B-Bäume, Hashtabellen, Cloning/Shadowing, Copy-on-Write

Thomas Maier

Proseminar: Ein- / Ausgabe – Stand der Wissenschaft

Gliederung

Hashtabelle

Eine Hashtabelle verwaltet Daten in einer Tabelle. Sie wird verwendet, um große Datenmengen effizient zu verwalten. Die Hashtabelle besitzt eine mathematische Funktion, welche Daten auf einen Integer Wert abbildet. Dieser Wert entspricht dann dem Index eines Arrays, in der diese Datei gespeichert wird. Im Idealfall bildet die Funktion jede Datei auf genau einen Integer Wert ab. In diesem Fall kann eine Datei direkt in eingefügt, gelöscht oder gefunden werden, da der Aufwand $O(1)$ ist.

Die Problematik einer Hashtabelle liegt jedoch darin, dass relativ schnell mehrere Kollisionen entstehen. Das heißt, dass mehrere Daten auf den gleichen Wert abgebildet werden und somit in den gleichen Index gespeichert werden. Daraus folgt, dass die Tabelle entartet und der Aufwand gegen $O(n)$ konvergiert.

Beispiel einer Hashtabelle:

$$Hashfunktion\ f(\mathrm{x}) = \begin{cases} 0, & falls\ x = a \\ 1, & falls\ x = b \\ 2, & falls\ x = c \end{cases}$$

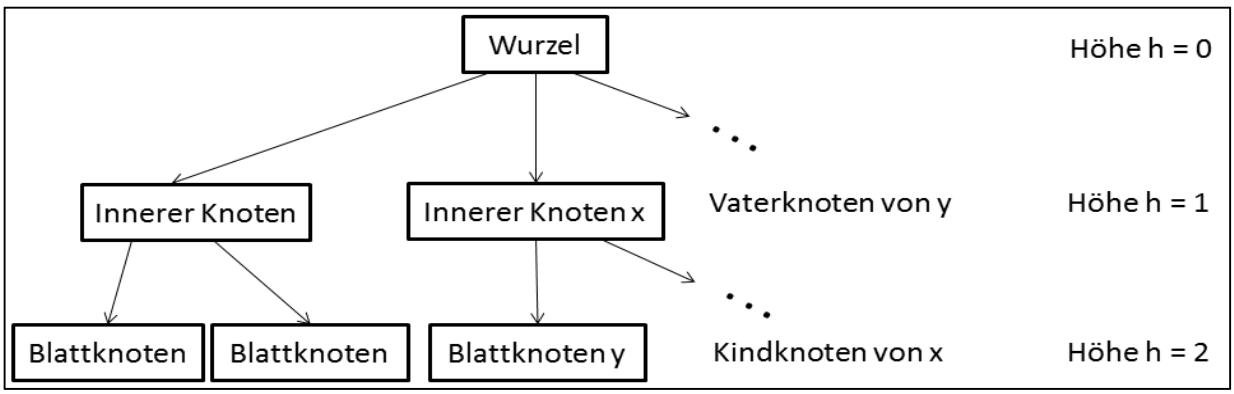
Array

Index $f(x)$	**Datei**
0	a
1	b
2	c

B-Bäume

Der B-Baum (engl. B-tree) ist eine Daten- / Indexstruktur in Form eines vollständig ausbalancierten Baumes, welcher häufig in Datenbanken eingesetzt wird. Der B-Baum wurde 1972 von Rudolf Bayer und Edward M. McCreight entwickelt.

Begriffserklärung: Der oberste Knoten eines Baumes wird *Wurzel* genannt. Die untersten Knoten werden als *Blattknoten* bezeichnet und alle Knoten zwischen Wurzel und Blattknoten werden *innere Knoten* genannt.



Motivation: Bevor wir zur Definition des B-Baumes kommen, betrachten wir einen binären Baum(siehe Abb. 1), welcher eine große Menge an Daten verwaltet. Der Aufwand eine Datei zu finden liegt bei $O(h)$. Da es sich hier jedoch um einen binären Baum handelt, wird h bei einer großen Menge von Daten sehr groß. Somit wird der Aufwand bei einer großen Anzahl an Elementen sehr hoch und der Zugriff auf die Datenbank relativ langsam.

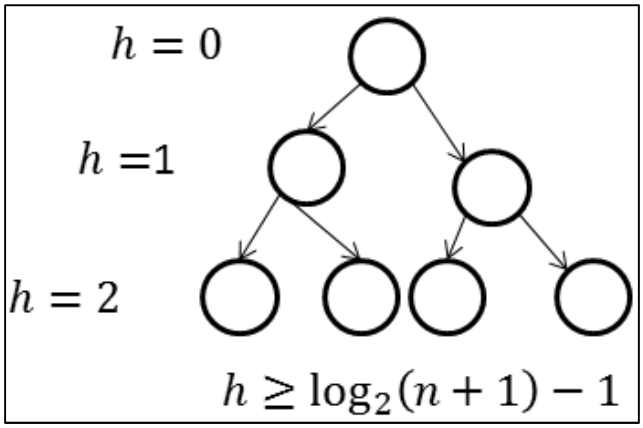


Abbildung 1

Definition: Der B-Baum hat folgende Eigenschaften: Jeder Knoten hat eine variable Anzahl an Schlüsseln, welche jeweils auf eine Datei verweisen und der Größe nach geordnet sind. Diese Anzahl wird mit der Variable s bezeichnet. Des Weiteren gibt es zu jedem Baum einen Parameter t, welcher auch als die *Ordnung des Baumes* bezeichnet wird. Durch diesen Parameter wird s beschränkt, denn für jeden Knoten, außer der Wurzel, gilt, dass die Anzahl der Schlüssel zwischen t und $2t$ liegen muss. Für die Wurzel gilt, dass diese mindestens einen und höchstens $2t$ Schlüssel enthalten darf.

