Indexstrukturen

Dateistrukturen und typische Aktionen

Wir betrachten einige typische Aktionen und vergleichen die Kosten für die Durchführung je nach Organisation der Daten. Wir untersuchen die drei Dateistrukturen aus dem vorherigen Kapitel:

1. Die *Haufendatei* (*Heap File*): Dateien enthalten Datensätze in zufälliger Reihenfolge.
2. Die *Sortierte Datei*: Dateien enthalten Datensätze, die nach einem (oder mehreren) Felder(n) der Datensätze sortiert sind.
3. Die *Hashdatei*: Dateien enthalten Datensätze in Seiten, die Zuordnung ergibt sich durch die Behälternummer ermittelt aus einer Hashfunktion.

Typische Aktionen Wir wollen folgende 5 typische Aktionen untersuchen, wie sie bei den jeweiligen Dateistrukturen durchzuführen sind:

1. Scan, d.h. sequenzielles Lesen aller Datensätze der Datei
   `select * from R`
2. Punktsuche
   `select * from R where A = 42`
3. Bereichssuche
   `select * from R where A between 0 and 100`
4. Einfügen eines Datensatzes
   `insert into R values ...`
5. Löschen eines Datensatzes identifiziert durch seine Satz-Id (*record id* `rid`)

Kostenmodell Wir unterstellen die Parameter aus Tabelle 1. Die Blockgröße ist gleich der Seitengröße.

Kosten des Scans einer Datei

1. Haufendatei
   Man muss alle b Blöcke lesen und jeden der r Datensätze verarbeiten, also:
   $$\mathcal{C} = b \cdot (D + r \cdot C)$$

Tabelle 1: Parameter für einfaches Kostenmodell

Parameter	**Beschreibung**
b	Zahl der Blöcke in der Datei
r	Zahl der Datensätze in einem Block
D	Zeit, einen Block zu lesen oder zu schreiben, etwa 10 ms
C	CPU-Zeit für die Verarbeitung eines Datensatzes, etwa 0,1 μs (Mikrosekunden = $10^{-6}s$)
H	CPU-Zeit für die Berechnung der Hashfunktion für einen Datensatz, etwa 0,1 μs
$\mathcal{C}$	Gesamtkosten

2. Sortierte Datei
   Die Sortierung hilft hier nichts, also muss man dasselbe tun wie bei der Haufendatei:

$$\mathcal{C} = b \cdot (D + r \cdot C)$$

3. Hashdatei:
   Auch die Hashfunktion hilft wenig. Dadurch, dass wir in den Blöcken Platz gelassen haben, müssen wir mehr Blöcke lesen:

$$\mathcal{C} = (100/80) \cdot b \cdot (D + r \cdot C)$$

Kosten der Punktsuche

1. Haufendatei
   Angenommen, es ist bekannt, dass die Werte des betroffenen Feldes eindeutig sind, dann ist man im Durchschnitt fertig, wenn man Hälfte der Datensätze gelesen hat, also:

$$\mathcal{C} = 1/2 \cdot b \cdot (D + r \cdot C)$$

   Wenn es sich um ein anderes Feld handelt, dann muss man stets die komplette Datei durchlaufen, also:

$$\mathcal{C} = b \cdot (D + r \cdot C)$$

2. Sortierte Datei
   Wenn die Datei nach dem Feld sortiert ist, für das wir einen bestimmten Wert suchen, dann kann man binäre Suche machen, und erhält:

$$\mathcal{C} = log_2 b \cdot (D + \log_2 r \cdot C)$$

Hat das Feld keine eindeutigen Werte, dann folgen die gesuchten Datensätze unmittelbar auf den ersten gefundenen Datensatz.

(Bemerkung: DBMS verwenden tatsächlich nicht binäre Suche für die Punktsuche, sondern Indexstrukturen, die wir weiter unten untersuchen.)

Die Sortierung hilft natürlich nur, wenn wir auch einen Wert des Sortierfeldes suchen. Andernfalls sind wir wieder auf das Vorgehen wie in der Haufendatei verwiesen.

3. Hashdatei:
   Wenn sich die Hashfunktion auf das gesuchte Feld bezieht, führt sie uns *direkt* zu dem gesuchten Block (oder der Überlaufkette des Blocks). Wenn das Feld eindeutige Werte hat, ergibt sich also:

$$\mathcal{C} = H + D + 1/2 \cdot r \cdot C$$

   Sind die Werte im Feld nicht eindeutig, hat man

$$\mathcal{C} = H + D + r \cdot C$$

   (Der Einfachheit halber ist unterstellt, dass es keine Überlaufkette gibt.)

