Ludwig-Maximilians-Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung

Anfragebearbeitung und Indexstrukturen in Datenbanksystemen

im Wintersemester 2012/2013

Prof. Dr. Hans-Peter Kriegel

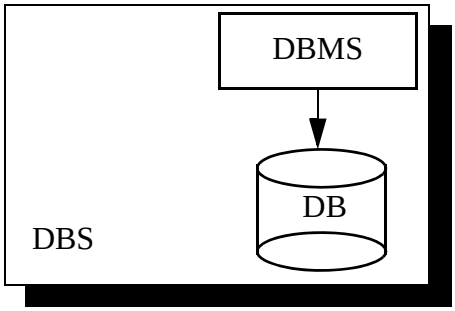
1 Einleitung

Um die Anfragen und Operationen in einem Datenbanksystem zu unterstützen, müssen die Daten durch entsprechende *Datenstrukturen* und *Speicherungsverfahren* geeignet organisiert werden. Datenstrukturen und Speicherungsverfahren, die den besonderen Anforderungen von Datenbanksystemen gerecht werden, werden als *Index- und Speicherungsstrukturen* bezeichnet.

1.1 Datenbanksysteme

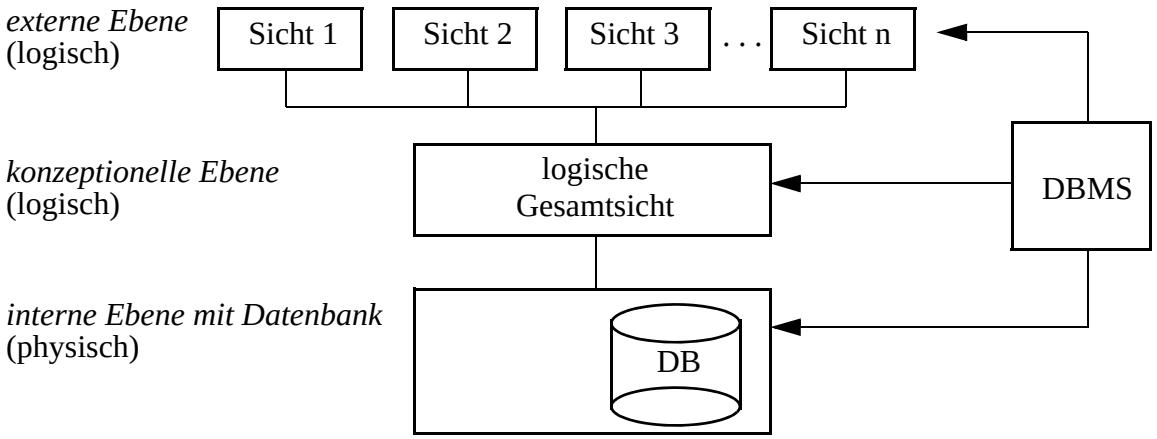
Ein ***Datenbanksystem (DBS)*** besteht aus:

- ***Datenbank (DB)***
  Sammlung aller gespeicherten Daten samt ihrer Beschreibung.
- ***Datenbank-Managementsystem (DBMS)***
  Programmsystem, das die DB verwaltet, fortschreibt und alle Zugriffe auf die DB regelt.



Architekturebenen eines Datenbanksystems:

- ***Externe Ebene***
  umfaßt alle *individuelle Sichten von Benutzern* oder Benutzergruppen auf die Datenbank.
- ***Konzeptionelle Ebene***
  repräsentiert die *logische Gesamtsicht* aller Daten in der Datenbank.
- ***Interne Ebene***
  legt *internen Aufbau* und *Zugriff auf die Daten* fest *(physische Datenorganisation)*.



Beschreibung der Ebenen:

- durch *externes, konzeptionelles und internes Schema*.

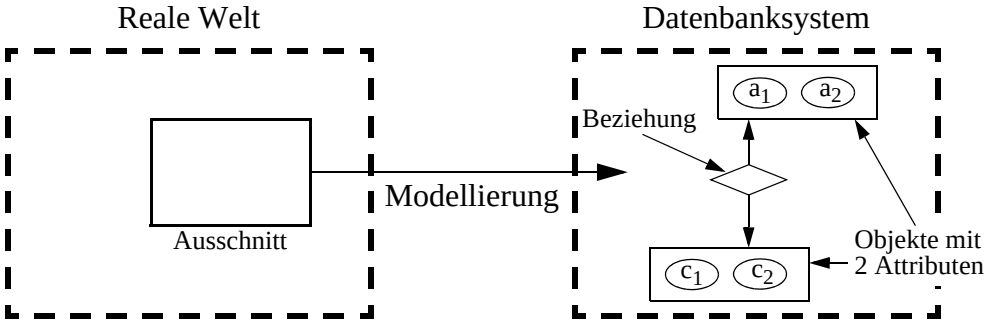
Verbindung der Ebenen:

- durch *Transformationen*
  (Schnittstellen, die die einzelnen Anfragen, Operationen und Resultate transformieren).

1.2 Objekte

Modellbildung

Für die Abbildung eines Ausschnittes der realen Welt in ein DBS ist eine Abstraktion, eine ***Modellierung*** dieser Welt notwendig.



Beschreibung der Welt im Modell durch

- ***Objekte (Entities)*** mit
- ***Eigenschaften (Attributen)*** und
- ***Beziehungen*** untereinander.

Datensätze

- Objekte werden durch eine Reihe von Eigenschaften ***(Attribute)*** beschrieben.

  Anzahl der Sitzplätze

- Jedem Attribut a ist ein ***Wertebereich*** *(Domain)* A zugeordnet.

  CARDINAL

- Damit kann die *Ausprägung eines bestimmten Objektes* als *geordnetes k-Tupel*
  $(a_1 , ... , a_k)$
  beschrieben werden.

  a_i ist der ***Attributswert*** des i-ten Attributs; es gilt $a_i \in A_i$. k ist die ***Dimension***.

  (VW, Passat, silbergrau, 5)

  Dieses Tupel wird ***logischer Datensatz (Record)*** genannt.

- In der Datenbank physisch gespeicherte Datensätze werden oft um Verwaltungsinformationen angereichert und komprimiert ***(physische Datensätze)***.

  (VW, Passat, silbergrau, 5) $\rightarrow$ (4890, 67, 203, 5)

Dateien

- Gleichartige Objekte (d.h. mit gleichen Attributen) werden auf *logischer Ebene* in ***Entity-Sets*** zusammengefaßt.