Ordnung des Baumes:

Für jeden Knoten außer der Wurzel gilt:	$t \leq Anzahl\ der\ Schlüssel \leq 2t$
Für die Wurzel gilt:	$1 \leq Anzahl\ der\ Schlüssel \leq 2t$

Zusätzlich gilt für jeden inneren Knoten, dass er $s + 1$ verweise auf Kindsknoten hat.

Beispiel eines Knotens: $|| \ k_1 \ || \ k_2 \ || \ k_3 \ ||$ Dieser Knoten hat 3 Schlüssel(k_i) und 4 verweise auf Kindsknoten(||).

B-Bäume sind vollständig ausbalanciert. Das Bedeutet, dass sich jedes Blatt des Baumes auf derselben Höhe befindet.

Suchen von Daten: In einem B-Baum soll ein Schlüssel k gesucht werden. Die Suche beginnt in der Wurzel. Dort wird der kleinste Schlüssel k_i ermittelt, für den $k \leq k_i$ gilt. Nun unterscheidet man in 3 Fälle:

1. Fall ($k = k_i$): In diesem Fall wurde der Schlüssel gefunden und wir können die Suche erfolgreich beenden.

2. Fall ($k < k_i$): In diesem Fall gehen wir in den Unterbaum links von k_i und wiederholen den Schritt im aktuellen Knoten.

3. Fall (Es existiert kein solches k_i): In diesem Fall gehen wir in den Unterbaum ganz rechts und wiederholen den Schritt.

Wenn wir uns in einem Blatt befinden und der 2. oder 3. Fall eintritt, endet die Suche erfolglos.

Beispiel: Wir betrachten den folgenden Baum und wollen dort den Schlüssel $k = 31$ suchen.

Wir starten in der Wurzel (Abb. 2, roter Pfeil).

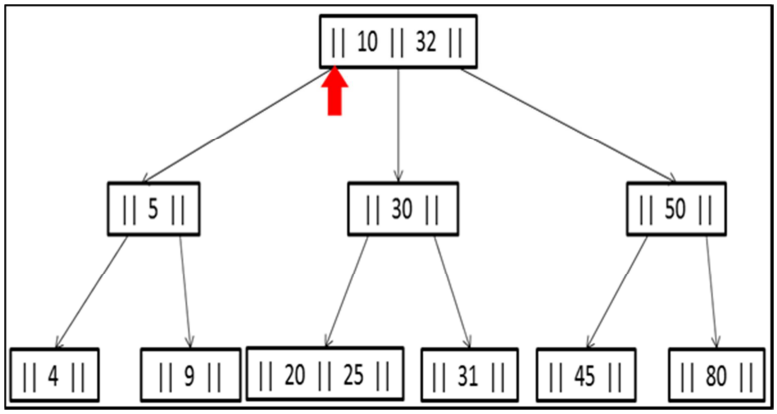


Abbildung 2

Nun suchen wir den kleinsten Schlüssel k_i , welcher größer als 31 ist (Abb. 3, blauer Pfeil). Es tritt der 2. Fall ein und wir wandern in den Unterbaum links von der 32 (Abb. 3, roter Pfeil). Wir befinden uns nun in einem anderen Knoten und wiederholen den Schritt. Es tritt der 3. Fall ein, da sich in diesem Knoten kein Schlüssel befindet, welcher größer als 31 ist. Wir wandern also in den Unterbaum ganz rechts und wiederholen das Ganze erneut. Es trifft nun der erste Fall ein, da sich dort der Schlüssel 31 befindet (Abb. 4, grüner Pfeil). Da wir nun den Schlüssel gefunden haben endet die Suche erfolgreich und wir sind fertig.