   Die Hashfunktion hilft nur, wenn wir nach einem Wert des Feldes suchen, auf das die Hashfunktion angewendet wird.

Kosten der Bereichssuche

1. Haufendatei:
   Die gesuchten Datensätze können irgendwo in der Haufendatei stehen, also

$$\mathcal{C} = b \cdot (D + r \cdot C)$$

2. Sortierte Datei (sortiert nach A):
   Man sucht zunächst die untere Grenze per binärer Suche und liest dann sequenziell bis die obere Grenze überschritten wird ($n+1$ sei die Zahl der Datensätze im Bereich):

$$\mathcal{C} = log_2 b \cdot (D + log_2 r \cdot C) + \lceil n/r \rceil \cdot D + n \cdot C$$

3. Hashdatei:
   Im Unterschied zur Punktsuche hilft die Hashfunktion nichts, da die Bilder aufeinanderfolgender Werte unter der Hashfunktion verstreut sind, also wie beim Scan der Datei:

$$\mathcal{C} = (100/80) \cdot b \cdot (D + r \cdot C)$$

Kosten des Einfügens

1. Haufendatei:
   Wir können den Datensatz in einem beliebigen Block einfügen, der noch nicht voll ist, etwa am Ende. Ohne die Zeit für das Finden dieses Blocks ergibt sich:

$$\mathcal{C} = 2 \cdot D + C$$

2. Sortierte Datei:
   Wir müssen den Datensatz entsprechend der Sortierung einfügen, also im Durchschnitt in der Mitte. Dazu müssen wir die Häfte der Datensätze nach hinten verschieben, also:

$$\mathcal{C} = log_2 b \cdot (D + log_2 r \cdot C) + 1/2 \cdot b \cdot \underbrace{(D + r \cdot C + D)}_{lesen+shiften+schreiben}$$

3. Hashdatei:
   Wir berechnen die Hashfunktion, lesen den entsprechenden Block und schreiben ihn (wir nehmen an, dass wir noch freien Platz haben):

$$\mathcal{C} = H + D + C + D$$

Kosten des Löschens eines Datensatzes identifiziert durch rid

1. Haufendatei:
   Wir setzen voraus, dass wir die Datei nach dem Löschen nicht verkleinern müssen, weil unsere Haufendatei die freien Slots verwaltet, z.B. durch eine Löschkette o.ä. Also müssen wir nur den Block direkt lesen, im Speicher als gelöscht markieren, in die Löschkette eintragen und wieder schreiben:

$$\mathcal{C} = D + C + D$$

2. Sortierte Datei:
   Wir lesen den Datensatz und schieben dann (im Durchschnitt) die andere Hälfte der Datensätze vor:

$$\mathcal{C} = D + 1/2 \cdot b \cdot (D + r \cdot C + D)$$

3. Hashdatei:
   Da wir die rid kennen, brauchen wir nicht mal die Hashfunktion berechnen und im Block suchen, also wie bei der Haufendatei:

$$\mathcal{C} = D + C + D$$

Fazit Es gibt keine Strukturierung der Datei, die für alle betrachteten typischen Aktionen gleichermaßen geeignet wäre.

Es gibt jedoch Index-Strukturen, die die Vorteile sortierter Dateien haben und gleichwohl Einfügen und Löschen effizient unterstützen: B^+-*Bäume*.

Grundlegendes zu Indexen

Ehe wir B^+-Bäume untersuchen einige allgemeine Bemerkungen zu Indexen.

Grundidee Ein Index ist eine Datenstruktur, die die Suche auf einem bestimmten Attribut der Relation (oder einer Kombination) beschleunigt.

Man geht im Prinzip so vor:

1. Durchsuche den Index nach dem gesuchten Wert, den nennt man auch den Suchschlüssel. Der Index muss nun die tatsächliche Fundstelle der gesuchten Daten enthalten.
2. Greife nun *direkt* auf die gesuchten Daten zu.