  { (VW, Passat, silbergrau, 5), (VW, Golf, schwarz, 4), (VW, Jetta, schwarz, 5), ... }

- Datensätze eines Entity-Sets werden i.allg. auf *physischer Ebene* in einer ***Datei (File)*** zusammengefaßt und gespeichert.

1.3 Anfragen

Schlüssel

- ***Schlüssel (Key)***
  Ein oder mehrere Attribute, die einen Datensatz in einer Datei eindeutig identifizieren.
  (Name, Vorname, Geburtsdatum, Geburtsort)
- ***Primärschlüssel***
  Ein Attribut, das einen Datensatz in einer Datei eindeutig identifiziert.
  (Fahrgestellnummer)
- ***Sekundärschlüssel / Suchschlüssel***
  Ein oder mehrere Attribute, nach denen ein Datensatz gesucht werden kann.
  Sie brauchen einen Datensatz aber nicht eindeutig zu identifizieren.
  (Fahrzeugmodell, Farbe)

Anfragen

- ***Primärschlüsselsuche***
  Eine Anfrage über ein Suchattribut x_i, wobei x_i Primärschlüssel ist.
- ***Sekundärschlüsselsuche / Multiattributssuche***
  Eine Anfrage über einen Sekundärschlüssel.

Typen von Anfragen:

- ***Exact Match Query***
  $(x_1 , ... , x_k)$
  spezifiziert k Attribute exakt.
- ***Partial Match Query***
  $(* , x_{i1} , * , ... , x_{i2} , ... , x_{is})$
  spezifiziert Werte für $s < k$ Attribute; k-s Attribute bleiben unspezifiziert (*).
- ***Range Query***
  $([u_1,o_1] , ... , [u_k,o_k])$
  spezifiziert k Bereiche mit $u_i \leq o_i$, $1\ i \leq k$.
- ***Partial Range Query***
  $(* , [u_{i1},o_{i1}] , * , ... , [u_{is},o_{is}])$

Nichtstandard-Datenbanksysteme müssen weitere Anfragen unterstützen (siehe später).

Kernfrage:

Wie finden wir effizient (schnell) die gewünschten Daten in einer Datenbank ?

Lösungsprinzip:

Durch Einsatz geeigneter Datenstrukturen und Speicherungsverfahren,
d.h. *Index- und Speicherungsstrukturen*.

1.4 Physische Speicherung von Daten

Bevor wir die Anforderungen an die Datenstrukturen und Speicherungsverfahren spezifizieren können, müssen wir kurz die physische Speicherung von Daten genauer betrachten.

Anforderungen

- ***Persistenz***
  dauerhafte Speicherung der Daten.
- ***Verwaltung sehr großer Datenmengen***
  Datenvolumen im GigaByte-Bereich**.**

⇒ Speicherung auf ***Sekundärspeicher***, in der Regel auf *Magnetplatten*.

⇒ Nur kleinere Teile der Daten im *Hauptspeicher*:

- Hauptspeicher reicht nicht aus.
- Es ist unzweckmäßig, eine Datei vollständig vom Sekundärspeicher in den Hauptspeicher zu laden, um beispielsweise nur eine Anfrage zu beantworten.

Aufbau von Plattenspeichern

Plattenspeicher sind (wie auch andere Speichergeräte) *seitenorientiert*:

Seiten (Blöcke)

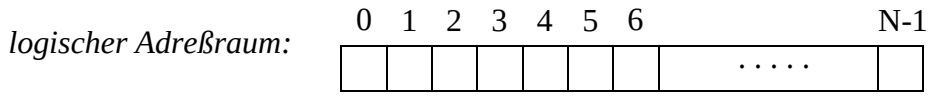
- *Kleinste Transfereinheit,* die zwischen Haupt- und Sekundärspeicher übertragen wird.
- *Wahlfreier (direkter) Zugriff.*
- *Feste Größe*

  zwischen 128 Byte und 8 KByte.
- Die Größe der Seiten ist Kompromiß zwischen *hoher Datenrate* (Anzahl der übertragenen Bytes pro Zeiteinheit) und *guter Plattenausnutzung*:
  - große Seiten = hohe Datenrate
  - kleine Seiten = gute Plattenausnutzung
- Eine Datei verteilt sich je nach Größe auf mehrere Seiten; jede Datei nutzt eine Seite exklusiv, d.h. auf einer Seite befinden sich nur Datensätze einer Datei und damit eines Entity-Sets.

Logischer Aufbau

- Adressierung durch das DBS über *logische Seitennummern*.

  Diese werden vom Betriebssystem auf die tatsächliche physische Adresse der Seite auf dem Plattenspeicher (z.B. Zylinder-, Spur- und Sektornummer) transformiert.

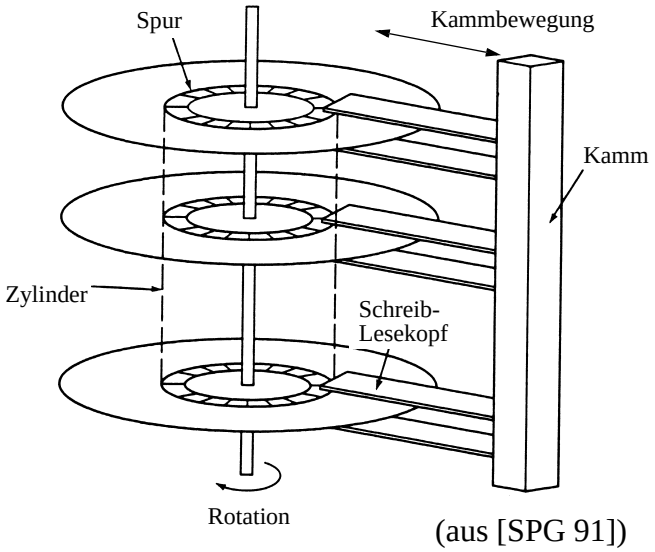


Physischer Aufbau eines Magnetplattenspeichers

- Eine Reihe übereinanderliegender, rotierender Magnetplatten.
- Strukturierung:

  Zylinder, *Spur* und *Sektor*

- Zugriff erfolgt über einen Kamm mit Schreib-/Leseköpfen, der quer zur Rotation bewegt wird.



