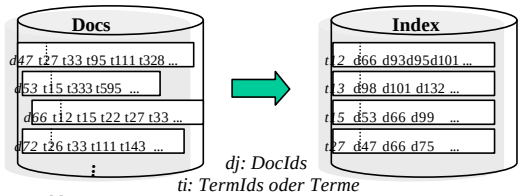
5 Indexstrukturen

5.1 Mehrwegesuchbäume, speziell B*-Bäume
5.2 Digitalbäume, speziell PAT-Tries
5.3 Hashing
5.4 Signaturen

Invertierte Dateien

Konzeptionell:
invertierte Dateien (invertierte Listen) mit binärer Suche nach Suchschlüsseln (Felder von Records, Strings in Texten)



dj: DocIds
ti: TermIds oder Terme

Problem:
Speicherungsorganisation in Plattenblöcken (pagination) und effiziente Implementierung der (binären) Suche für
- Exact-Match-Suche: search (key) returns ids
- Bereichssuche: search (lowkey, highkey) returns ids
- Präfixstringsuche: search (prefix) returns ids

bei dynamischen Updates

Binäre Suchbäume (für Hauptspeicher)

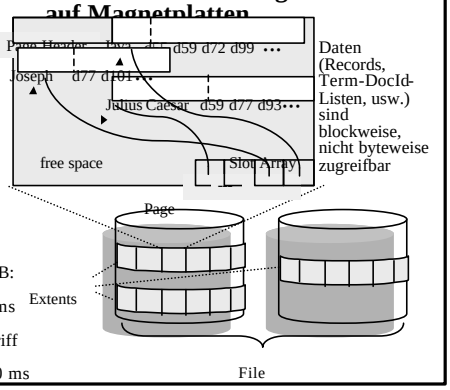
Schlüssel (mit den ihnen zugeordneten Daten, z.B. Terme mit DocId-Listen) bilden die Knoten eines binären Baums mit der Invariante:
für jeden Knoten t mit Schlüssel t.key und alle Knoten l im linken Teilbaum von t, t.left, und alle Knoten r im rechten Teilbaum von t gilt: l.key ≤ t.key ≤ r.key

Suchen eines Schlüssels k:
Traversieren des Pfades von der Wurzel bis zu k bzw. einem Blatt
Einfügen eines Schlüssels k:
Suchen von k und Anfügen eines neuen Blatts
Löschen eines Schlüssel k:
Ersetzen von k durch das „rechteste“ Blatt links von k

Worst-Case-Suchzeit für n Schlüssel: O(n)
bei geeigneten Rebalancierungsalgorithmen (AVL-Bäume, Rot-Schwarz-Bäume, usw.): O(log n)

Exkurs: Blockorientierte Datenorganisation auf Magnetplatten

Files sind Sequenzen von Blöcken (Seiten) fester Größe (z.B. 32KBytes)



Daten (Records, Term-DocId-Listen, usw.) sind blockweise, nicht byteweise zugreifbar

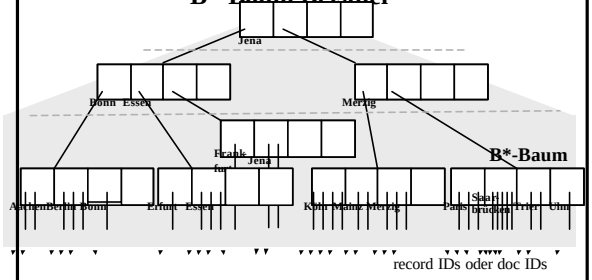
Direkter Zugriff (random I/O) auf 1 Block a 32KB: Latenzzeit + Transferzeit ≈ 10 ms

Sequentieller Zugriff (sequential I/O) auf 10 Blöcke ≈ 20 ms

B*-Bäume: Seitenstrukturierte Mehrwegbäume

- Hohler Mehrwegebaum mit hohem Fanout (⇒ kleiner Tiefe)
- Knoten = Seite auf Platte
- Knoteninhalt:
  - (Sohnzeiger, Schlüssel)-Paare in inneren Knoten
  - Schlüssel (mit weiteren Daten) in Blättern
- perfekt balanciert: alle Blätter haben dieselbe Distanz zur Wurzel
- Sucheffizienz $O(\log_k n/C)$ Seitenzugriffe (Platten-I/Os)
  bei n Schlüsseln, Seitenkapazität C und Fanout k
  pro Baumniveau: bestimme kleinsten Schlüssel ≥ q und suche weiter im Teilbaum links von q
- Kosten einer Einfüge- oder Löschoperation $O(\log_k n/C)$
- mittlere Speicherplatzauslastung bei zufälligem Einfügen: ln 2 ≈ 0.69

