpi.htwm.de/~Geiler/Lehre/masch_c/
- Manual-Pages

Stefan Pohl
spohl1@htwm.de

8. Bemerkungen

Hiermit versichere ich den Beleg selbständig, nur unter Zuhilfenahme oben genannter Quellen, erstellt zu haben.

Unterschrift

Mittweida, den 18.03.2001

Stefan Pohl

9. Quellcode