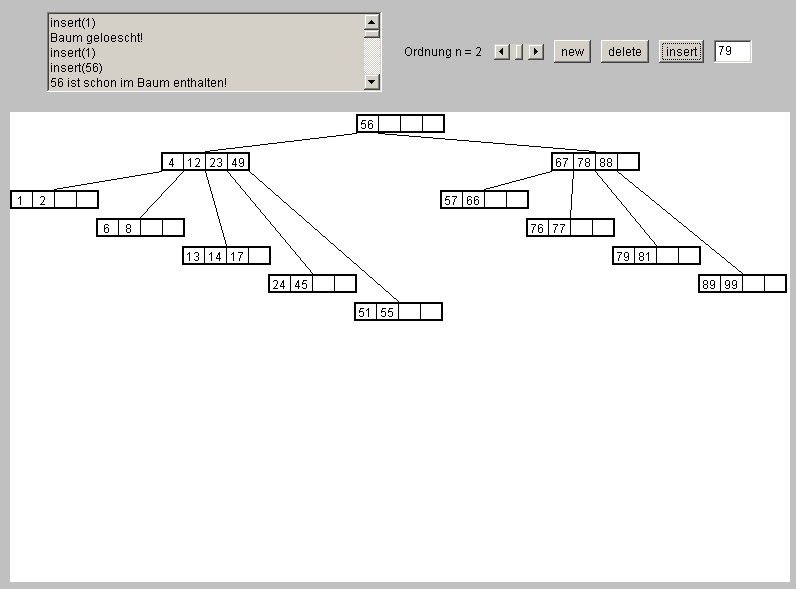
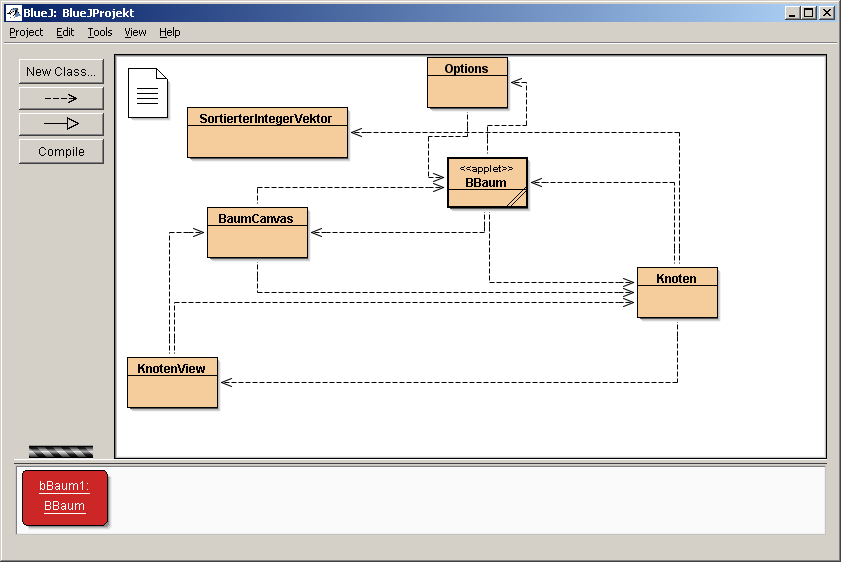
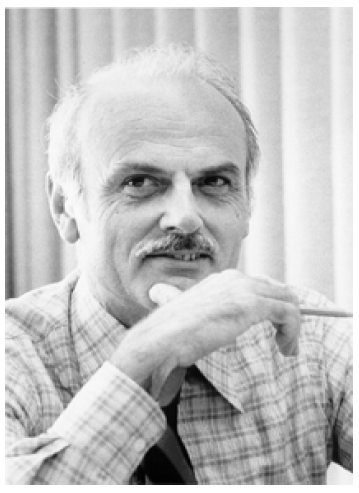
Indexstrukturen  
B-Bäume

 <https://www.cs.usfca.edu/~galles/visualization/BTree.html> 

Codd: Bayer:

I. Einleitung 3

I.1. Motivation für Indexstrukturen 3

I.2. Welche Indexstrukturen gibt es? 3

I.3. Anwendungsbereiche 4

II. B-Bäume 4

II.1. Aufbau 5

II.2. Funktion 6

II.3. Eigenschaften 8

II.4. Spezielle B-Bäume 8

III. Aufgabe 1 9

Grosse Datenmegen erzeugen mit T-SQL : 10

Indexe optimieren : 10

# Einleitung

## Motivation für Indexstrukturen

Durch wachsende Rechen- und Speicherkapazitäten können immer mehr Daten verarbeitet werden. Bei der Suche in grossen Datenmengen ist ein „Durchforsten“ der gesamten Daten nicht sinnvoll. So würde ein lineares Durchsuchen (Full Table Scan) einer 10 GB-Tabelle:

bei 10 MB/s 🡺 17 Minuten

Typischerweise betreffen Suchanfragen aber nur einen verhältnismässig kleinen Anteil der vorhandenen Daten. Je nach Einschränkung der Suche umfasst die Antwort nur wenige Prozent oder Promille (oder noch weniger) aller Daten. Aus diesem Grunde ist die Verwendung von Indexstrukturen ist sinnvoll. Diese unterteilen die Daten bereits beim Einsortieren und können so die Anzahl der bei einer Suche zu lesenden Datenseiten zu minimieren und damit die Geschwindigkeit zu erhöhen.

* <http://de.wikipedia.org/wiki/B-Baum>
* <http://people.ksp.sk/~kuko/bak/index.html>
* <http://www.youtube.com/watch?v=coRJrcIYbF4>
* <http://www.sqlservercentral.com/articles/Indexing/68439/>
* <http://www.sqlservercentral.com/articles/Indexing/68563/>
* <http://www.sqlservercentral.com/articles/Indexing/68636/>

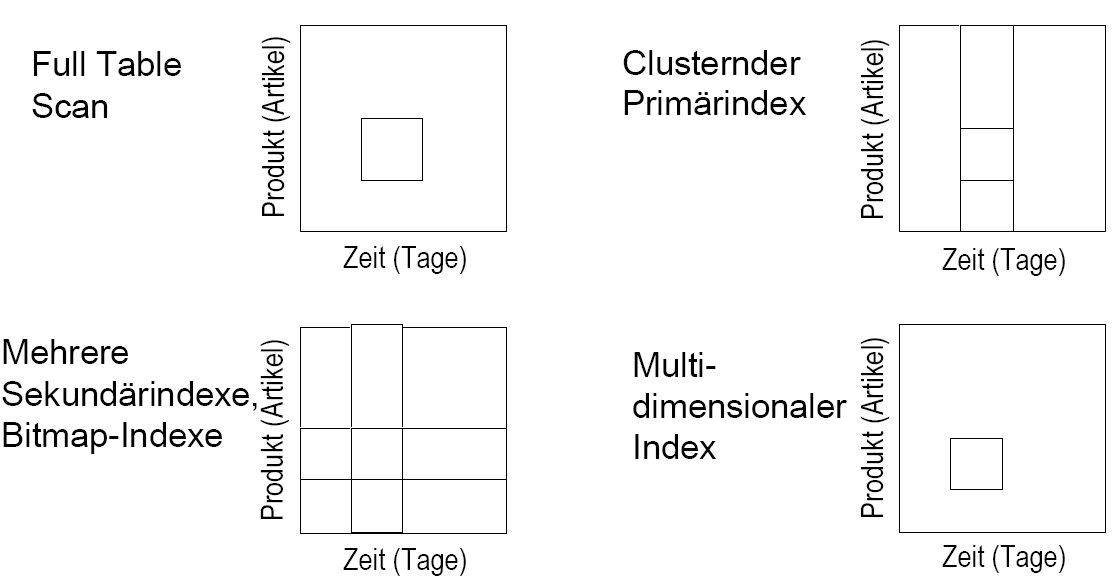
## Welche Indexstrukturen gibt es?

Das einer geordneten Indexstruktur zugrunde liegende Konzept ist vergleichbar mit dem Index eines Buches.