B*-Baum-Beispiel



record IDs oder doc IDs

B*-Baum-Definition

Ein Mehrwegbaum heißt B*-Baum der Ordnung (m, m*), wenn gilt:

- Jeder Nichtblattknoten außer der Wurzel enthält mindestens m ≥1 und höchstens 2m Schlüssel (Wegweiser).
- Ein Nichtblattknoten mit k Schlüssel x1, ..., xk hat genau k+1 Söhne t1, ..., t(k+1), so daß
  - für alle Schlüssel s im Teilbaum ti (2≤i≤k) gilt x(i-1) < s ≤ xi und
  - für alle Schlüssel s im Teilbaum t1 gilt s ≤ x1 und
  - für alle Schlüssel im Teilbaum t(k+1) gilt xk < s.
- Alle Blätter haben dasselbe Niveau (Distanz von der Wurzel)
- Jedes Blatt enthält mindestens m* ≥1 und höchstens 2m* Schlüssel.

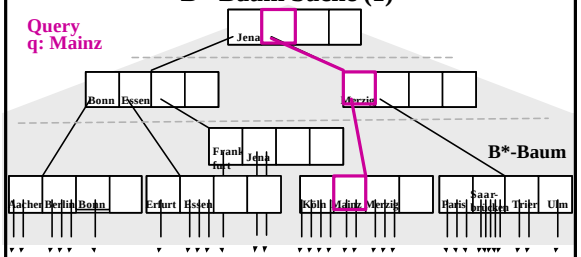
Achtung: Implementierungen verwenden **variabel lange Schlüssel** und eine Knotenkapazität in Bytes statt Konstanten 2m und 2m*

Sonderfall m=m*=1: **2-3-Bäume** als Hauptspeicherdatenstruktur

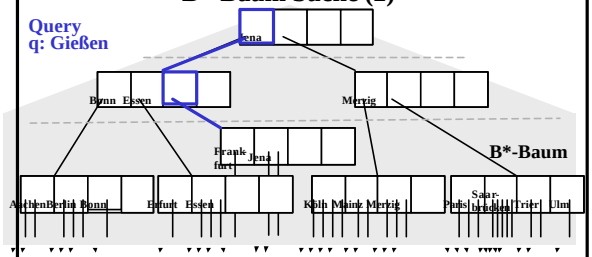
Pseudocode für B*-Baum-Suche

Suchen von Schlüssel s in B*-Baum mit Wurzel t:
t habe k Schlüssel x1, ..., xk und k+1 Söhne t1, ..., t(k+1)
(letzteres sofern t kein Blatt ist)
Bestimme den kleinsten Schlüssel xi, so daß s ≤ xi
if s = xi (für ein i ≤ k) und t ist ein Blatt
then Schlüssel gefunden
else
  if t ist kein Blatt then
    if s ≤ xi (für ein i ≤ k)
    then suche s im Teilbaum ti
    else suche s im Teilbaum t(k+1) fi
  else Schlüssel s ist nicht vorhanden fi
fi

B*-Baum-Suche (1)



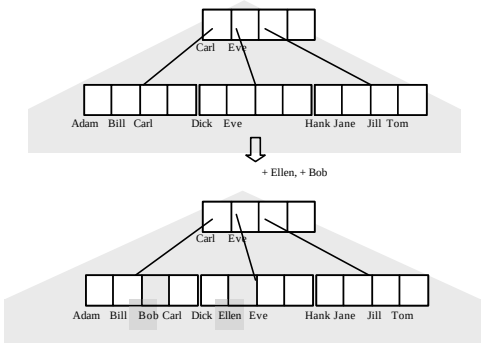
B*-Baum-Suche (2)