Man kann Indexe auf verschiedene Weise organisieren:

1. Der Index selbst enthält die Nutzdaten. D.h. die Daten selbst sind nach dem Attribut des Indexes sortiert. Es kann sinnvoll sein, einen Index für eine Tabelle so zu organisieren; entspricht dann einer sortierten Datei für die eigentlichen Daten.
2. Der Index enthält Paare von (Suchschlüssel, Satz-Id rid).
3. Der Index enthält Paare von (Suchschlüssel, Liste von rids). Dies kann sinnvoll sein, wenn die Werte des Indexes nicht eindeutig in den Daten sind, allerdings bekommt man dann Indexeinträge variabler Länge.

Clusterindex Wenn eine Datei einen Index hat, der (im Prinzip) darin besteht, dass die Datei nach den Werten des Index sortiert ist, dann spricht man von einem *Clusterindex*.

DBMS haben normalerweise Anweisungen, mit denen für eine Tabelle ein Clusterindex angelegt werden kann. Selbstverständlich kann es für eine Tabelle nur *einen* Clusterindex geben.

Beispiele:

- PostgreSQL hat einen Befehl `cluster <table> using <index>`. Dadurch wird die Tabelle `<table>` gemäß des Index `<index>` sortiert. Allerdings berücksichtigen Änderungen an der Tabelle die Sortierung *nicht*! Dieser Befehl ist nicht SQL-Standard.
- Bei Oracle kann man beim `create table` mit dem Schlüsselwort `organization` angeben, wie die Tabelle physisch organisiert werden soll: `organization heap` bestimmt als Dateityp eine Haufendatei und `organization index` bestimmt die Datei als einen nach dem Primärschlüssel organisierten B^*-Baum.

  Darüberhinaus hat Oracle noch das Konzept eines `Cluster`s, eine Datei, die Daten aus mehreren Tabellen enthalten kann, z.B. eine Kombination von Primär- und Fremdschlüsseln in einer Beziehung – dieses Konzept dient der Beschleunigung von Joins bei sogenannten *Master-Detail*-Beziehungen.

Speicherbedarf von Indexen Ein Index, der nur einen Indexeintrag *pro Block* enthält, heißt *dünn besetzt*. Der Indexeintrag enthält den kleinsten Wert zum Index im Block.

Ein Index, der einen Indexeintrag *pro Datensatz* enthält, heißt *dicht besetzt*.

Offensichtlich ist das Finden von Datensätzen bei einem dünn besetzten Index etwas aufwändiger, weil man innerhalb des Blocks suchen muss; andererseits benötigt ein dünn besetzter Index weniger Speicherplatz als ein dichter Index.

Man beachte aber: Bei einem dünn besetzten Index *müssen* die eigentlichen Daten in den Seiten sortiert sein (Man sagt: ein *geclusterter* – was ein Wort! – Index), denn wir müssen ja die auf den kleinsten Wert folgenden Datensätze in *dem* Block finden, auf den der Wert im Indexeintrag zeigt. Als Korollar folgt, dass es höchstens einen dünn besetzten Index pro Datei geben kann.

B^+-Bäume

Aufbau von B^+-Bäumen

Wir unterscheiden, was den Aufbau eines B^+-Baums angeht, zwischen den *Blättern* und den *inneren Knoten*.

Die Blätter bilden in der Regel eine doppelt verkettete Liste, die sogenannte Sequenzmenge. Jedes Blatt enthält zu den Suchschlüsseln die eigentlichen Datensätze oder Referenzen auf die Datensätze, also rids. Da zweiteres die Regel ist, gehen wir im Folgenden stets davon aus. Ein

Paar (k, rid) bestehend aus einem Wert k zum Attribut, nach dem indiziert wird und einer Referenz auf einen Datensatz rid bezeichnet man als Indexeintrag und schreibt ihn als k^*. (Wir gehen im Folgenden davon aus, dass der Index nur eindeutige Werte hat.)

Innere Knoten enthalten abwechselnd Suchschlüssel und Referenzen auf Subbäume des B$^+$-Baums. Zu jedem Suchschlüssel zeigt die Referenz *vor* dem Suchschlüssel k auf die Subbäume mit den Suchschlüsseln $< k$ und die Referenz *nach* k auf die Subbäume mit den Suchschlüsseln $\geq k$.

Abb. 1 zeigt einen B$^+$-Baum der Ordnung 2.