Indexstrukturen lassen sich nach der Anzahl der Attribute (Dimensionen), die für die Berechnung des Index-Schlüssels verwendet werden klassifizieren. So gibt es eindimensionale Indexe (z.B. B-Baum und Linear Hashing), die nach einem Primärschlüssel indiziert werden und damit eine Clusterung der Datei bewirken und multi-dimensionale Indexe (R-Baum, UB-Baum), die bei der Suche mit Einschränkung mehrerer Attribute hilfreich sind (siehe Abbildung 1). Bitmap-Indexe gibt es sowohl mit eindimensionaler wie in multi-dimensionaler Ausprägung. Diese beschreiben Attribute genauer und sind einfacher auszuwerten.

Der Index speichert in der Regel jeden Wert des Indexfeldes zusammen mit einer Liste von Verweisen auf alle Datenblöcke, die Datensätze mit diesem Feldwert enthalten. Die Indexdatei ist viel kleiner als die Datendatei.

Abbildung 1: Klassifikation von Indexstrukturen



## Anwendungsbereiche

In grossen Datenbanken kommen oft zwei Arten des Suchens vor:

* Sequentielles (lineares) Suchen, Ausgabe des 1.,2.,3.,... Datensatzes
* Eingeschränktes (Random) Suchen, Ausgabe eines bestimmten oder eingeschränkten

Für das Sequentielle Suchen benötigt man die Basis-Operation “Next”, die auf den nächsten Datensatz verweist. Für diese spezielle Anforderung (z.B. Ausgabe einer kompletten Artikelliste) eignen sich besonders B+-Bäume.

Für das eingeschränkte Suchen benötigt man die Basis-Operationen “Insert”, “Delete”, “Find”. Diese nennt man auch Random-Operations. Bei Datenbanken, die vor allem diese Funktionen erfordern, bietet sich eine Indizierung mit den einfachen Hashing- oder B-Baumstrukturen an.

Bitmap-Indexe bieten sich für die Indizierung innerhalb von Data-Warehouses an. Dort werden von den Benutzern komplexe Anfragen (wie Filterung und Gruppierung) auf grossen Datenmengen ausgeführt. Um diese Anfragen zu beschleunigen, werden die Daten mit speziellen Indexen beschrieben (z.B. mit Bitmap-Indexen) oder es werden bestimmte Dimensionen schon beim Update vorausgewertet. Für Attribute, die einen sehr begrenzten Wertebereich haben (z.B. Geschlecht = nur männlich oder weiblich), bieten sich die Bitmap-Indexe besonders an.

In den folgenden Kapiteln werden B-Bäume, Linear Hashing und Bitmap-Indexe erläutert. Dabei werden in den Kapiteln der Aufbau, die Funktion und die Eigenschaften dieser Indexstrukturen näher erläutert.

# B-Bäume

Ein B-Baum ist eine Indexstruktur, die sich von Binären Bäumen herleiten lässt und für die schnelle Datenmanipulation geeignet ist.

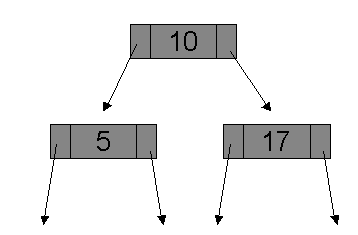
Die Bedeutung des Begriffs „B-Baum“ wurde von den Autoren nie erklärt. Das „B“ könnte „Balanced“, „Broad“, „Bushy“ bedeuten. Andere sagen es könnte auch Boeing sein, weil die Entwicklung unter anderem in einem Forschungslabor dieses Flugzeugherstellers stattfand. Weil Bayer 1972 diesem Konstrukt den Namen gab, wäre es passend ihn „Bayer“-Tree zu nennen.

Im Folgenden wird zuerst der Standard-B-Baum erläutert. Im Abschnitt Spezielle B-Bäume werden dann einige Varianten näher vorgestellt.

## Aufbau

Wie oben schon erwähnt leiten sich B-Bäume von den Binären Bäumen ab. Bei diesen hat jeder Knoten genau einen Wert und zwei Zeiger auf die beiden Söhne (deswegen auch binär=zwei). Abbildung 2 zeigt ein Beispiel für einen Binären Baum. In jedem Knoten zeigt der linke Zeiger immer auf einen Knoten mit einem kleineren Wert, der rechte Zeiger auf einen Knoten mit einem grösseren Wert. So erfordert die Suche nach einem bestimmten Index an jedem Knoten einen Vergleich.