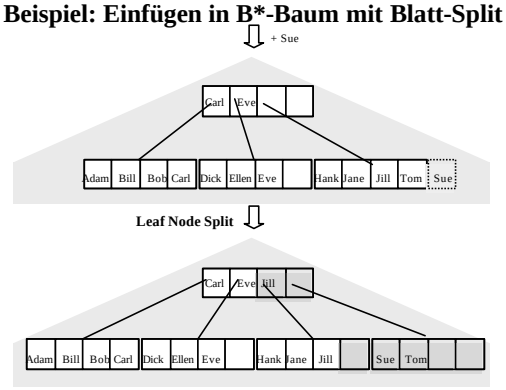
Pseudocode für Einfügen in B*-Baum (Grow&Post)

Suche nach einzufügendem Schlüssel e
if e ist noch nicht vorhanden then
  Sei t das Blatt, bei dem die Suche erfolglos geendet hat
  repeat
    if t hat weniger als 2m* bzw. 2m Schlüssel (d.h. ist nicht voll)
    then füge e in t ein
    else /* *Knoten-Split* */
      Bestimme Median s der 2m* +1 bzw. 2m+1 Schlüssel inkl. e
      Erzeuge Bruderknoten t' /* *Grow-Phase* */
      if t ist Blattknoten then
        Speichere Schlüssel ≤ s in t und Schlüssel > s in t'
      else Speichere Schlüssel < s (mit Sohnzeigern) in t und
                Schlüssel > s (mit Sohnzeigern) in t' fi
      if t ist Wurzel /* *Post-Phase* */
      then Erzeuge neue Wurzel r mit Schlüssel s und Zeigern auf t und t'
      else Betrachte Vater von t als neues t und s (mit Zeiger auf t') als e f
    fi
  until kein Knoten-Split mehr erfolgt
fi

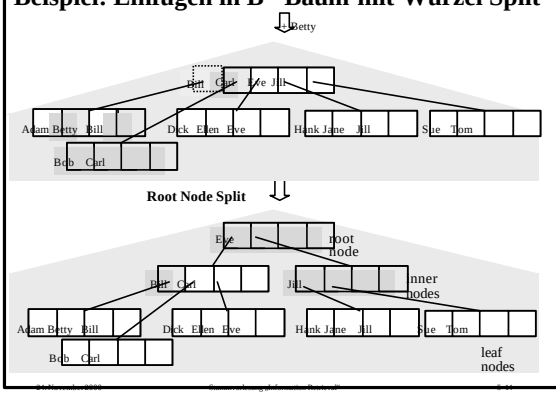
Beispiel: Einfügen in B*-Baum



Beispiel: Einfügen in B*-Baum mit Blatt-Split



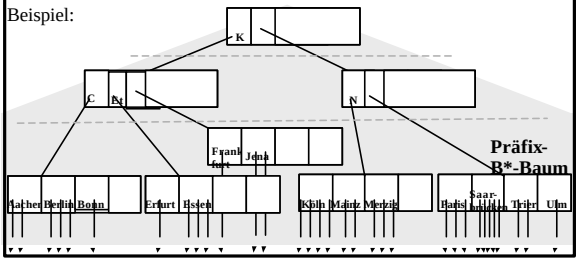
Beispiel: Einfügen in B*-Baum mit Wurzel-Split



Präfix-B*-Bäume für Strings als Schlüssel

Schlüssel in inneren Knoten sind nur **Wegweiser (Router)** zur Partitionierung des Schlüsselraums.
Statt xi = max{s: s ist ein Schlüssel im Teilbaum ti} genügt ein (kürzerer) Wegweiser xi‘ mit si ≤ xi‘< x(i+1) für alle si in ti und alle s(i+1) in t(i+1).
Eine Wahl wäre xi‘ = kürzester String mit der o.a. Eigenschaft.
→ höherer Fanout, potentiell kleinere Baumhöhe

Beispiel:



Weitere Varianten und Einsatz von B*-Bäumen

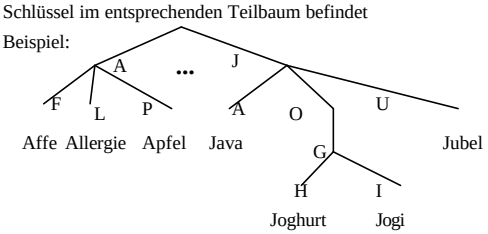
Bitlisten-Index:
Statt Listen von RecordIDs bzw. DocIDs können bei den Schlüsseln in Blattknoten auch (Zeiger auf) komprimierte Bitlisten gespeichert werden, deren Bitpositionen IDs entsprechen und mit einer 1 auf Position i das Vorhandensein von ID i in der Liste anzeigen.