(aus [SPG 91])

Zugriff auf Seiten

Phasen:

- *Positionierung des Schreib-/Lesekopfes*
  Zeit für die Kammbewegung [3,9 ms]
- *Warten auf den Sektor / Seite*
  ∅ halbe Rotationszeit der Platte [2 ms]
- *Übertragung der Seite*
  Zeit für Schreiben bzw. Lesen [0,02 - 0,04 ms/ 4KByte Seite]
- *Kontrolle der Übertragung*
  Zeit des Platten-Controllers [0,1 ms]

In eckigen Klammern sind die Zeiten einer typischen Server-Platte angegeben (Seagate Cheetah 15K.7, 600 GBytes).

Zeit für Zugriff auf eine Seite >> Zeit für Operation im Hauptspeicher !

Entwicklung:

- Die Kapazität von Plattenspeichern hat sich drastisch erhöht (Faktor 100 in den letzten 10 Jahren)
- Die Zugriffszeit hat sich relativ dazu kaum verändert (Faktor 2 in den letzten 10 Jahren).

Leistungsmaß:

- Annahme: der Zugriff auf Seiten erfolgt unabhängig voneinander.

⇒ Die Anzahl der Seiten, auf die zugegriffen wird, ist ein geeignetes Leistungsmaß für spätere Untersuchungen.

1.5 Indexstrukturen

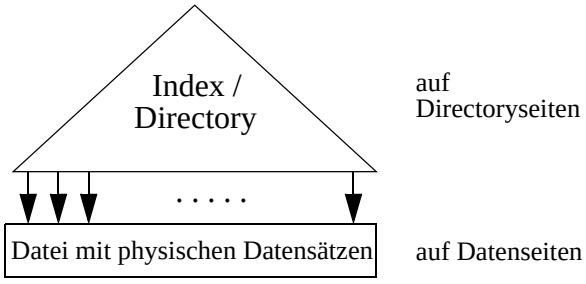
Um Anfragen und Operationen effizient durchführen zu können, setzt die interne Ebene des Datenbanksystemes geeignete Datenstrukturen und Speicherungsverfahren ***(Indexstrukturen)*** ein.

Aufgaben

- *Zuordnung eines Suchschlüssels* zu denjenigen physischen Datensätzen, die diese Wertekombination besitzen,
  d.h. Zuordnung zu der oder den Seiten der Datei, in denen diese Datensätze gespeichert sind.
  (VW, Golf, schwarz, M-ÜN 40) → (logische) Seite 37
- *Organisation der Seiten* einer Datei unter dynamischen Bedingungen.
  Überlauf einer Seite ⇒ Aufteilen der Seite auf zwei Seiten

Index / Directory

- *Strukturinformation* zur Zuordnung von Suchschlüsseln und zur Organisation der Datei.
  - ***Directoryseiten***
    Seiten in denen das Directory gespeichert wird.
  - ***Datenseiten***
    Seiten mit den eigentlichen physischen Datensätzen.



Klassen

- ***Datenorganisierende Strukturen***
  Organisiere die Menge der tatsächlich auftretenden *Daten (Suchbaumverfahren)*.
- ***Raumorganisierende Strukturen***
  Organisiere den Raum, in den die Daten eingebettet sind *(dynamische Hash-Verfahren).*
- ***Hybride Strukturen***
  Kombination beider Vorgehensweisen *(Hash-Bäume)*.

1.6 Allgemeine Anforderungen an Indexstrukturen

- ***Effizientes Suchen***

  Häufigste Operation in einem DBS: *Suchanfragen.*

  ⇒ Insbesondere Suchoperationen müssen mit wenig Seitenzugriffen auskommen.

  Beispiel: unsortierte sequentielle Datei

  - Einfügen und Löschen von Datensätzen werden effizient durchgeführt.
  - Suchanfragen müssen ggf. die gesamte Datei durchsuchen.

  ⇒ Eine Anfrage sollte daher mit Hilfe der Indexstruktur möglichst schnell zu der Seite oder den Seiten hingeführt werden, wo sich die gesuchten Datensätze befinden.

- ***Dynamisches Einfügen, Löschen und Verändern von Datensätzen***

  Der Datenbestand einer Datenbank verändert sich im Laufe der Zeit.

  ⇒ Verfahren, die zum Einfügen oder Löschen von Datensätzen eine Reorganisation der gesamten Datei erfordern, sind nicht akzeptabel.

  Beispiel: sortierte sequentielle Datei

  - Das Einfügen eines Datensatzes erfordert im schlechtesten Fall, daß alle Datensätze um eine Position verschoben werden müssen.
  - Folge: auf alle Seiten der Datei muß zugegriffen werden.

  ⇒ Das Einfügen, Löschen und Verändern von Datensätzen darf daher nur *lokale Änderungen* bewirken.

- ***Ordnungserhaltung***

  Datensätze, die in ihrer Sortierordnung direkt aufeinander folgen, werden oft gemeinsam angefragt.

  ⇒ In der Ordnung aufeinanderfolgende Datensätze sollten in der gleichen Seite oder in benachbarten Seiten gespeichert werden.

- ***Hohe Speicherplatzausnutzung***

  Dateien können sehr groß werden.

  ⇒ Eine möglichst *hohe Speicherplatzausnutzung* ist wichtig:

  - Möglichst geringer Speicherplatzverbrauch.
  - Im Durchschnitt befinden sich mehr Datensätze in einer Seite, wodurch auch die Effizienz des Suchens steigt und die Ordnungserhaltung an Bedeutung gewinnt.

- ***Implementierbarkeit, Nutzen des Implementationsaufwandes***
  - *Implementierbarkeit*
  - *Robustheit*
  - Der Mehraufwand der Implementierung einer komplexen Struktur A gegenüber einer einfachen Struktur B sollte in einem vertretbaren Verhältnis zum Nutzen (Effizienzgewinn) stehen.

1.7 Literatur

Derzeit gibt es kein Lehrbuch, welches sich ausschließlich mit Index- und Speicherungsstrukturen für Datenbanksysteme befaßt. Somit ist man in der Regel auf die Originalliteratur angewiesen, in der die Verfahren vorgestellt wurden. Die entsprechenden Literaturstellen werden in Rahmen dieses Skriptes genannt.

In den meisten ***Lehrbüchern über Datenbanksysteme oder Datenstrukturen*** gibt es Abschnitte oder Kapitel, die sich nur sehr oberflächlich mit Indexstrukturen für Standard-Datenbanksysteme befassen.