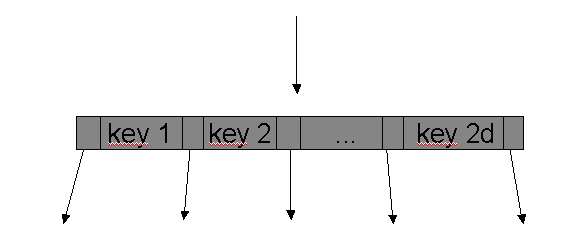
Abbildung 2: Beispiel für einen Binären Baum



B-Bäume sind insofern eine Generalisierung von Binären Bäumen, als dass sie mehr als einen Eintrag pro Knoten zulassen und dass sie balancierte Algorithmen zum Einfügen und Löschen verwenden.

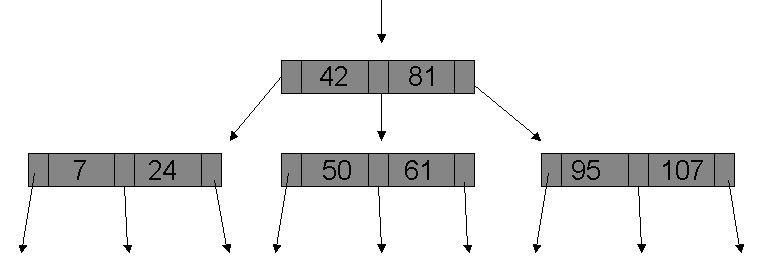
Ein Knoten in einem B-Baum der Ordnung d hat Platz für 2d Einträge und besitzt höchstens 2d+1 Zeiger (siehe Abbildung 3). Die Ordnung d bedeutet, dass in einem Knoten mind. d Elemente enthalten sein müssen (ausser der Wurzel), damit wird eine Auslastung der Indexstruktur von mind. 50% erreicht.

Abbildung 3: Aufbau eines Knotens im B-Baum mit Ordnung d



Für einen B-Baum der Ordnung 1 ergeben sich hiermit zwei Werte pro Knoten, die von drei Zeigern auf die Söhne eingeschlossen werden (siehe Abbildung 4).

Abbildung 4: B-Tree der Ordnung 1, mit 2 Werten pro Knoten und 3 Zeigern

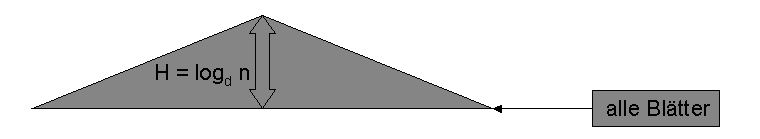


Eine weitere Eigenschaft von B-Bäumen ist, dass sie immer balanciert sind, dadurch haben alle Pfade von der Wurzel bis zu den Blättern die gleiche Länge. Dadurch hat jeder B-Baum äusserlich die Form eines Dreiecks, bei dem die untere Seite vollständig aus Blättern besteht und der Rest oberhalb aus inneren Knoten.

Die Höhe h eines Baumes ist das Maximum der Weglängen aller Knoten.

Sei d die Ordnung und n die Anzahl Blätter des B-Baums. Dann gilt für die Höhe h des Baums:

**log2d+1(n+1) <= H <= 1+logd+1((n+1)/2)**



Die ungefähre maximale Anzahl Knoten N in einem Baum der Höhe H mit Grad d ist

**N(d, H) = 1 +d +d2 +d3 + ... + dH = (dH+1-1)/(d-1)**(bitte Formel **nicht** in Prüfungen verwenden)

Zusammenfassend ist ein B-Baum mit der Ordnung d durch die folgenden Eigenschaften charakterisiert:

* Der Baum ist vollständig balanciert, d.h. alle Wege von der Wurzel zu den Blättern sind gleich lang.
* Jeder Knoten hat mindestens d und maximal 2d Einträge. Die Wurzel kann zwischen 1 und 2d Einträge haben. Die Einträge werden in allen Knoten sortiert gehalten
* Alle innere Knoten mit n Einträgen haben n+1 Kinder
* Ein Knoten muss folgendermassen organisiert sein:
* Seien S1 bis Sn die Schlüssel eines Knotens mit n+1 Kindern. V0 bis Vn seien die Verweise auf diese Kinder. Dann gilt:

o V0 weist auf Teilbaum mit Schlüsseln kleiner als S1.