Feldspezifische Indexstrukturen:
Bei unstrukturierten Dokumenten gibt es typischerweise nur einen einzigen Index für alle Terme aller Dokumente.
Bei **strukturierten Daten** - Records mit mehreren Feldern/Attributen – (z.B.: ein Mail-Archiv mit Feldern Sender, Date, Subject, usw.) legt man häufig separate Indexstrukturen für alle Werte eines Feldes oder einer Feldkombination an
(z.B.: einen Index auf Sender, einen Index auf Date, usw.)
Damit kann man konjunktive Anfragen vom Typ
Feld1 = Wert1 And Feld2 = Wert2 And ... bzw.
UntererWert1 ≤ Feld1 ≤ ObererWert1 And ...
gut unterstützen (durch Boolesche Operationen auf ID-Listen).

Digitalbäume (Tries) für Strings als Schlüssel

Mehrwegbaum, der auf Niveau i nach dem i-ten Zeichen im Schlüssel verzweigt (max. Fanout: |Σ|);
die Verzweigungen brechen ab, wenn sich nur noch ein Schlüssel im entsprechenden Teilbaum befindet

Beispiel:

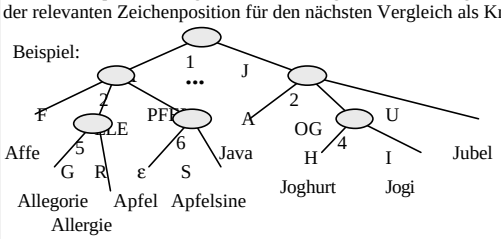


Suchkosten für String w (im Hauptspeicher): O(|w|)
unabhängig von der Anzahl der Schlüssel

PATRICIA-Bäume
(Practical Algorithm for The Retrieval of Information Coded In Alphanumeric)

Nichtverzweigende Äste werden zu einer Kante zusammengefaßt – mit dem entspr. Teilstring oder dessen Länge als Markierung bzw. der relevanten Zeichenposition für den nächsten Vergleich als Knoten

Beispiel:

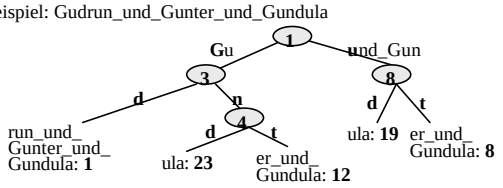


alternative Kantenmarkierungen: L statt LLE, P statt PFEL, usw.

PAT-Bäume

Konkatenation aller Dokumente in einen einzigen Text d[1..n]
und Speicherung aller Suffixe d[1..n], d[2..n], d[3..] usw.
(sog. „sistrings“) in einem (binären) PATRICIA-Baum
oder ggf. nur Suffixe ab Wortgrenzen: d[1..n], d[i_2..n], d[i_3..n] usw.
mit d[i_v-1] = Blank oder Trennzeichen
(ggf. mit limitierter Maximallänge der indexierten Suffixe)

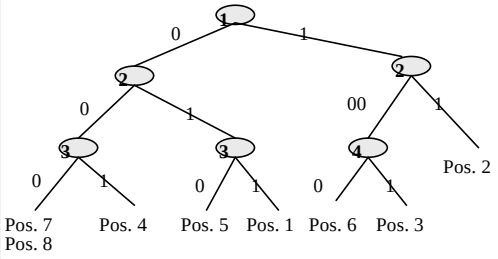
Beispiel: Gudrun_und_Gunter_und_Gundula



Blattknoten speichern die Position des Strings in d,
Innere Knoten speichern zusätzlich die Anzahl ihrer Blätter (=Strings)

Beispiel: Binärer PAT-Baum

Beispieltext: d[1..8] = 01100100[000...]