Über die Speicherung von Daten und Dateien auf dem Sekundärspeicher kann man in ***Literatur über Betriebssysteme oder Rechnerarchitektur*** nachlesen, beispielsweise in:

[HPG 02] Hennessy J.L., Patterson D.A., Goldberg D.: *'Computer Architecture: A Quantitative Approach'*, ISBN 1558605967, 2002.

[Sie 90] Sierra, H.M.: *'An Introduction to Direct Access Storage Devices'*, Academic Press, 1990.

[SGG 02] Silberschatz A., Galvin P., Gagne G.: *'Operating System Concepts, Sixth Edition'*, John Wiley & Sons, 2002.

[Tan 02] Tanenbaum A.S.: *'Moderne Betriebssysteme, 2. Auflage'*, Pearson Studium, 2002.

2 Baumstrukturen zur Primärschlüsselsuche

An dieser Stelle werden B-Bäume und B*-Bäume als bekannt vorausgesetzt.

Häufig tritt in Datenbankanwendungen neben der Primärschlüsselsuche auch sequentielle Verarbeitung auf.

Beispiele für sequentielle Verarbeitung:

- *Sortiertes Auslesen aller Datensätze*, die von einer Indexstruktur organisiert werden.
- *Unterstützung von Bereichsanfragen* der Form:
  "Nenne mir alle Studenten, deren Nachname im Bereich [Be ... Brz] liegt."

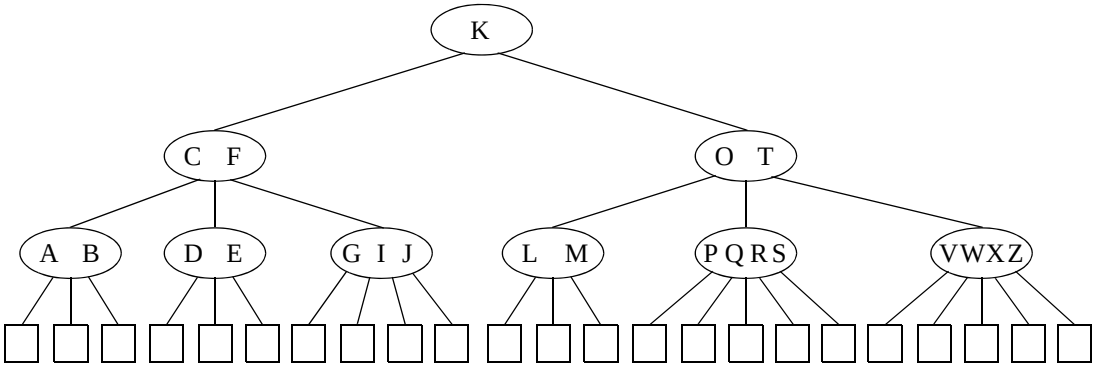
⇒ Die Indexstruktur sollte die *sequentielle Verarbeitung* unterstützen,
d.h. die Verarbeitung der Datensätze in aufsteigender Reihenfolge ihrer Primärschlüssel.

Betrachten wir das Beispiel eines B-Baumes:

Definition: B-Baum der Ordnung *m* (Bayer und McCreight (1972))

(1) Jeder Knoten enthält höchstens 2*m* Schlüssel.
(2) Jeder Knoten außer der Wurzel enthält mindestens *m* Schlüssel.
(3) Die Wurzel enthält mindestens einen Schlüssel.
(4) Ein Knoten mit *k* Schlüsseln hat genau *k*+1 Söhne.
(5) Alle Blätter befinden sich auf demselben Level.

Beispiel: für einen B-Baum der Ordnung 2



Warum unterstützt dieser B-Baum die folgende Bereichsanfrage nicht effizient?
"Lies die Informationen aller Datensätze im Bereich [B ... V] aus"

Grundidee:

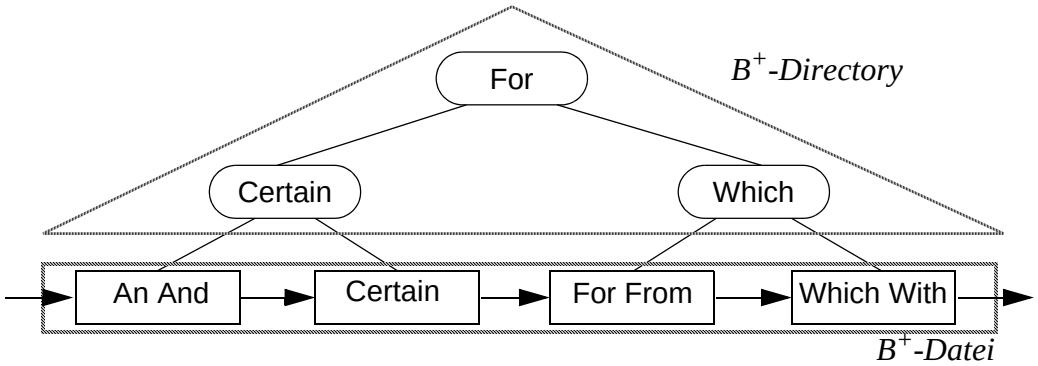
- Trennung der Indexstruktur in *Directory* und *Datei*.
- *Sequentielle Verkettung* der Daten in der Datei.

B^+-Baum

- ***B^+-Datei:***
  - Die Blätter des B^+-Baumes heißen ***Datenknoten*** oder ***Datenseiten***.
  - Die Datenknoten enthalten alle Datensätze.
  - Alle Datenknoten sind entsprechend der Ordnung auf den Primärschlüsseln *verkettet*.

- ***B$^+$-Directory***:
  - Die inneren Knoten des B$^+$-Baumes heißen ***Directoryknoten*** oder ***Directoryseiten***.
  - Directoryknoten enthalten nur noch ***Separatoren*** s.
  - Für jeden Separator s(u) eines Knotens u gelten folgende ***Separatoreneigenschaften***:
    - s(u) > s(v) für alle Directoryknoten v im linken Teilbaum von s(u).
    - s(u) < s(w) für alle Directoryknoten w im rechtenTeilbaum von s(u).
    - s(u) > k(v') für alle Primärschlüssel k(v') und alle Datenknoten v' im linken Teilbaum von s(u).
    - s(u) ≤ k(w') für alle Primärschlüssel k(w') und alle Datenknoten w' im rechten Teilbaum von s(u).