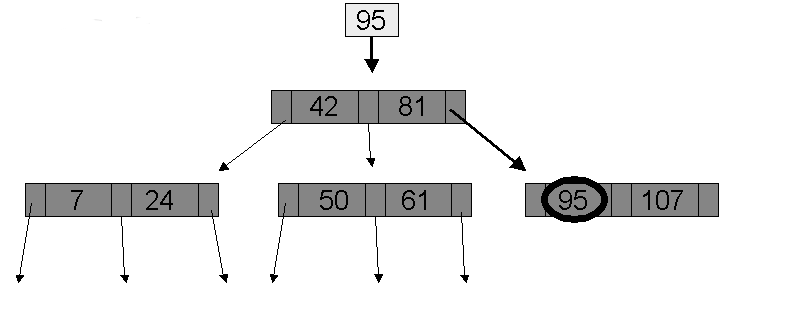
o Vi (i = 1...n-1) weist auf Teilbaum, dessen Schlüsseln zwischen Si und Si+1 liegen.

o Vn weist auf den Teilbaum mit Schlüsseln grösser als Sn.

## Funktion

Das Suchen in einem B-Baum gestaltet sich relativ einfach, da die Schlüssel sortiert sind. Somit müssen von der Wurzel her nur Vergleiche durchgeführt werden und man gelangt so zu dem gesuchten Index. Abbildung 5 zeigt wie der Wert 95 gesucht wird. Zuerst wird die 95 mit der 42 verglichen. Da 95 grösser ist muss weiter verglichen werden. Die 95 ist grösser als die 81 somit führt der Zeiger rechts von der 81 zum nächsten zu durchsuchenden Knoten. Dort würde die Prozedur von vorn beginnen, wenn die 95 nicht schon dort vorhanden wäre.

Abbildung 5: Suche in einem B-Baum der Ordnung 1



eigene Darstellung

B-Bäume sind immer balanciert. Das bedeutet für die Funktionen Einfügen und Löschen, dass sie immer einen balancierten Baum hinterlassen müssen.

Das Balancieren beschränkt sich bei einer Funktion auf einen Pfad von einem Blatt zur Wurzel, dies verringert die Komplexität und verbessert die Kosten dieser Funktionen.

Beispiel für die Einfüge-Funktion:

1. Suchen ob nicht einzufügender Wert schon vorhanden, dafür Durchlaufen des B-Baums und Vergleichen
2. falls nicht vorhanden, an dem Blatt einfügen, wo die Suche gescheitert ist
3. falls Knoten voll 🡪 Aufteilen des Inhalts in zwei neue Blätter (Splitting), untere (d-1) Schlüssel in linken Sohn, d als Separator in Vater und obere d bis 2d Schlüssel in rechten Sohn
4. falls Vaterknoten schon voll genauso wie bei 4 vorgehen

Nur wenn sich dieser Aufteil-Prozess bis zur Wurzel fortsetzt erhöht sich die Tiefe des B-Baums um eins. Dies ist sehr effektiv, da die Tiefe gering gehalten wird und sich so die Suchzeiten verringern.

Abbildung 7 und 8 verdeutlichen diesen Vorgang an einem Beispiel.

Abbildung 6: Einfügen in einem B-Baum

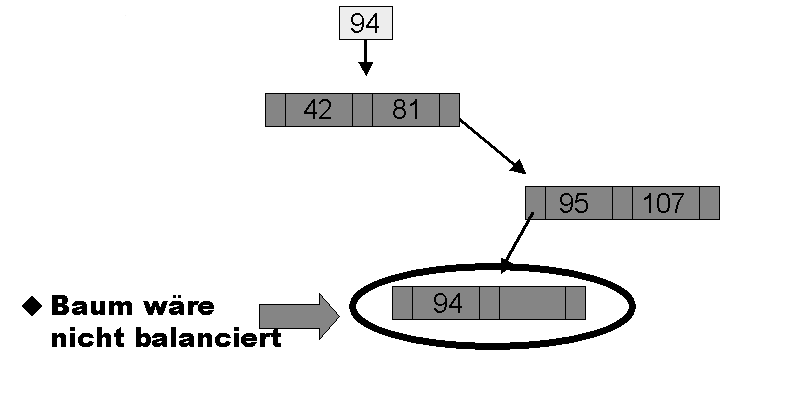
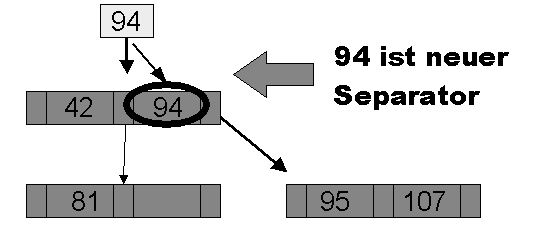