PAT-Baum-Algorithmen

Binärer PAT-Baum für d[1..n] hat n Blätter und n-1 innere Knoten.
Für zufälliges d hat der Baum die Höhe O(log n).

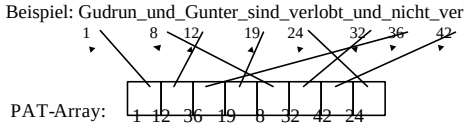
- **Stringsuche** mit p[1..m]:
  Traversiere den mit p[1..m] markierten Pfad im Baum
  bis 1) ≥ m Zeichen oder 2) Blatt erreicht.
  - Bei 1) enthält der erreichte Teilbaum alle Treffer.
  - Bei 2) wird p mit dem Blatt bzw. dessen Position in d verglichen.

  → O(min(m, log n)) Vergleiche
- **Distanzsuche:** Finde Strings p1[1..m1] und p2[1..m2] in d,
  die nicht weiter als k Zeichen auseinander liegen:
  Finde Teilbäume für p1 und p2
  Sortiere die kleinere Trefferliste nach Positionen in d
  Vergleiche die Positionen in der größeren Liste mit der sortierten Liste
- **Bereichssuche:** Finde Strings zwischen p1[1..m1] und p2[1..m2]:
  Finde Teilbäume für p1 und p2.
  Bestimme „kleinsten“ gemeinsamen Vorfahren t.
  Das Resultat sind alle Teilbäume unterhalb von t.
- **Suche nach dem häufigsten String usw.**

PAT-Arrays

Array speichert die Positionen aller Suffixe in d
in lexikographisch aufsteigender Ordnung der Suffixe

Beispiel: Gudrun_und_Gunter_sind_verlobt_und_nicht_verheiratet



ist gegenüber dem PAT-Baum platzeffizienter
und eher für Plattenspeicher geeignet,
hat aber schlechtere Zeiteffizienz

Hashing

Abbildung h: D → [0..m-1], genannt **Hash-Funktion**,
von Schlüsseln x1, ..., xn aus Domain D (z.B. Strings) auf Positionen
h(x1), ..., h(xn) in Array a[0..m-1], genannt **Hash-Tabelle** (mit n < m)
→ Speicherung von Schlüssel xi in a[h(xi)]

Anforderungen an h:
- surjektiv und „so injektiv wie möglich“
- sehr effiziente Berechenbarkeit
- zufällige „Streuung“ (Randomisierung) von x1, ..., xn auf [0..m-1]
  - Urbilder von j1, j2 ∈ [0..m-1] annähernd gleich groß für alle j1, j2 und alle möglichen x1, ..., xn
  - für geordnete Schlüssel x1 < x2 < ... < xn sollte die Ordnung von h(x1), h(x2), ..., h(xn) eine zufällige Permutation sein

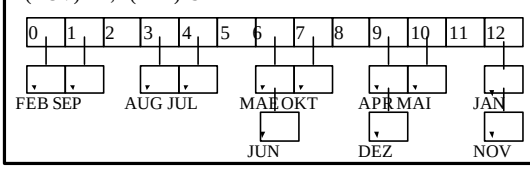
Beispiele für brauchbare Hash-Funktionen
h(x) = (ax + b) mod m für Integers x mit Konstanten a, b
h(x) = (mittlere k Ziffern von x^2) mod m für k-stellige Integers x
h(x) = (ord(c_1)+...+ord(c_k)) mod m für Strings $c_1c_2...c_k \in \Sigma^k$
mit ord: Σ → [1..|Σ|]

Behandlung von Hash-Kollisionen (1)

Hash-Kollisionen: zwei oder mehr Schlüssel x1, x2 mit h(x1)=h(x2)

Überlaufketten (Separate Chaining) pro Zelle der Hash-Tabelle

Beispiel:
Schlüssel JAN, FEB, MAE, APR, usw. (n=12, m=13)
h(xyz) = (ord(x)+ord(y)+ord(z)) mod m
h(JAN) = (10+1+14) mod 13 = 12, h(FEB)=0, h(MAE)=6, h(APR)=9,
h(MAI)=10, h(JUN)=6, h(JUL)=4, h(AUG)=3, h(SEP)=1, h(OKT)=7,
h(NOV)=12, h(DEZ)=9



Behandlung von Hash-Kollisionen (2)

Hash-Kollisionen: zwei oder mehr Schlüssel x1, x2 mit h(x1)=h(x2)

Offene Adressierung (Open Addressing):
bei bereits besetztem h(xi) wird eine Ersatzposition in [0..m-1] gesucht
- **Lineares Sondieren**: finde die erste frei Position > h(xi) (modulo m)
- **Rehashing**: wende (zweite) Hash-Funktion h'(xi) so oft auf h(xi) bzw. h'(xi) an, bis freie Position gefunden

Löschen von a[j]: Markieren der Zelle oder „Rehash" von h'(a[j]) usw.