Beispiel: B$^+$-Baum für die Zeichenketten: An, And, Certain, For, From, Which, With.



2.1 Erhöhung des Verzweigungsgrades: Präfix-Bäume

Wichtige Anforderungen an Suchbaumverfahren:

- *Geringe Höhe*

  Die Anzahl der Seitenzugriffe für Anfragen und andere Operationen korrespondiert direkt mit der Höhe des Baumes.

  ⇒ Eine niedrige Baumhöhe ist anzustreben.

- *Hoher Verzweigungsgrad*

  Ein wichtiges Kriterium, um die Höhe des Baumes zu beeinflussen, ist der *Verzweigungsgrad* der Knoten. Eine Erhöhung des Verzweigungsgrades schlägt sich in einer Reduktion der Höhe des Suchbaumes nieder.

  ⇒ Ein hoher Verzweigungsgrad ist anzustreben.

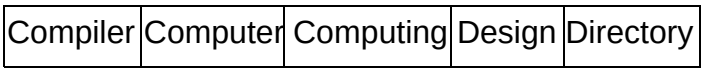
Separatoren mit Präfixeigenschaft: einfache Präfix-B$^+$-Bäume

Beobachtung:

- Im B$^+$-Baum werden *vollständige Schlüssel* als Separatoren verwendet.
- Kürzere Separatoren können die Separatoreneigenschaften genauso gut erfüllen.
- Kürzere Separatoren erhöhen den Verzweigungsgrad.

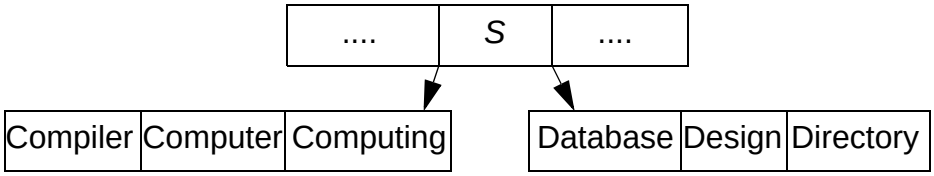
Beispiel:

Ein Datenknoten enthalte folgende Schlüssel und sei schon voll:



Einfügen des Schlüssels “Database”.

⇒ Der Knoten muß in zwei aufgespalten werden ***(Split)***:



Wir können jedes *S* als *Separator* wählen, das die Eigenschaft erfüllt:

$$\text{Computing} < S \leq \text{Database}$$

Präfix-Eigenschaft

Die Schlüssel seien Worte über einem Alphabet und die Ordnung der Schlüssel die lexikographische Ordnung. Dann gilt folgende *Präfix-Eigenschaft*:

Für zwei Schlüssel x und y mit x < y gibt es einen *Präfix* $\overline{y}$ von y mit:

- $\overline{y}$ ist Separator zwischen x und y,
- kein anderer Separator zwischen x und y ist kürzer als $\overline{y}$.

Beispiel (oben): $\overline{y}$ = D.

Einfacher Präfix-B^+-Baum [BU 77]

Ein *einfacher Präfix-B^+-Baum* ist ein B^+-Baum, dessen Directory aus einem B-Baum von Separatoren variabler Länge besteht, die die Präfix-Eigenschaft erfüllen.

Prinzip:

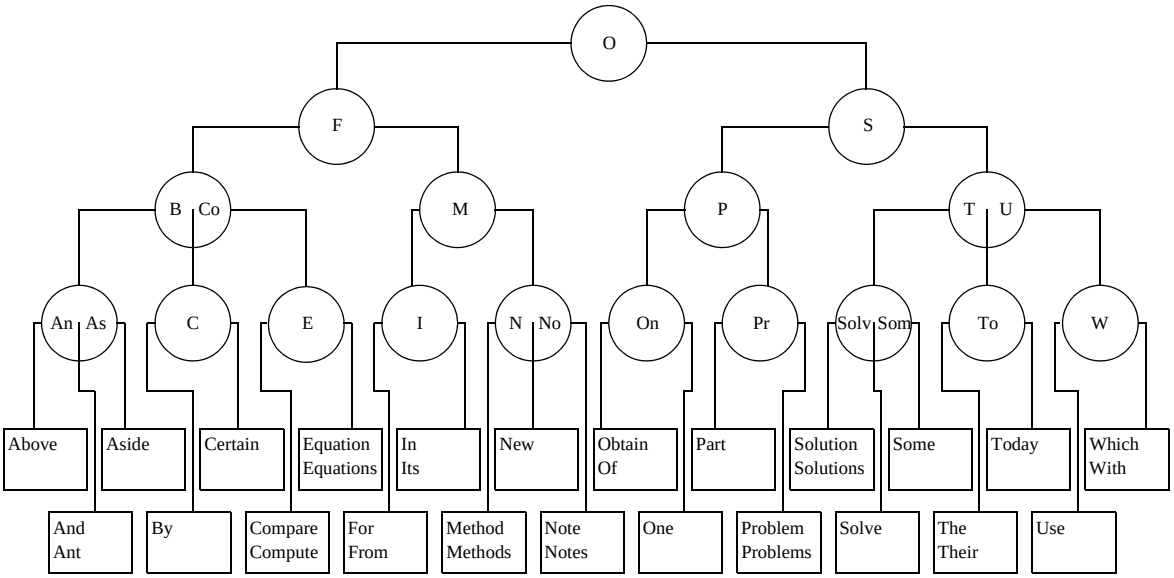
Je kürzer die Separatoren in den Directoryknoten

$\Rightarrow$ desto höher der Verzweigungsgrad,

$\Rightarrow$ desto niedriger der Baum,

$\Rightarrow$ desto besser das Leistungsverhalten.

Beispiel:



Vorgehen beim Split von Knoten:

- ***Split eines Datenknotens***
  Ein Separator mit Präfix-Eigenschaft wird gebildet und in den Vaterknoten eingefügt.
- ***Split eines Directoryknotens***
  Der Separator des überlaufenden Knotens, der die beiden neuen Knoten trennt, wird in den Vater eingefügt.

Split-Intervall

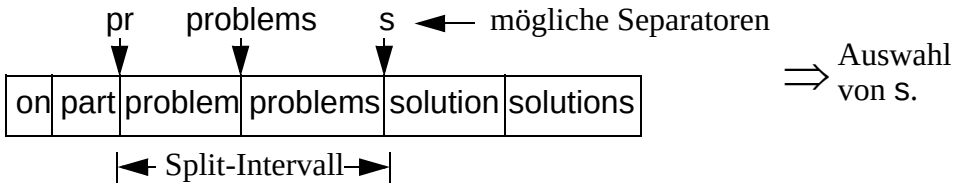
Ziel:

Erhöhung des Verzweigungsgrades.