Abbildung 7: Balancieren eines B-Baums beim Einfügen durch Splitting



eigene Darstellung

Die Lösch-Funktion gestaltet sich ähnlich:

1. Suchen nach zu löschendem Wert, dafür Durchlaufen des B-Baums und Vergleichen
2. falls vorhanden, Wert aus dem Knoten löschen
3. falls Knoten unterbesetzt (weniger als d Elemente) 🡪 Zusammenführen des Inhalts zweier Knoten
4. falls Vaterknoten ebenfalls unterbesetzt 🡪 genauso wie bei 4 vorgehen

Nur wenn sich dieser Zusammenführungs-Prozess bis zur Wurzel fortsetzt verringert sich die Tiefe des B-Baums um eins.

Abbildung 8: Zum Löschen, muss das Element zuerst gesucht werden

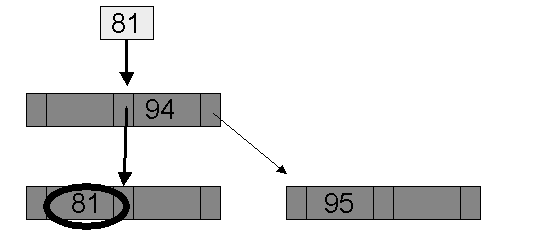
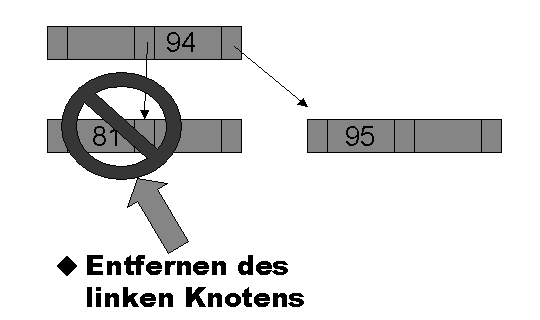


Abbildung 9: Löschen eines Knotens, wegen Unterbelegung



## Eigenschaften

Zeit ist bekanntlich Geld und so werden die Indexstrukturen auch auf ihre Effektivität (Kosten) hin untersucht. Ein Mass für die Kosten einer Operation ist die benötigte Zeit, diese lässt sich auch durch die Anzahl der durchlaufenen Knoten darstellen. Ein weiteres Kriterium (weniger wichtig) ist die Speicherausnutzung, also das Verhältnis von Index zu Datengrösse.

B-Bäume mit mehreren Schlüsseln und Verzweigungen verringern die Suchkosten gegenüber Binären Bäumen, da sie balanciert sind und mehr als zwei Einträge pro Knoten aufnehmen können. Diese schnellere Suche wird aber mit höheren Kosten für Einfüge und Lösch-Operationen erkauft. Diese höheren Kosten werden aber durch die schnellere Suche wieder mehr als kompensiert. Für eine Such-Operation in einem unbalancierten Baum, muss man im worst case für n Schlüssel auch n Knoten durchlaufen. In einem balancierten B-Baum dagegen sind dies für n Schlüssel ebenfalls im worst case nur 1 + logd n . Diese logarithmischen wachsenden Kosten ergeben sich daraus, dass für eine Suchoperation max. ein Pfad von der Wurzel bis zu einem Blatt durchlaufen werden muss. Da ein B-Baum maximal die Höhe logd n hat, kann dieser Pfad nicht länger sein als die Höhe des B-Baums.

Die zusätzlichen Operationen für die Einfüge- und Löschfunktion sind lokal begrenzt. Sie beziehen sich nur auf einen Pfad vom Blatt zur Wurzel, in dem balanciert wird. Für die Anwendung in Data-Warehouses ist dieser Reorganisationsaufwand von untergeordneter Bedeutung.