Eigenschaften Ein B$^+$-Baum der Ordnung m hat folgende Eigenschaften:

- Ein Blatt hat höchstens $2m$ Indexeinträge

  $$k_1^* \quad k_2^* \quad k_3^* \quad \dots \quad k_n^*$$

  mit

  $$k_1 < k_2 < k_3 < \cdots < k_n$$

- Die inneren Knoten bestehen aus Suchschlüsseln k_i und Referenzen auf Subbäume p_i:

  $$p_0 \quad k_1 \quad p_1 \quad k_2 \quad p_2 \quad \dots \quad k_n \quad p_n$$

  mit

  $$k_1 < k_2 < k_3 < \cdots < k_n$$

- Jeder Knoten (mit Ausnahme der Wurzel) besitzt zwischen m und $2m$ Suchschlüssel bzw. Indexeinträge.
- Alle Pfade von der Wurzel zu einem Blatt sind gleich lang, man sagt: der Baum ist balanciert.
- Jeder innere Knoten mit n Suchschlüsseln hat genau $n+1$ Kinder.
- Benachbarte Blattknoten sind doppelt verkettet.

Suchen in B$^+$-Bäumen

Die Idee ist offensichtlich: Auf der Suche nach der rid zu einem Schlüsselwert k startet man an der Wurzel und nimmt jeweils den richtigen Pointer bis zum Blatt. Innerhalb des Blattknotens sind die Indexeinträge sortiert, man findet den gesuchten Indexeintrag als z.B. per binärer Suche.

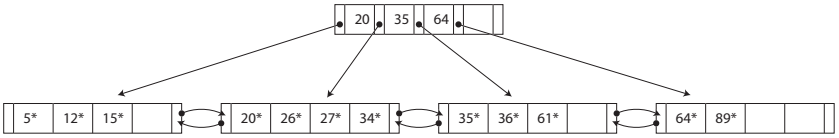

Abbildung 1: Beispiel eines B^+-Baums der Ordnung 2

Algorithmus zur Suche

Suche im Blatt:

```
rid leaf_search(leaf, k) {
  bin_search(leaf, k);
  if (found) return rid of k*;
  else return null;
}
```

Suche in Subbaum:

```
node tree_search(node, k) {
  if (node is a leaf) return node;
  switch (k) {
    case k < k_1:
      return tree_search(p_0, k);
    case k_i <= k < k_{i+1}:
      return tree_search(p_i, k);
    case k_n <= k:
    // n largest filled slot in node
      return tree_search(p_n, k);
  }
}
```

Suche im B^+-Baum:

```
rid search(k) {
  leaf = tree_search(root, k);
  return leaf_search(leaf, k);
}
```

Manipulation von B^+-Bäumen

B^+-Bäume sind balanciert, d.h. bei Änderungen am Baum durch Einfügen neuer Indexeinträge oder durch Löschen von Indexeinträgen muss diese Eigenschaft (samt all der anderen Eigenschaften des B^+-Baums) erhalten bleiben.

Einfügen

Die Idee:
Im ersten Schritt sucht man den Blattknoten, in den der neue Indexeintrag k^* eingefügt werden muss. Dazu können wir `tree_search` verwenden.
Im zweiten Schritt fügt man den Indexeintrag ein, sofern noch genügend Platz für einen Indexeintrag frei ist. Andernfalls muss man das Blatt in zwei Blätter teilen (*split*) und das neue Blatt im Vater verankern. Dies kann natürlich auch zu einem Teilen des Vaters zwingen usw. bis eventuell sogar die Wurzel geteilt werden muss.

Algorithmus zum Einfügen Einfügen in ein Blatt:
`leaf_insert` fügt einen neuen Indexeintrag in ein Blatt ein. Die Funktion gibt ein Paar aus einem Suchschlüssel k^+ und einer Referenz auf ein Blatt zurück. Ist das Paar `(null, null)`, dann musste kein neues Blatt erzeugt werden. Andernfalls ist der Suchschlüssel k^+ der kleinste Wert im neuen Blatt und p^+ die Referenz auf das neue Blatt.

```
(k+,p+) leaf_insert(leaf, k*) {
  if (entry fits in leaf) {
    insert k* in leaf; //ordered
    return (null, null);
  } else {
    allocate new leaf; //referenced by p+
    move last m entries from node to new leaf
    if (k < k_{m+1} insert (k, p) in node
    else insert (k, p ) in new leaf
    return(k_{m+1},p+)
  }
```