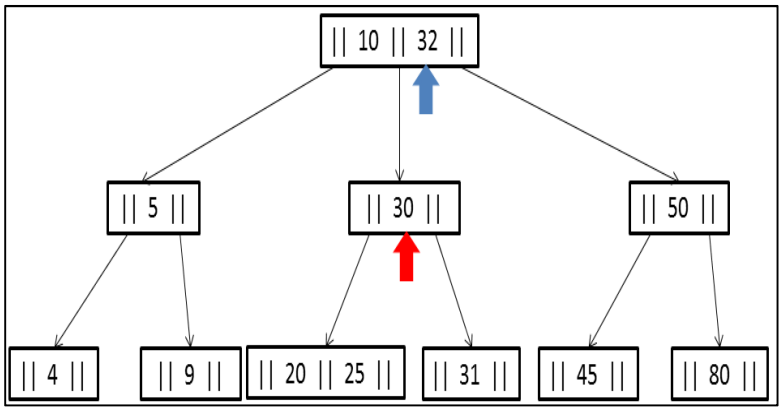


Abbildung 3

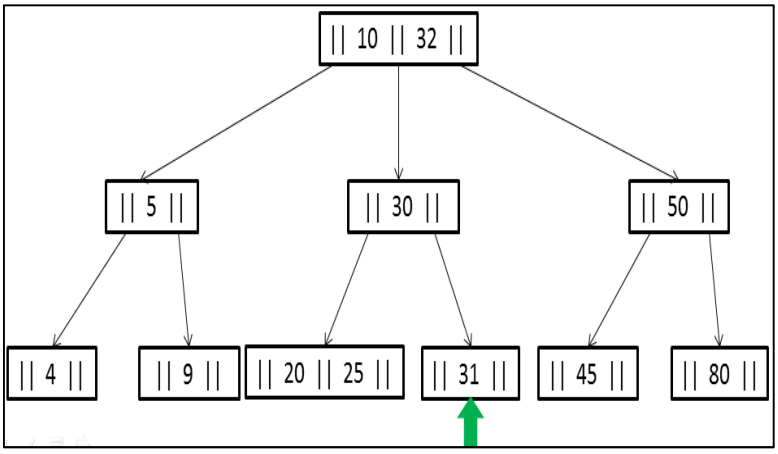


Abbildung 4

Einfügen von Daten: Nun wollen wir einen Schlüssel k einfügen. Der erste Schritt ist es, den Schlüssel k, mit dem oben genannten Algorithmus, zu suchen. Wenn wir den Schlüssel finden, wird das Einfügen abgebrochen, da sich der Schlüssel bereits im Baum befindet. Andernfalls endet die Suche erfolglos in einem Blatt. In diesem Blatt wird der Schlüssel k eingefügt. Dabei sind wieder 2 Fälle zu unterscheiden.

1. Fall (das Blatt ist nicht voll, d.h. enthält weniger als $2t$ Schlüssel): In diesem Fall fügen wir den Schlüssel k einfach ein und sind fertig.

2. Fall (das Blatt ist voll): In diesem Fall fügen wir den Knoten ebenfalls ein. Da wir nun aber gegen die Ordnung des Baumes verstoßen, splitten wir den Knoten wie folgt: Wir ziehen den mittleren Schlüssel raus. Dieser bildet mit der linken bzw. rechten Hälfte des Knotens je einen neuen Knoten (Abb. 5). Nun fügen wir den mittleren einelementigen Knoten rekursiv in den Vaterknoten ein.

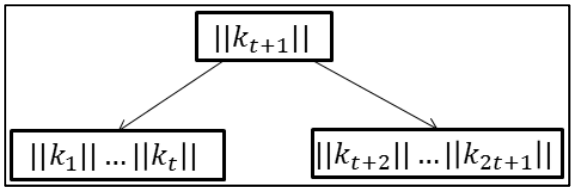


Abbildung 5

Beispiel: Wir wollen nun im folgendem Baum mit der Ordnung $t = 1$ den Schlüssel $k = 23$ einfügen. Also wird dieser Schlüssel zunächst in dem Baum gesucht. Da dieser Schlüssel noch nicht existiert endet die Suche erfolglos in einem Blatt(Abb. 6, roter Pfeil). Da dieser Knoten voll ist betrachten wir den oben beschriebenen 2. Fall. Wir fügen den gewünschten Schlüssel ein (Abb.7) und splitten anschließend den Knoten (Abb. 8). Nun wird der mittlere Schlüssel (in unserem Fall $k = 23$) in den Vaterknoten eingefügt (Abb. 9) und wir sind fertig.