Idee:

Man teilt eine Seite nicht genau in der Mitte auf, sondern wählt den Splitpunkt aus einem Bereich. Die Wahl des tatsächlichen Splitpunktes erfolgt gemäß der Länge der möglichen Separatoren.

Beispiel:



Split-Intervall σ

Gibt das Intervall an, in dem der Splitpunkt liegt.

Vorgehen:

- Auswahl des kürzesten Separators aus dem Split-Intervall.
- Aufteilen der Knoten gemäß dieses Separators.

σ_{dat}*: Split-Intervall für Datenknoten*

- je größer σ_{dat}, desto weniger Directoryknoten.

σ_{dir}*: Split-Intervall für Directoryknoten*

- je größer σ_{dir}, desto höher der Verzweigungsgrad nahe der Wurzel.

Nachteil:

Speicherplatzausnutzung kann unter 50 % sinken.

Frontkomprimierung: Präfix-B$^+$-Bäume

Ziel:

Weiteres Verkürzen der Separatoren.

Idee:

Zusammensetzen der Separatoren beim Durchlaufen des Baumes von der Wurzel zum gesuchten Blatt.

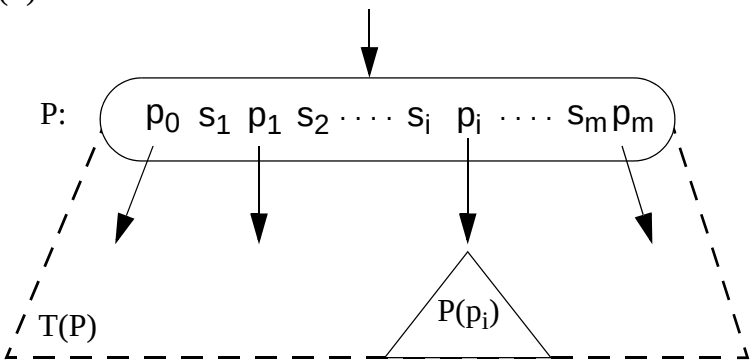
Beobachtung:

Für jeden Directoryknoten P in einem einfachen Präfix-B$^+$-Baum gibt es *Schranken*:

- eine *größte untere Schranke* $\lambda(P)$ und
- eine *kleinste obere Schranke* $\mu(P)$,

s.d. für alle Schlüssel k und alle Separatoren s in T(P) (= Teilbaum mit Wurzel P) gilt:

- $\lambda(P) \leq k < \mu(P)$
- $\lambda(P) \leq s < \mu(P)$



Berechnung von $\lambda(P)$ und $\mu(P)$:

Wurzel:

- $\lambda(R) = b_0$ (b_0 ist der kleinste Buchstabe im Alphabet)
- $\mu(R) = \infty$ (∞ ist größer als jeder Buchstabe im Alphabet)

Sonst:

- $\lambda(P(p_i)) = \lambda(p_i) = \begin{cases} s_i & \text{für } i > 0 \\ \lambda(P) & \text{für } i = 0 \end{cases}$
- $\mu(P(p_i)) = \mu(p_i) = \begin{cases} s_{i+1} & \text{für } i < m \\ \mu(P) & \text{für } i = m \end{cases}$

Beobachtung:

Alle Separatoren und Schlüssel in $T(P(p_i))$ haben einen gemeinsamen (möglicherweise leeren) Präfix.

$g(p_i)$: der *längste gemeinsame Präfix aller Separatoren und Schlüssel in $T(P(p_i))$*

Berechnung des gemeinsamen Präfix $g(p_i)$:

$\overline{k}_i$: der *längste gemeinsame Präfix von $\lambda(p_i)$ und $\mu(p_i)$*
($\overline{k}_i$ kann möglicherweise auch eine leere Zeichenkette sein).

Für ***$g(p_i)$*** gilt:

$$g(p_i) = \begin{cases} \overline{k}_i b_j & \text{wenn } \lambda(p_i) = \overline{k}_i b_j z \text{ und } \mu(p_i) = \overline{k}_i b_{j+1}, \text{ wobei z eine beliebige Zeichenkette ist und der Buchstabe } b_{j+1} \text{ dem Buchstaben } b_j \text{ direkt in der alphabetischen Reihenfolge folgt.} \\ \overline{k}_i & \text{sonst} \end{cases}$$

Beispiele

1. T ($P(p_i)$) enthalte die Schlüssel aab, aac, aaf, aag, aaz.

Seien $\lambda(p_i)$ = aab, $\mu(p_i)$ = ab, dann ist $g(p_i)$ = aa nach dem ersten obigen Fall.

2. T ($P(p_i)$) enthalte nun die Schlüssel aab, aac, aaf, aag, aah.

Seien $\lambda(p_i)$ = aab, $\mu(p_i)$ = aai, dann ist $g(p_i)$ = aa nach dem zweiten obigen Fall.

Bestimmung des Separators aus dem Präfix:

vollständiger Separator $s = g(p_i)\,\overline{s}$

frontkomprimierter Separator
Präfix

- $\lambda(p_i)$, $\mu(p_i)$ und $g(p_i)$ können beim Durchlauf des Baumes von der Wurzel zum Knoten $P(p_i)$ abgeleitet werden.

$\Rightarrow$ Nur noch frontkomprimierte Separatoren sind abzuspeichern.

Präfix-B^+-Baum [BU 77]

- in den Directoryknoten werden nur noch frontkomprimierte Separatoren abgespeichert.
- die Separatoren werden beim Traversieren abgeleitet.

Leistungsvergleich

Ersparnis von Seitenzugriffen bei Primärschlüsselsuche:

einfacher Präfix-B^+-Baum gegenüber B^+-Baum: 25 %

Präfix-B^+-Baum gegenüber einfachem Präfix-B^+-Baum: 2 %

Zusätzlicher Aufwand für Implementierung und CPU-Zeit:

einfacher Präfix-B^+-Baum gegenüber B^+-Baum:

Speicherung und Suche variabel langer Schlüssel

Präfix-B^+-Baum gegenüber einfachem Präfix-B^+-Baum:

Rekonstruktion der Separatoren aus den komprimierten Separatoren

$\Rightarrow$ Der einfache Präfix-B^+-Baum ist der beste Kompromiß zwischen Effizienz und Komplexität der Implementierung.