Einfügen in einen inneren Knotens:
`node_insert` fügt einen Wert k und eine Referenz p auf einen Kindknoten in einen inneren Knoten ein. Wenn eine Aufteilung nötig ist, gibt die Funktion ein Paar mit dem kleinsten Wert k^+ und einer Referenz p^+ auf den neuen Knoten zurück.

```
(k+,p+) node_insert(node, k, p) {
  if (entry fits in node) {
    insert (k, p) in node; //ordered
    return (null, null);
  } else {
    allocate new leaf; //referenced by p+
    move last m entries from node to new leaf
    if (k < k_{m+1} insert (k, p) in node
    else insert (k, p ) in new leaf
    return(k_{m+1},p+)
  }
```

Einfügen in einen Subbaum:

```
(k+, p+) tree_insert(node, k*) {
  if (node is a leaf) return leaf_insert(node, k*);
  else {
    switch (k) {
      case k < k_1:
        (k+, p+) = tree_insert(p_0, k*);
      case k_i <= k < k_{i+1}:
        (k+, p+) = tree_insert(p_i, k*);
      case k_n <= k:
      // n largest filled slot in node
        (k+, p*) = tree_insert(p_n, k*);
    }
  if (k+ == null) return (null, null);
  else return node_insert(node, k+, p+);
  }
}
```

Einfügen in den B^+-Baum:

```
insert(k*) {
  (k+, p+) = tree_insert(root, k*);
  if (k+ != null) {
    allocate new root node r;
    set in r:
      p_0 = root; //old root
      k_1 = k+;
      p_1 = p+;
    root = r;
  }
}
```

Bemerkung: Der hier beschriebene Algorithmus geht beim Teilen von Knoten so vor, dass zunächst geteilt wird, und dann der neue Eintrag in den passenden Knoten eingefügt wird. Eine andere Variante: man erzeugt zunächst einen Knoten mit $2m + 1$ Einträgen, in den der neue Eintrag eingeordnet ist und teilt dann an der Position k_{m+1}.

Optimierung des Algorithmus Angenommen wir haben in einem B^+-Baum der Ordnung 2 in zwei benachbarten Blattknoten folgende Situation:

$$[2^* \quad 3^* \quad 5^* \quad 7^*] \quad [11^* \quad 12^* \quad frei \quad frei]$$

Wenn man jetzt 6^* einfügen möchte, würde unser Algorithmus zu einer Teilung des ersten Blattknotens führen. Man könnte aber auch 7^* in das zweite Blatt verschieben, weil dort ja noch Platz ist. Man muss dann natürlich den Vater des zweiten Blatts ändern, weil ja jetzt 7 der kleinste Suchschlüssel im Blatt ist.

Durch diese Optimierung kann man Teilung vermeiden. Implementierungen verwenden diese Technik oft nicht bei allen Knoten des B^+-

Baums, sondern nur bei den Blättern. Dies liegt nahe, denn die Blätter sind ja doppelt verkettet, d.h. die Geschwister sind leicht zu erreichen.

Löschen von Indexeinträgen Wieder ist klar, dass man den Indexeintrag k^*, den man löschen möchte, sucht. Hat das Blatt mit dem Indexeintrag mehr als m Einträge, dann löscht man den Eintrag einfach. Sind es jedoch genau m Einträge, dann muss man den B^+-Baum umbauen.

Welche Möglichkeiten hat man, wenn durch das Löschen ein Blatt zu wenig Einträge haben würde?

- Verteilung auf die Blätter ändern
  Hat ein Geschwister des Blattes > m Einträge, dann kann man die Einträge auf die Blätter so verteilen, dass alle genügend Einträge haben.
- Verschmelzen von Blättern
  Wenn man nicht mehr andere Einträge von Geschwistern zum Auffüllen hat, muss man zwei Blätter verschmelzen und den Vater entsprechend anpassen.
- Verschmelzen von inneren Knoten
  Durch das Verschmelzen von Blättern kann es auch vorkommen, dass innere Knoten zu wenige Einträge bekommen würden, man muss also ggf. auch sie verschmelzen.
- Verteilung auf inneren Knoten ändern
  Ein Option kann es auch sein, die Verteilung der Einträge zwischen den Geschwistern innerer Knoten zu ändern, um das Verschmelzen zu vermeiden.
- Löschen der Wurzel
  Das Verschmelzen innerer Knoten kann dazu führen, dass der letzte Eintrag in der Wurzel gelöscht wird. Der zuletzt durch Verschmelzen entstandene Knoten wird dann die neue Wurzel.

Implementierungen von B^+-Bäumen in DBMS vermeiden oft den Aufwand beim Löschen von Einträgen im Baum, indem sie auch Knoten mit weniger als m Einträgen verwenden.