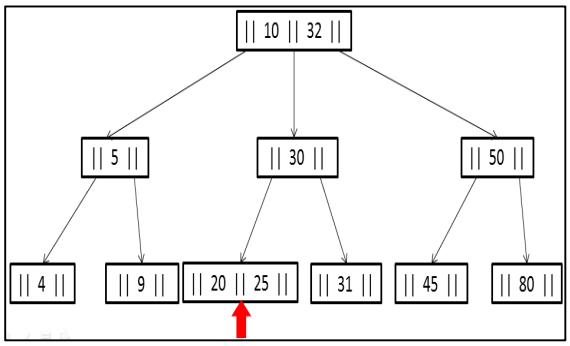


Abbildung 6

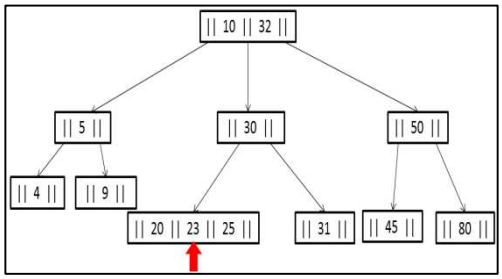


Abbildung 7

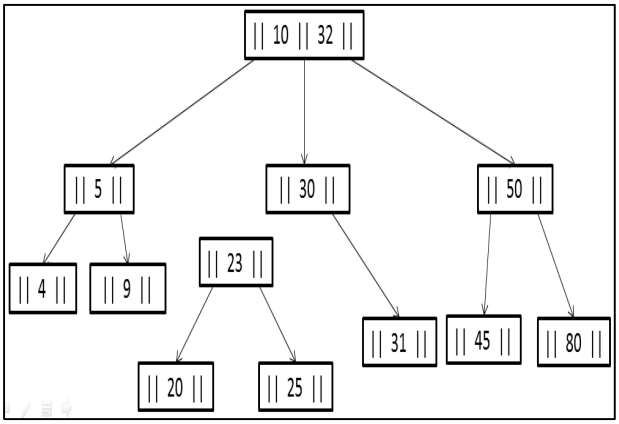


Abbildung 8

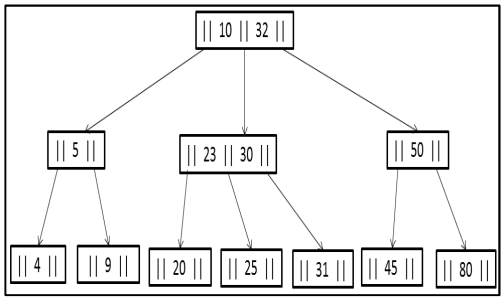


Abbildung 9

Löschen von Daten: Das Löschen von Daten aus einem B-Baum ist eine sehr komplexe Operation. Hierbei wird wieder in zwei Fälle unterschieden. Der erste Fall behandelt das Löschen aus einem Blatt und der zweite Fall beschreibt das Löschen aus einem inneren Knoten. In beiden Fällen muss selbstverständlich die Ordnung des Baumes wiederhergestellt werden. Bevor wir jedoch die Fälle betrachten werden wir zwei nützliche Verfahren anschauen.

Verschieben von Schlüsseln: Dazu betrachten wir den folgenden Teilbaum (Abb. 10). Nehmen wir an wir wollen im linken Knoten einen Schlüssel löschen. Dann stehen wir vor dem Problem, dass wir gegen die Ordnung verstoßen. Im Rechten Knoten haben wir jedoch einen Schlüssel übrig. Also verschieben wir den Schlüssel k'_2 in den linken Knoten und den Schlüssel $\boldsymbol{k_1}$ in seinen Vaterknoten. Der Pfad zum Knoten x wird nach rechts zu k'_2 verlegt und wir erhalten den folgenden Baum aus Abbildung 11. Der Pfad vom Knoten x kann verschoben werden, da jeder Schlüssel aus x größer als k'_2 und kleiner als $\boldsymbol{k_1}$ ist.

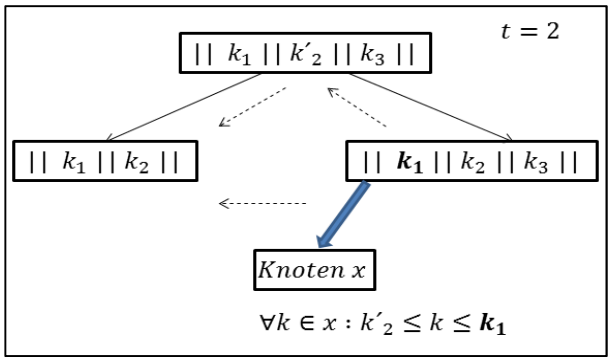


Abbildung 10

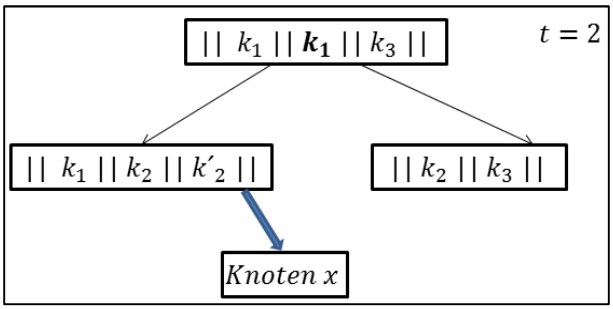


Abbildung 11

Verschmelzung zweier Knoten:
Wir betrachten nun den folgenden Teilbaum (Abb. 12). Wenn wir nun einen Schlüssel aus den unteren beiden Knoten löschen wollen, haben wir das Problem, dass wir in beiden Knoten zu wenig Schlüssel haben und auch keine Schlüssel verschieben können.
Deshalb wird der Schlüssel k'_1 nach unten gezogen und mit den beiden Kindsknoten verschmolzen(Abb. 13). Nun muss nur noch der gewünschte Schlüssel aus dem unteren Knoten entfernt werden, damit die Ordnung erhalten bleibt. Jetzt sind wir in der Lage jeden Knoten aus einem Blatt zu löschen.