2.2 Literatur

Bäume (einschließlich B-Bäume) werden in der Regel in ***Lehrbüchern über Datenstrukturen*** dargestellt, z.B. in:

[GD 03] Güting R.H., Dieker S.: *'Datenstrukturen und Algorithmen, 2. Auflage'*, Teubner Verlag, 2003.

[OW 02] Ottmann T., Widmayer P.: *'Algorithmen und Datenstrukturen, 4. Auflage'*, Spektrum Akademischer Verlag, 2002.

[Wir 96] Wirth N.: *'Algorithmen und Datenstrukturen mit Modula-2'*, Teubner Verlag, 1996.

B- und B*-Bäume wurden auch bereits vorgestellt in:

[Knu 99] Knuth D.E.: *'The Art of Computer Programming, Vol. 3'*, Addison Wesley Longman, 1999.

Originalartikel:

[AVL 62] Adel'son-Vel'skii G.M., Landis E.M., Soviet Math, Vol. 3, pp. 1259-63.

[BM 72] Bayer R., McCreight E.M.: *'Organization and Maintenance of Large Ordered Indexes'*, Acta Informatica, Vol. 1, No. 3, 1972, pp. 173-189.

[BU 77] Bayer R., Unterauer K.: *'Prefix B-Trees'*, ACM Transactions on Database Systems, Vol. 2, No. 1, 1977, pp. 11-28.

3 Baumstrukturen zur Sekundärschlüsselsuche

Ziel: Unterstützung von Anfragen über mehrere Attribute
(Sekundärschlüsselsuche = Multiattributssuche).

3.1 Invertierte Listen

In fast allen kommerziell vertriebenen Datenbanksystemen:

Multiattributssuche mit Hilfe ***invertierter Listen***.

- ***Primärindex***
  Index über den Primärschlüssel.
- ***Sekundärindex***
  Index über ein Attribut, das kein Primärschlüssel ist.

  Im Gegensatz zu einem Primärindex beeinflußt der Sekundärindex den Ort der Speicherung eines Datensatzes nicht. Es werden nur Verweise gespeichert.

Konzept der invertierten Listen:

- Für anfragerelevante Attribute werden Sekundärindizes ***(invertierte Listen)*** angelegt.

$\Rightarrow$ Damit steht für jedes relevante Attribut eine eindimensionale Indexstruktur zur Verfügung.

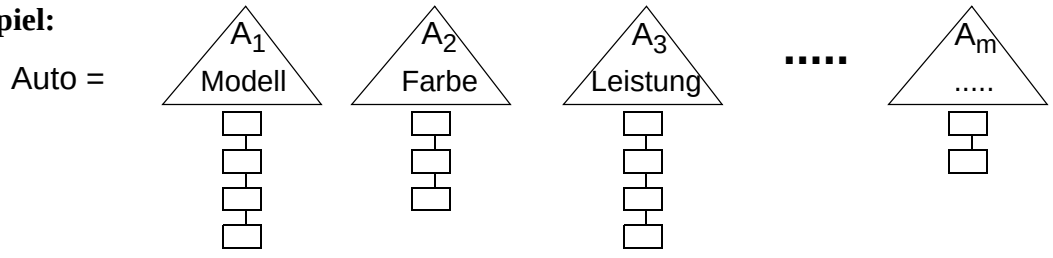
Multiattributssuche für invertierte Listen:

Eine Anfrage spezifiziere die Attribute A_1, ... ,A_m:

- *m Anfragen über m Indexstrukturen*

  Ergebnis:
  m Listen mit Verweisen auf die entsprechenden Antwortkandidaten in der Datei.
- *Mengentheoretische Verknüpfung* (z.B. Durchschnitt) der m Listen gemäß der Anfrage.

Beispiel:



Anfrage: Gesucht sind alle Autos mit Modell = GLD, Farbe = rot und Leistung = 75 PS

Eigenschaften:

- Die Antwortzeit ist nicht proportional zur Anzahl der Antworten.
- Die Suche dauert umso länger, je mehr Attribute spezifiziert sind.

  Ursache für beide Beobachtungen:
  Die Attributswerte eines Datensatzes sind nicht in einer Struktur miteinander verbunden.
- Invertierte Listen sind einigermaßen effizient, wenn die Antwortlisten sehr klein sind.
- Invertierte Listen haben hohe Kosten für Update-Operationen.
- Sekundärindizes beeinflußen die physische Speicherung der Datensätze nicht.

  $\Rightarrow$ Ordnungserhaltung über den Sekundärschlüssel nicht möglich.

  $\Rightarrow$ schlechtes Leistungsverhalten von invertierten Listen.

3.2 Hierarchie von B-Bäumen: MDB-Bäume

Ziel: Speicherung multidimensionaler Schlüssel $(a_k, \ldots, a_1)$ in einer Indexstruktur.

Idee: Hierarchie von Bäumen**,** wobei jede Hierarchiestufe jeweils einem Attribut entspricht.

Notation:

Im folgenden werden für eine einfachere Notation die Attributswerte multidimensionaler Schlüssel absteigend numeriert $(a_k, \ldots, a_1)$.

Außerdem ändert sich die Numerierung der Höhen: Blattknoten eines Baumes haben die Höhe 1 und die Höhe der Wurzel entspricht der Höhe des Baumes.

Multidimensionale B-Bäume (MDB-Bäume) [SO 82]

- k-stufige Hierarchie von B-Bäumen.
- Jede Hierarchiestufe ***(Level)*** entspricht einem Attribut.
  Werte des Attributs a_i werden in Level i $(1 \le i \le k)$ gespeichert.
- Die B-Bäume des Levels i haben die *Ordnung* m_i;
  m_i hängt von der Länge der Werte des i-ten Attributs ab.