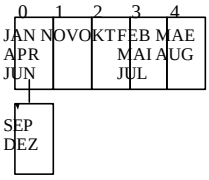
Beispiel für lineares Sondieren:

0	1	2	3	4	5	6	7	8	9	10	11	12
FEB	SEP	NOV	AUG	JUL		MAE	JUN	OKT	APR	MAI	DEZ	JAN

Hash-Files auf Magnetplatten

Organisation der Hash-"Tabelle" als externes File mit
m Buckets, die jeweils einen Block plus ggf. Überlaufblöcke umfassen
+ ggf. dynamische Hash-Verfahren mit dynamischer Erhöhung von m

Beispiel:
Schlüssel JAN, FEB, MAE, APR, usw. (n=12)
m = 5 Buckets, Blockkapazität b = 3
h(xyz) = (ord(x)+ord(y)+ord(z)) mod m
h(JAN) = (10+1+14) mod 5 = 0, h(FEB)=3, h(MAE)=4, h(APR)=0,
h(MAI)=3, h(JUN)=0, h(JUL)=3, h(AUG)=4, h(SEP)=0, h(OKT)=2,
h(NOV)=1, h(DEZ)=0



Signaturen

Repräsentation jedes Dokuments bzw. Blocks eines Dokuments d durch einen Bitvektor fester Länge, die **Signatur** s(d) des Dokuments.

Signaturen werden durch Hashing von Termen in d auf Bits oder Bitpositionen in s(d) erzeugt.
Zu einer konjunktiven Query q={d: d enthält Terme t1, ..., tk} wird analog eine Anfragesignatur s(q). Es gilt:
d ist Treffer für q Þ *s(d)* Ê *s(q) bzgl. der Positionen mit 1-Bits*
Disjunktive Queries $q=q_1 \vee ... \vee q_p$ werden als Menge von Einzelqueries $q_1, ..., q_p$ behandelt.

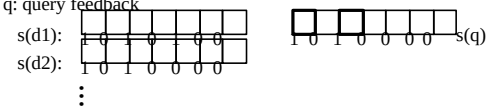
Anfrageauswertung für konjunktive Query q
durch sequentiellen Scan eines Signature-Files:

```
for all documents d do
    if s(d) Ê s(q) then teste d gegen q
                (* Elimination von „False Drops" *) fi
od
```


Signaturgenerierung: Überlagerte Codierung (Superimposed Coding)

Jeder in einem Dokument vorkommende Term eines Termvektorraums wird auf eine (oder mehrere) Positionen eines Bitvektors abgebildet.

Beispiel (für Bitvektor s[1..7]):
d1: *user* julia provided *feedback* on her *queries*
d2: *query* processing is sped up by *index* structures
...
h(feedback)=3, h(index)=3, h(query)=1, h(user)=5, ...
q: query feedback



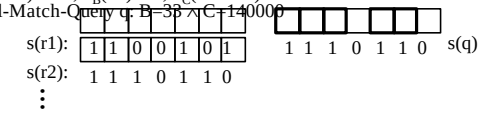
d1 und d2 sind Kandidaten, d1 ist Treffer, d2 ist „False Drop"

unterstützt ggf. auch Substring-Suche durch Abbildung von N-Grammen

Signaturgenerierung: Disjunkte Codierung (Disjoint Coding)

Jeder Attributwert eines mehrdimensionalen Datenraums wird auf eine kurze Bitfolge abgebildet, und die Bitfolgen aller Attribute eines Records werden zu einer Signatur konkateniert.