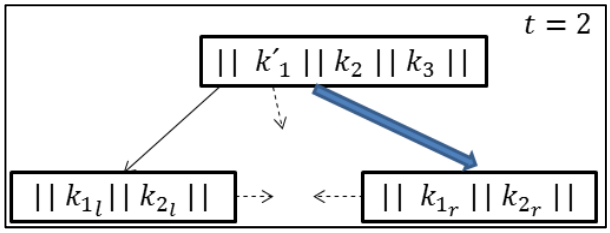


Abbildung 12

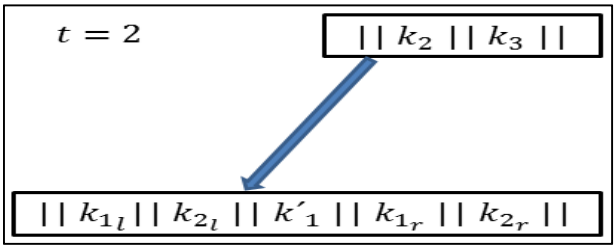


Abbildung 13

Betrachten wir nun das Verfahren für den 2. Fall (löschen aus einem inneren Knoten).

Dazu betrachten wir den folgenden Teil eines B-Baumes (Abb. 14).

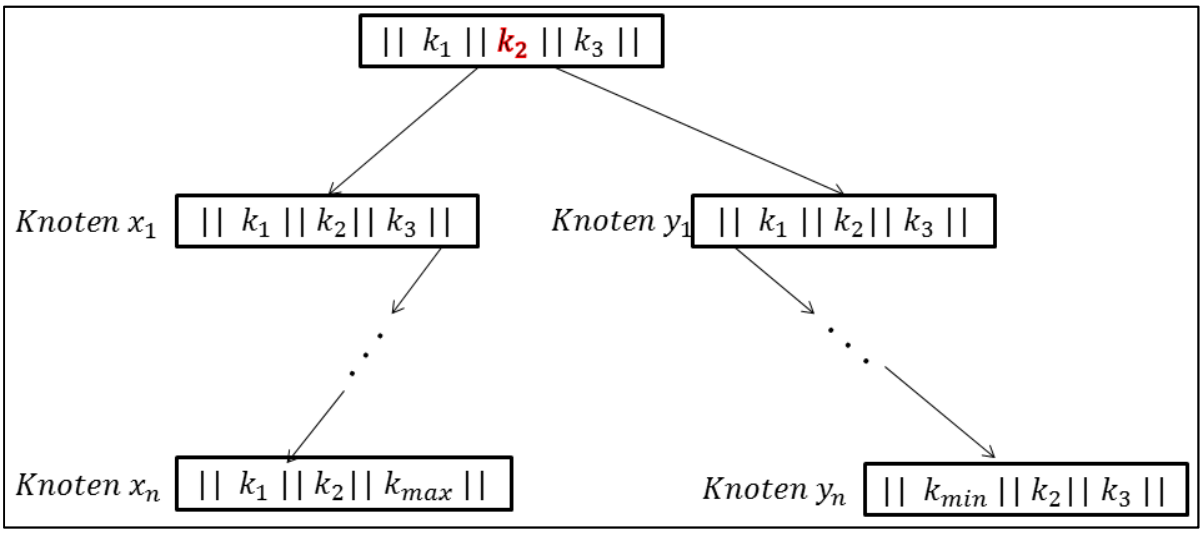


Abbildung 14

Wir wollen nun den rot markierten Schlüssel k_2 löschen. Zunächst definieren wir aber noch den symmetrischen Vorgänger und den symmetrischen Nachfolger von k_2.

Der *symmetrischer Vorgänger von* k_2 , wird mit $k_{\max}$ bezeichnet und ist der größte Schlüssel aus allen Blattknoten, welcher kleiner ist als k_2. Dieser wird wie folgt ermittelt: Wir wandern den Pfad links von k_2 in den Kindsknoten x_1 und wählen anschließend immer wieder den Pfad ganz rechts in den nächsten Knoten, bis wir uns in einem Blatt befinden. Dort befindet sich ganz rechts der symmetrische Vorgänger.

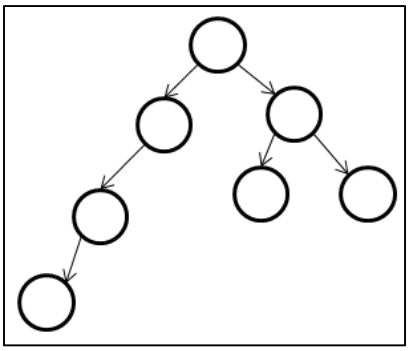
Es gilt: $\forall k_i \in x_i\ (1 \leq i \leq n) : k_i \leq k_{max} < k_2$

Der *symmetrische Nachfolger von* k_2 wird mit k_{min} bezeichnet und ist der kleinste Schlüssel aus allen Blattknoten, welcher größer ist als k_2. Dieser wird wie folgt ermittelt: Wir wandern den Pfad rechts von k_2 in den Knoten y_1 und folgen stets dem Pfad ganz links bis zum Blattknoten. Dort befindet sich ganz links der symmetrische Nachfolger von k_2.