## Spezielle B-Bäume

Der B\*-Baum ist eine strengere Definition des Standard-B-Baums. Hier ist die Vorgabe, dass die Knoten immer zu 2/3 gefüllt sein müssen. Dies bringt eine Speicherauslastung von 66% und eine schnellere Suche (da der Baum weniger tief ist). Der Nachteil ist, dass die Entfernen- und Einfüge-Operationen teurer sind, da Knoten sehr oft schon voll oder unterbelegt sind.

Ein Problem beim Standard B-Baum ist, dass sequentielle Operationen (z.B next) sehr kostenintensiv sind (viele Knoten müssen durchlaufen werden, ein Stack angelegt werden usw.. Die Lösung dafür sind B+-Bäume. Dies ist eine Kombination aus B-Baum und verketteter Liste. Alle Schlüssel sind in den Blättern des Baums gespeichert. Alle inneren Knoten dienen nur als Wegweiser, um von der Wurzel zu den Blättern zu kommen. Die Blätter sind alle mit einer verketteten List miteinander verbunden. Schlüssel sind in sich verkettete Liste mit Index-Struktur oben drüber.

Der Vorteil liegt klar auf der Hand: sowohl eingeschränkter als auch sequentieller Zugriff ist schnell möglich. B+-Bäume eignen sich so besonders für Bereichsanfragen oder die Ausgabe von kompletten Listen.

# Aufgabe 1

Üben sie B-Bäume zeichnen mit dem abgegebenem Applet!

Erzeugen sie mit dem Applet einen B-Tree der Ordnung 2. Geben sie solange Daten ein bis die Tiefe des Baumes auf 3 angestiegen ist.

<https://www.cs.usfca.edu/~galles/visualization/BTree.html>

<https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html>

# 

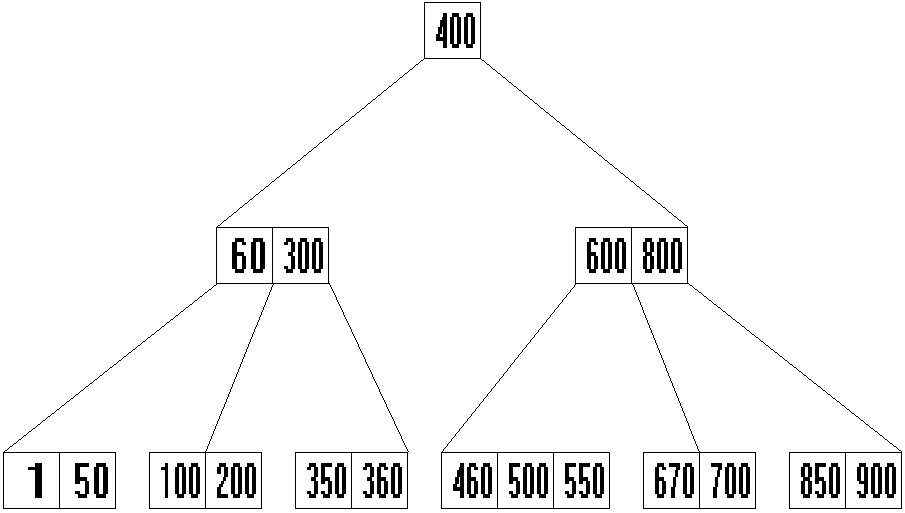
**Weitere Aufgaben mit Lösungen zum Thema Indexe:**

**** ****

**Formeln:**

Maximale Anzahl der Schlüssel *N* in einem B-Baum der Höhe *h* und der Ordnung *d*:

N = 2d+(2d+1)12d+(2d+1)22d+(2d+1)32d+ … +(2d+1)h2d

**Beispiel-1:**

Ordnung = 2  
Höhe = 2

N = 2\*2+(4+1)2\*2+52\*4 = 4 + 20 + 100 = **124**

**Beispiel-2:**Ordnung = 400   
Höhe = 2

N = 2\*400+(2\*400+1)2\*400+(2\*400+1)22\*400 = **513’922’400**

# Grosse Datenmegen erzeugen mit T-SQL :

SELECT TOP 10000000 --<<<LOOK! CHANGE THIS NUMBER TO CHANGE THE NUMBER OF ROWS!