Duplikate in Schlüsseln Duplikate in Schlüsseln komplizieren die Sache etwas. Was kann man tun?

- Das DBMS bildet intern einen eindeutigen Schlüssel, in dem es zu k eine Tupel-Id hinzufügt. Tupel-Ids sind interne eindeutige Identifizierer für einen Datensatz in einer Tabelle. (So macht es DB2.)

- In den Indexeinträgen k^* wird nicht nur eine *rid* gespeichert, sondern eine Liste von *rid*s. Dadurch können die Blätter variable Länge bekommen. (Informix macht es so.)
- Man berücksichtigt Duplikate in allen Algorithmen des B^+-Baums.

Hashbasierte Indexe

Konzept

Ein hashbasierter Index verwendet eine Hashfunktion auf dem Indexattribut, um das Tupel zu indizieren. Ist das Attribut A, dann ist eine Hashfunktion h eine Funktion $h : V \to [0, 1, \ldots n-1]$ für eine natürliche Zahl n und den Wertebereich V des Attributs.

Der Index besteht dann aus einem Array von n Blöcken. Der i-te Block nimmt die Indexeinträge k^* der Tupel auf, für die $h(k) = i$ gilt. Man spricht auch vom *Bucket*-Array. Ein Behälter (*bucket*) selbst enthält die Indexeinträge in beliebiger Reihenfolge. Es kann auch sein, dass für die Behälter eine fixe Zahl von Blöcken > 1 verwendet wird.

Ist die Zahl n fest gegeben, spricht man von einem *statischen* hashbasierten Index. Da die Zahl n in der Regel kleiner ist als Zahl der Indexeinträge, kann es vorkommen, dass die vorgesehene Zahl von Blöcken für einen Behälter zu klein ist, um alle Indexeinträge aufzunehmen. Dann werden Überlaufblöcke verwendet.

Eigenschaften

Hashbasierte Indexe sind „unschlagbar“ bei der Punktsuche. Andererseits eignen sie sich nicht für die Bereichssuche.

Statische hashbasierte Indexe haben große Nachteile, wenn die Datenmenge stark wächst oder variiert: man braucht viele Überlaufblöcke oder man benötigt viel leeren Platz in der Hashtabelle. Besser geeignet sind dafür dynamische Hashverfahren wie erweiterbares Hashing und lineares Hashing.

Wahl der Hashfunktion

Modulo-Funktion Ist das Attribut ganzzahlig, so sei k sein Wert. Hat es einen anderen Typ, nimmt man einfach die Binärdarstellung des Werts als Zahl k. Die Modulo-Funktion ist dann

$$h(k) = k \ mod \ n$$

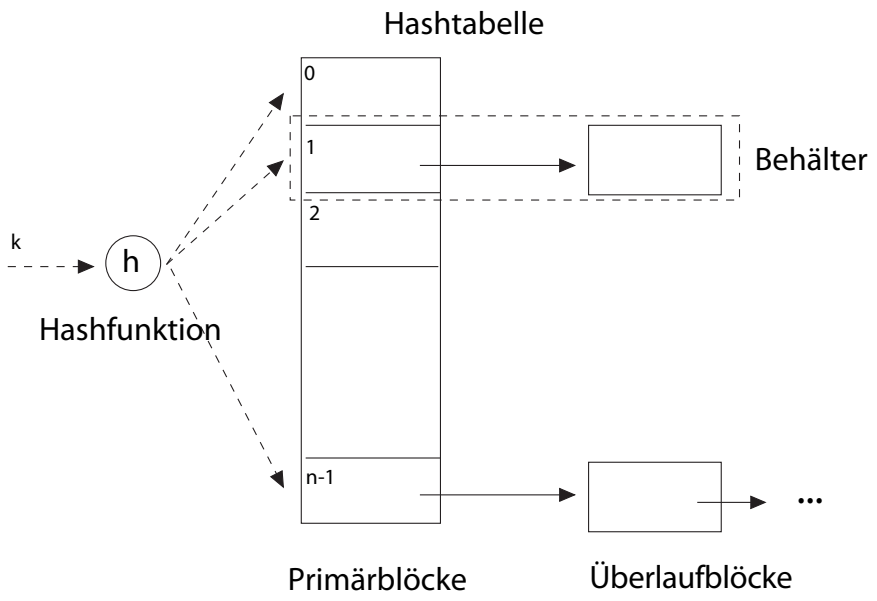

Abbildung 2: Aufbau eines hashbasierten Index

Am besten eignen sich Primzahlen als Modul n, um eine Gleichverteilung zu erreichen.