Es gilt: $\forall k_i \in y_i\ (1 \leq i \leq n): k_i \geq k_{min} > k_2$

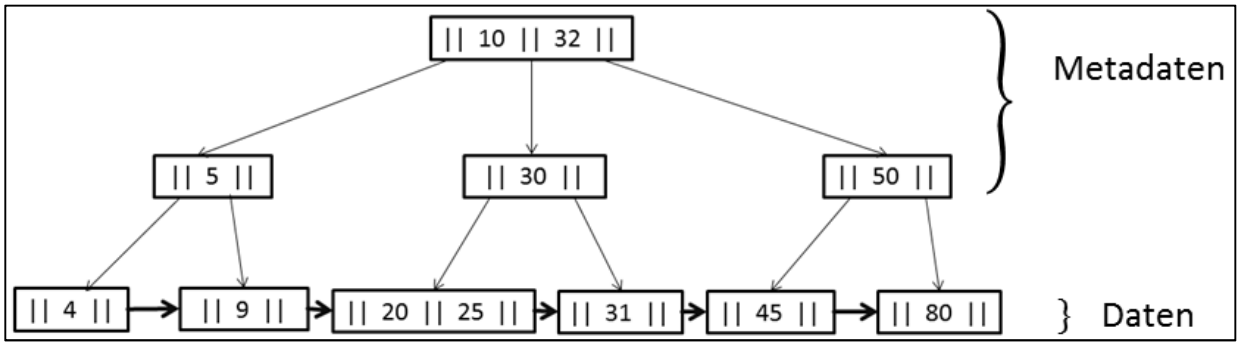
Wenn wir nun den Schlüssel k_2 aus dem inneren Knoten löschen wollen, gehen wir wie folgt vor: Wir ermitteln zunächst den symmetrischen Vorgänger und Nachfolger. Anschließend ersetzen wir k_2 durch den symmetrischen Vorgänger oder Nachfolger und löschen diesen anschließend aus dem Blatt. Die Wahl ob der Vorgänger oder Nachfolger gewählt wird, hängt von der Anzahl der Schlüssel der jeweiligen Knoten von k_{max} bzw. k_{min} ab.

Aufwand: Da wir nun wissen wie B-Bäume implementiert sind, können wir nun dessen Aufwand betrachten. Der Aufwand ist $O(\log_t n)$. Dieser ist wesentlich kleiner als der von binären Bäumen (dort war der Aufwand $O(\log_2 n)$), da die Basis t in der Realität viel größer ist als 2. Ein weiterer Vorteil ist, dass B-Bäume nicht entarten können wie binäre Bäume, da Sie stets vollständig ausbalanciert sind. Somit kann der Aufwand nicht gegen $O(n)$ konvergieren. Dies erfordert allerdings nach jedem Einfügen und Löschen von Schlüsseln eine Reorganisation der Elemente, was zwar mit einem sehr kleinen Aufwand verbunden ist, aber extrem häufig vorkommt.



Entarteter Baum

Beispiel (B^+-Baum): In der Praxis werden häufig B^+-Bäume verwendet. Bei diesen Bäumen verweisen lediglich die Schlüssel der Blätter auf Daten. Alle restlichen Schlüssel sind Metadaten und dienen lediglich dazu, die Schlüssel in den Blättern zu finden. Des Weiteren sind alle Blattknoten von links nach rechts verkettet. Dies hat den Vorteil direkt den nächstgrößeren Schlüssel aufzurufen, ohne erneut durch den Baum wandern zu müssen.



B^+-Baum

Parallele Operationen auf B-Bäume: Wir betrachten das folgende Szenario[1]: Zwei Prozesse versuchen gleichzeitig auf den B-Baum (Abb. 15) zuzugreifen. Der erste Prozess möchte die Datei 12 suchen. Der zweite Prozess möchte die Datei 3 einfügen. Der erste Prozess geht in den Knoten B. Nun fügt der zweite Prozess die 3 ein und spaltet den Knoten B in zwei Knoten B und C (siehe Abb. 16), wobei der erste Prozess in B bleibt (roter Pfeil). Da sich der erste Prozess nun in einem Blatt ohne 12 befindet wird die Suche fälschlicherweise erfolglos beendet.

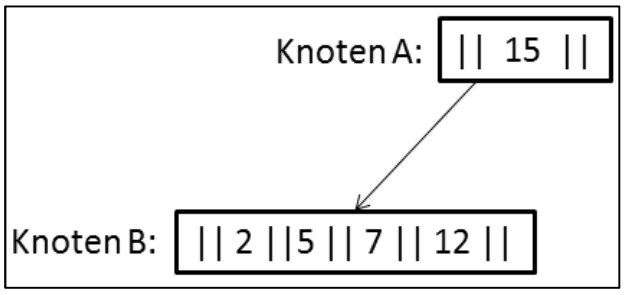


Abbildung 15

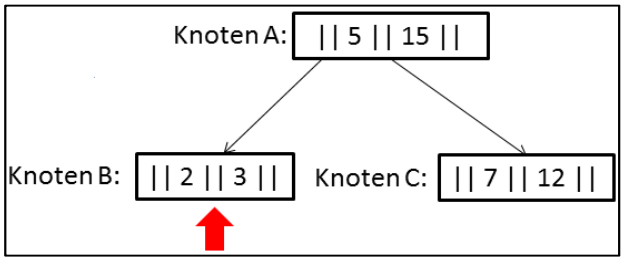


Abbildung 16

1. Modifiziertes Beispiel von: http://wwwbayer.in.tum.de/lehre/WS2001/HSEM-bayer/BTreesAusarbeitung.pdf

Einfachste Lösung: Jeder Prozess markiert die Wurzel wie folgt mit einer Sperre. Jeder Leseprozess markiert sie mit einem S. Die Sperre S lässt dann nur noch lesende Prozesse zu, da diese den Baum nicht verändern. Jeder schreibende Prozess markiert die Wurzel mit einem X. Die Sperre X sperrt alle anderen Prozesse, so dass der schreibende Prozess die Struktur des Baumes verändern kann.