Beispiel (3-dim. Datenraum mit A: PLZ, B: Alter, C: Einkommen):
r1: 66123 33 120000
r2: 70129 31 142000
...
$h_A(66123)=110$, $h_B(33)=01$, $h_C(120000)=01$
$h_A(70129)=111$, $h_B(31)=01$, $h_C(140000)=10$
Partial-Match-Query q: B=35 ∧ C=140000



geeignet für strukt. Records mit mehreren Feldern (z.B. Mailattribute), wenig sinnvoll für unstrukt. Textdokumente

Organisation von Signatur-Files

Signatur-Files können auf verschiedene Arten in Plattenblöcken organisiert werden:
- in **sequentiellen Files** werden einfach alle Signaturen nach DocIDs geordnet zusammengehängt
- beim **Bit-Slicing** werden alle Signaturen bitpositionsweise gespeichert (wie bei einer spaltenweise gespeicherten Matrix), so daß Queries mit wenigen 1-Bits nur wenige „Bit-Slices" scannen müssen
- in mehreren **File-Partititionen**, die z.B. durch Round-Robin der Blocknummern über mehrere Platten oder Rechner verteilt und parallel verarbeitet werden können
- in **hierarchischen Files**, bei denen für jeden Block von Signaturen eine – durch Disjunktion gebildete - kombinierte Signatur im Vaterknoten eines Baums gespeichert wird, so daß Queries die Blätter aussichtsloser Teilbäume früh eliminieren können

Bestimmung einer geeigneten Signaturlänge

Betrachte Dokumentblöcke der festen Länge k (Terme).
Gesucht ist eine optimale Signaturlänge, die einerseits möglichst kurz sein sollte, andererseits aber möglichst wenig False Drops produziert.

Parameter:
L – Länge der Signatur (in Bits)
g – Termgewicht (Anzahl der 1-Bits pro Term)
k – Anzahl der Terme pro Block
a – Anzahl der Terme in einer Anfrage
t - Anzahl der 1-Bits in einer Anfragesignatur (Anfragegewicht)
F – Anzahl der False Drops einer Anfrage
N - Anzahl der Signaturen (Gesamtzahl der Blöcke aller Dokumente)

Gesucht: optimale Werte für L und g (bei gegebenen k und a)

Zusammenfassung

B*-Bäume:
+ effizient, universell/vielseitig

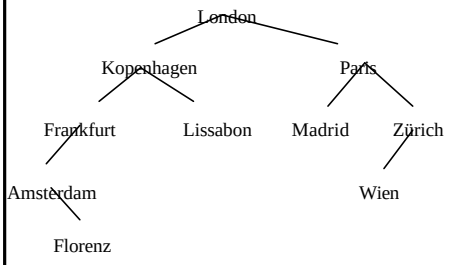
PAT-Bäume:
+ effizient, universell/vielseitig
– nicht für Sekundärspeicher konzipiert

Hashing:
+ effizient
– nur für Exact-Match-Suche
– primär für im voraus bekanntes Datenvolumen

Signaturen:
+ hinreichend effizient
+ leicht parallelisierbar
– sensitiv bzgl. Signaturgenerierung

Beispiel für einen binären Suchbaum

London, Paris, Madrid, Kopenhagen, Lissabon, Zürich, Frankfurt, Wien, Amsterdam, Florenz



Platz- und Kostenberechnung eines B*-Baums (1)

Beispiel 1:

Seitengröße 32 KB, Nettokapazität (ohne Header etc.): 32000 Bytes
Füllgrad 80 Prozent (→ 25600 Bytes pro Seite),
Zeigerlänge 4 Bytes, Record/Doc-ID-Länge 8 Bytes

10 Mio. Records (z.B. Mail) mit indiziertem Feld (Datum) a 20 Bytes
10 000 verschiedene Schlüssel (z.B. 1 Jahr mit Stundengenauigkeit)
→ 1000 Zeiger pro Schlüssel → 8020 Bytes pro Blatteintrag
→ 4 Einträge pro Blatt → 2500 Blätter

→ 24 Bytes pro Wegweiser → Fanout 1065
→ Baumhöhe: ceil ($\log_{1065}$ 2500) +1 = 3
→ 2500 Blätter (Niveau 3) ≈ 100 MBytes,
ceil (2500/1065)=3 innere Knoten (Niveau 2),
ceil (3(1065)=1 Wurzel (Niveau 1)