Verzeigerung in einem B-Baum:

- linker Teilbaum bzgl. a_i: ***LOSON(a_i)***
- rechter Teilbaum bzgl. a_i: ***HISON(a_i)***

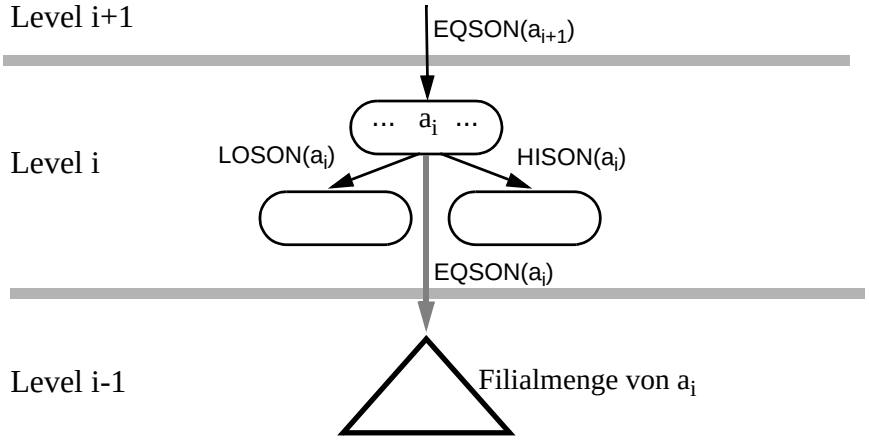
Verknüpfung der Level:

- Jeder Attributswert a_i hat zusätzlich einen *EQSON-Zeiger*:
  EQSON(a_i) zeigt auf den B-Baum des Levels i-1, der die verschiedenen Werte abspeichert, die zu Schlüsseln mit Attributswert a_i gehören.

  Filialmenge

  Für einen *Präfix* $(a_k, \ldots, a_i)$ eines Schlüssels $(a_k, \ldots, a_1)$, i < k, betrachte man die Menge M aller Schlüssel in der Datei mit diesem Präfix. Die Menge der (i-1)-dimensionalen Schlüssel, die man aus M durch Weglassen des gemeinsamen Präfixes erhält, heißt *Filialmenge* von $(a_k, \ldots, a_i)$ (oder kurz: Filialmenge von a_i).

$\Rightarrow$ EQSON(a_i) zeigt auf einen B-Baum, der die Filialmenge von a_i abspeichert.



Exact Match Queries:
können effizient beantwortet werden.

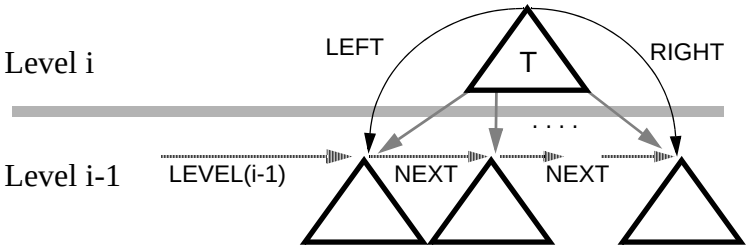
Range, Partial Match und Partial Range Queries:
erfordern zusätzlich sequentielle Verarbeitung auf jedem Level.

Unterstützung von Range Queries:

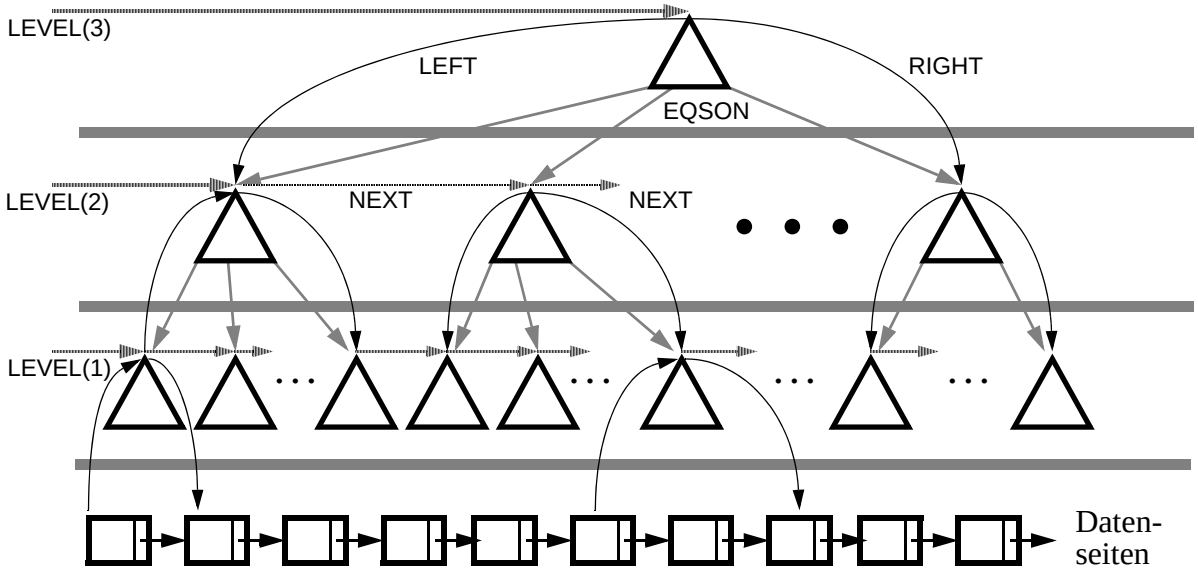
- *Verkettung der Wurzeln* der B-Bäume eines Levels: ***NEXT-Zeiger***.

Unterstützung von Partial Match Queries und Partial Range Queries:

- *Einstiegszeiger für jedes Level:*
  - ***LEVEL(i)*** zeigt für Level i auf den Beginn der verketteten Liste aus NEXT-Zeigern
- *Überspringzeiger* von jeder Level-Wurzel eines Level-Baumes T zum folgenden Level:
  - ***LEFT(T)*** zeigt zur Filialmenge des kleinsten Schlüssels von T
  - ***RIGHT(T)*** zeigt zur Filialmenge des größten Schlüssels von T.



Beispiel: MDB-Baum



Höhenabschätzung

Annahme: Für jedes Attribut gilt:

- Die Werte sind gleichverteilt im zugehörigen Wertebereich.
- Die Werte eines Attributs sind unabhängig von den Werten anderer Attribute *(stochastische Unabhängigkeit der Attribute)*

⇒ Alle Bäume auf demselben Level haben dieselbe Höhe.

⇒ Die maximale Höhe ist O (log N + k).

Einwand: Die Annahmen sind in realen Dateien äußerst selten erfüllt.

Die maximale Höhe eines MDB-Baumes, der die obige Annahme nicht erfüllt, beträgt:

O (k*log N)