Mit dieser Lösung entsteht aber ein neues Problem. Wenn immer ein lesender Prozess im Baum ausgeführt wird, kann nie geschrieben werden. Die schreibenden Prozesse würden also verhungern und der Baum könnte nicht aktualisiert werden.

Das nächste Kapitel löst unser jetziges Problem.

Copy-On-Write

Copy on Write bedeutet Kopieren beim Schreiben und ist ein Verfahren um schreibende Prozesse auf Dateisystemen problemlos ausführen zu können. Bei diesem Verfahren greift jeder lesende Prozess auf das originale Dateisystem zu, da das System bei diesen Prozessen nicht verändert wird. Bei den schreibenden Prozessen wird zunächst eine Schattenkopie des betroffenen Teilbaumes des Systems erstellt (Abb. 17) und anschließend auf dieser Kopie der Schreibvorgang ausgeführt. Wenn dieser erfolgreich ausgeführt wurde wird der alte Pfad aktualisiert und die weiteren Prozesse können auf den neuen Teilbaum zugreifen (Abb. 18).
Das Copy-On-Write Verfahren bietet also die Möglichkeit lesende und schreibende Prozesse parallel ausführen zu können.

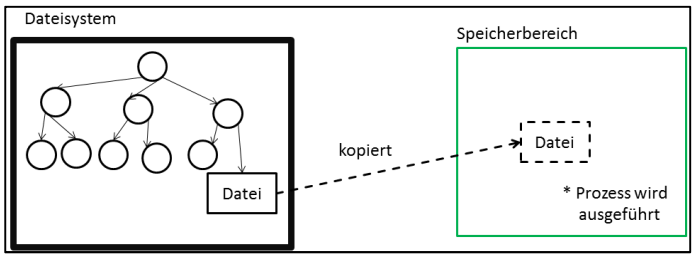


Abbildung 17

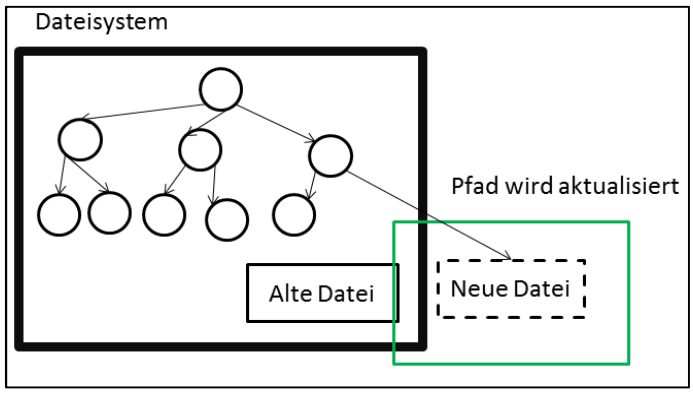


Abbildung 18

Shadowing/ Cloning

Das Shadowing beziehungsweise Cloning ist ein Verfahren, welches zur Laufzeit eine exakte Schattenkopie des Dateisystems auf einer anderen Serverinstanz erstellt. Im Notfall kann diese mit einem minimalen Zeitverlust aktiviert werden und die Produktionsdatenbank ersetzen. Dieses Verfahren wird unter anderen auch beim Copy-on-Write Verfahren genutzt.

Vorteile: Das Shadowing bietet einen zusätzlichen Schutz zu Backups. Bei einem Systemausfall, kann die Schattenkopie von einer anderen Serverinstanz mit minimalem Zeitverlust aktiviert werden. Des Weiteren läuft das Verfahren ohne Verwaltungsaufwand im Hintergrund. Da die Schattenkopie ein exaktes Abbild der Produktionsdatenbank ist, benötigt diese wesentlich weniger Speicher als Logdateien.

Nachteile: Wie bereits oben erwähnt, ist dieses Verfahren nur ein zusätzlicher Schutz zu Backups. Das liegt daran, dass beim Shadowing nicht vor Benutzerfehlern geschützt wird und es somit kein ganzes Sicherheitssystem ersetzt. Zudem lässt sich die Datenbank nicht zurücksetzen. Dadurch, dass die Kopie live erstellt bzw. aktualisiert wird, entsteht ein kleiner Leistungsabfall, da jeder Prozess zweimal ausgeführt wird.