→Kosten für Bereichssuche über 200 Schlüssel (ca. 1 Woche):
→3 I/Os bis zum ersten Blatt + 200/4 –1 = 49 I/Os für weitere Blätter

Platz- und Kostenberechnung eines B*-Baums (2)

Beispiel 2:

Seitengröße 32 KB, Nettokapazität (ohne Header etc.): 32000 Bytes
Füllgrad 80 Prozent (→ 25600 Bytes pro Seite),
Zeigerlänge 4 Bytes, Record/Doc-ID-Länge 8 Bytes

10 Mio. Docs mit jeweils 1000 Termen a 20 Bytes
aus insgesamt 1 Mio. Termen (= verschiedene Schlüssel)
→ 10 Mio.* 1000 / 1 Mio. = 10000 Zeiger pro Schlüssel
→ 80020 Bytes pro Schlüssel
→ 1 Eintrag pro Blatt (plus je 3 weitere Seiten) → 1 Mio. Blätter

→ 24 Bytes pro Wegweiser → Fanout 1065
→ Baumhöhe: ceil ($\log_{1065}$ 1 000 000) +1 = 3
→ 1 Mio. Blätter (Niveau 3) plus 3 Mio. weitere Seiten ≈ 120 GBytes,
ceil (1 Mio. / 1065) ≈ 1000 innere Knoten (Niveau 2)
ceil (1000/1065)=1 Wurzel (Niveau 1)

→Kosten für Suche nach 1 Term:
→3 I/Os bis zum Blatt + ggf. 3 I/Os für weitere Seiten der ID-Liste

Effizienz von Hashing (1)

Average-Case-Komplexität für Suche nach einem Schlüssel: O(1)
(zumindest für n << m, genauer: abhängig von Auslastung α := n/m)
aber: Worst-Case-Komplexität: O(n)
keine Unterstützung von Bereichssuchen

Analyse von idealisiertem Hashing mit offener Adressierung
(Gleichverteilung der belegten Zellen in [0..m-1]):
Z = Anzahl getesteter Zellen zum Einfügen bzw. erfolglosen Suchen
Y = Anzahl getesteter Zellen beim erfolgreichen Suchen

$$E[Z] = \sum_{r=1}^{m} r \binom{m-r}{n-r+1} \Big/ \binom{m}{n} = \frac{m+1}{m-n+1} = \frac{1+1/m}{1-a+1/m} \approx \frac{1}{1-a}$$

$$E[Y] = \frac{1}{n}\sum_{k=1}^{n} \frac{m+1}{m-(k-1)+1} = \frac{m+1}{n}\left(H_{m+1} - H_{m-n+1}\right) \approx \frac{m+1}{n} \ln \frac{m+1}{m-n+1} \approx \frac{1}{a} \ln \frac{1}{1-a}$$

Effizienz von Hashing (2)

Analyse von Bucket-Hashing mit Blockkapazität b:

$$P[\textit{Bucket hat k Schlüssel}] = \binom{n}{k}\left(\frac{1}{m}\right)^{k}\left(\frac{m-1}{m}\right)^{n-k}$$

$$P[\textit{Bucket hat} \le k \textit{ Schlüssel}] = \sum_{i=0}^{k}\binom{n}{i}\left(\frac{1}{m}\right)^{i}\left(\frac{m-1}{m}\right)^{n-i} =: p_k$$

$$P[\textit{Bucket hat} \le k \textit{ Blöcke}] = \sum_{i=0}^{kb}\binom{n}{i}\left(\frac{1}{m}\right)^{i}\left(\frac{m-1}{m}\right)^{n-i} =: P_k$$

$$P[\textit{Bucket hat k Blöcke}] = P_k - P_{k-1}$$

$$E[Z] = \sum_{k=1}^{ceil(n/b)} k(P_k - P_{k-1}) \qquad \approx a + e^{-a} \quad \text{für b=1}$$

$$E[Y] = \sum_{k=1}^{ceil(n/b)} (P_k - P_{k-1}) \sum_{i=1}^{k} i\frac{1}{k} \qquad \approx 1 + \frac{a}{2} \quad \text{für b=1}$$