RowNum = IDENTITY(INT,1,1),

SomeID = ABS(CHECKSUM(NEWID()))%25+1, --<<<LOOK! CHANGE THIS NUMBER TO 1/400th THE ROW COUNT

SomeCode = CHAR(ABS(CHECKSUM(NEWID()))%26+65) + CHAR(ABS(CHECKSUM(NEWID()))%26+65)

INTO dbo.TestData

FROM Master.dbo.SysColumns t1, Master.dbo.SysColumns t2

PK-Index erstellen:

ALTER TABLE dbo.TestData

ADD PRIMARY KEY CLUSTERED (RowNum)

# Indexe optimieren :



SET NOCOUNT ON;

DECLARE @objectid int;

DECLARE @indexid int;

DECLARE @partitioncount bigint;

DECLARE @schemaname nvarchar(130);

DECLARE @objectname nvarchar(130);

DECLARE @indexname nvarchar(130);

DECLARE @partitionnum bigint;

DECLARE @partitions bigint;

DECLARE @frag float;

DECLARE @command nvarchar(4000);

-- Conditionally select tables and indexes from the sys.dm\_db\_index\_physical\_stats function

-- and convert object and index IDs to names.

SELECT

object\_id AS objectid,

index\_id AS indexid,

partition\_number AS partitionnum,

avg\_fragmentation\_in\_percent AS frag

INTO #work\_to\_do

FROM sys.dm\_db\_index\_physical\_stats (DB\_ID(), NULL, NULL , NULL, 'LIMITED')

WHERE avg\_fragmentation\_in\_percent > 10.0 AND index\_id > 0;

-- Declare the cursor for the list of partitions to be processed.

DECLARE partitions CURSOR FOR SELECT \* FROM #work\_to\_do;

-- Open the cursor.

OPEN partitions;

-- Loop through the partitions.

WHILE (1=1)

BEGIN;

FETCH NEXT

FROM partitions

INTO @objectid, @indexid, @partitionnum, @frag;

IF @@FETCH\_STATUS < 0 BREAK;

SELECT @objectname = QUOTENAME(o.name), @schemaname = QUOTENAME(s.name)

FROM sys.objects AS o

JOIN sys.schemas as s ON s.schema\_id = o.schema\_id

WHERE o.object\_id = @objectid;

SELECT @indexname = QUOTENAME(name)

FROM sys.indexes

WHERE object\_id = @objectid AND index\_id = @indexid;

SELECT @partitioncount = count (\*)

FROM sys.partitions

WHERE object\_id = @objectid AND index\_id = @indexid;

-- 30 is an arbitrary decision point at which to switch between reorganizing and rebuilding.

IF @frag < 30.0

SET @command = N'ALTER INDEX ' + @indexname + N' ON ' + @schemaname + N'.' + @objectname + N' REORGANIZE';

IF @frag >= 30.0

SET @command = N'ALTER INDEX ' + @indexname + N' ON ' + @schemaname + N'.' + @objectname + N' REBUILD';

IF @partitioncount > 1

SET @command = @command + N' PARTITION=' + CAST(@partitionnum AS nvarchar(10));

EXEC (@command);

PRINT N'Executed: ' + @command;

END;

-- Close and deallocate the cursor.

CLOSE partitions;

DEALLOCATE partitions;

-- Drop the temporary table.

DROP TABLE #work\_to\_do;

**SQL-Skripts im Zusammenhang mit Indexen:**



# Glossar

Height, depth = Tiefe des Baums, wird synonym gebraucht (vgl. Comer 1979: 127)

Key = Schlüssel

Leaf = Blatt

Load = Auslastung der Indexstruktur (nach Litwin 1980: 212)

Node = Knoten

nonleaf node = innerer Knoten = Knoten der kein Blatt ist

Path = Pfad, von der Wurzel zu einem Blatt durchlaufener Weg

Primary key (Litwin 1996) = RID (Row IDentifier) (O'Neil / Quass 1997: 39)

Splitting = Aufteilen eines Knoten in zwei Söhne und einen Separator-Schlüssel im Vater (beim Einfügen)