Stand der Wissenschaft (BTRFS)

BTRFS steht für B-Tree Filesystem und ist ein Copy-on-Write Dateisystem von der Oracle Corporation, welches mit B-Bäumen arbeitet. Das System ist seit 2007 in der Entwicklungsphase (Stand 29.April 2013). Ursprünglich sollte es das ext-3 System ersetzen. Da es nun bereits ext-4 Systeme gibt, ist geplant, dass diese nun von BTRFS ersetzt werden. BTRFS hat viele neue Features. Beispielsweise ist es möglich, das System während des Betriebs zu defragmentieren oder Datenkompressionen vorzunehmen. Des Weiteren verfügt das System über einen erweiterten Speicherbereich von 2^{64} Byte (= 16 Exbibyte)*Quelle: http://de.wikipedia.org/wiki/Btrfs. Aufgrund dieser neuen Features speichert BTRFS effizienter als ext-3/4 Systeme. Zusätzlich ist es auch für Solid-State-Drives optimiert und in der Lage selbstständig Fehler zu finden und diese zu beheben.

BTRFS speichert in zwei Baumstrukturen. Eine für Verzeichnis- und Dateinamen und die andere für Datenblöcke. In der unteren Tabelle wurden, auf denselben Testrechner, ein paar Testwerte von den 3 Systemen ext-3/4 und BTRFS ermittelt.

	EXT-3	**EXT-4**	**BTRFS**
Anlegen von acht Dateien á 1 GByte			
Zeit	155.8s	145.1s	120.6s
Durchsatz beim Schreiben	55.4 Mbyte	59.3 MByte	68.5 Mbyte
Löschen von acht Dateien á 1GByte			
Zeit	11.87s	0.33s	0.63s
10 000 zufällige Lese-/ Schreiboperationen in 8GByte			
Operationen/s	80.0s	80.7	115.2s

Werte von **http://www.heise.de/open/artikel/Snapshots-Subvolumes-Performance-224650.html**

Wie wir an den Performancewerten sehen können, ist BTRFS, beim Anlegen der Daten mehrere Sekunden schneller als EXT-3/4. Beim Löschen jedoch braucht das System ca. doppelt so lange wie EXT-4. Wir können daraus schließen, dass BTRFS sehr hohes Potenzial hat, jedoch kein Performancewunder ist.

Die erste stabile Version 1.0 von BTRFS wurde bereits 2008 geplant, wurde aber seitdem noch nicht veröffentlicht (Stand 29.04.2013). Die Betaversion gibt es aber schon seit dem 9. Januar 2009. Es wurde in den Linuxkernel 2.6.29 das erste Mal aufgenommen.

Zusammenfassung

Wir haben zwei Möglichkeiten behandelt große Dateisysteme zu verwalten. Die Hashtabellen und die B-Bäume. Die Hashtabelle hat einen Aufwand von $O(1)$ und ist theoretisch enorm schnell. In der Praxis jedoch entarten solche Systeme sehr schnell und der Aufwand konvergiert gegen $O(n)$. B-Bäume verwalten ihre Daten in großen Baumstrukturen. Der Aufwand hängt bei diesen Systemen von ihrer Ordnung ab. Da diese aber relativ groß ist, bleibt der Aufwand relativ klein. Weiterhin sind B-Bäume vollständig ausbalanciert, was verhindert, dass die Struktur ausarten kann. Das Copy-on-Write Verfahren ermöglicht mehrere parallele Zugriffe auf die Datenbank und nutzt dabei live Kopien der Produktionsdatenbank. Das aktuellste System zur Verwaltung großer Dateisystemen ist derzeit BTRFS. BTRFS bietet viele Features und soll zeitnah ext-3/4 ersetzen. Es befindet sich jedoch seit 2007 in der Entwicklungsphase und ist momentan nicht für den produktiven Gebrauch bestimmt.

Quellen

B-Baum

- http://de.wikipedia.org/wiki/B-Baum (19.04.2013)
- http://en.wikipedia.org/wiki/B-tree (19.04.2013)
- http://martin-thoma.com/b-baume/ (19.04.2013)
- http://wwwbayer.in.tum.de/lehre/WS2001/HSEM-bayer/BTreesAusarbeitung.pdf (26.04.2013)

Shadowing/Cloning

- http://www.ibexpert.net/ibe_de/index.php?n=Doku.DatenbankShadow (19.04.2013)
- http://www.searchdatacenter.de/themenbereiche/storage/storage-management/articles/153256/index2.html (19.04.2013)
- http://msdn.microsoft.com/de-de/library/ms189852.aspx#Overview (26.04.2013)
- http://www.searchsecurity.de/themenbereiche/applikationssicherheit/datenbank-sicherheit/articles/119874/ (3.5.2013)
- http://winswitch.org/documentation/shadow.html (3.5.2013)

Hashtabelle

- http://de.wikipedia.org/wiki/Hashtabelle (23.04.2013)
- SE 1 Script

Copy-on-Write

- http://de.wikipedia.org/wiki/Copy-On-Write (3.5.2013)
- http://en.wikipedia.org/wiki/Copy-on-write (3.5.2013)
- http://www.it-business.de/glossar/Kopieren%20beim%20Schreiben/articles/194011/ (03.05.2013)

BTRFS

- http://de.wikipedia.org/wiki/Btrfs (6.5.2013)
- http://www.oracle.com/us/technologies/linux/btrfs-features-1405320.pdf (6.5.2013)
- https://btrfs.wiki.kernel.org/index.php/Main_Page (6.5.2013)
- http://www.heise.de/open/artikel/Snapshots-Subvolumes-Performance-224650.html (6.5.2013)