Multiplikationsverfahren Man wählt eine Zahl Z und berechnet dann

$$h(k) = \lfloor n \cdot (Z \cdot k - \lfloor Z \cdot k \rfloor) \rfloor$$

Eine gute Wahl für Z ist $Z = (\sqrt{5} - 1)/2$ (siehe D.E. Knuth: *The Art of Computer Programming Vol. 3 Sorting and Searching*, S. 517)

Beispiel: $Z = 0.7 und n = 20$
$h(0) = 0$
$h(1) = \lfloor 20 \cdot (0.7 - 0) \rfloor = \lfloor 20 \cdot 0.7 \rfloor = 14$
$h(2) = \lfloor 20 \cdot (0.7 \cdot 2 - 1) \rfloor = \lfloor 20 \cdot 0.4 \rfloor = 8$

Erweiterbares Hashing

Die Grundidee besteht darin, dass man ein Hashverzeichnis hat, das seine Größe je nach Zahl der verwendeten Behälter ändert. Sobald ein Behälter seine Blöcke ausfüllt und ein neuer Indexeintrag eingetragen werden muss, wird der Behälter aufgeteilt und das Hashverzeichnis entsprechend erweitert.

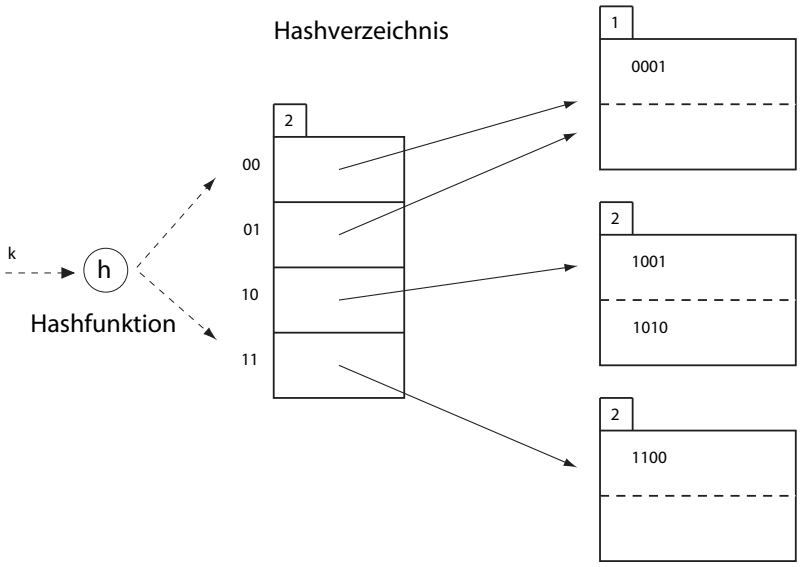

Abbildung 3: Erweiterbarer hashbasierter Index

Abb. 3 zeigt ein einfaches Beispiel eines erweiterbaren hashbasierten Index[1]. In diesem Beispiel gehen wir von folgender Situation aus:

Die Blöcke fassen 2 Indexeinträge. In den Abbildungen wird nicht der Indexeintrag gezeigt, sondern der Hashwert seines Suchschlüssels.

Die Hashfunktion produziert für jeden Schlüssel im Index eine Folge von $k = 4$ Bits. Das Hashverzeichnis verwendet je nach Zahl der zu indizierenden Einträge $i = 2$ dieser k Bits. Die Zahl i wird im Header des Hashverzeichnisses angegeben, und sie bestimmt natürlich auch die Größe des Hashverzeichnisses.

Jeder Eintrag im Hashverzeichnis verweist auf ein Behälter in der eigentlichen Hashtabelle. Sie enthält dann die Indexeinträge. In der Abbildung werden jedoch nicht die Indexeinträge gezeigt, sondern der Hashwert des Indexwerts. Jeder Behälter in der Hashtabelle hat eine Angabe, die angibt, wieviele Bits für die Zuordnung in diesem Behälter berücksichtigt werden müssen.

In Abb. 3 haben wir also 4 Schlüsseleinträge mit den Hashwerten 0001, 1001, 1010, 1100.

[1] Quelle: Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom *Database Systems The Complete Book* S. 654

Einfügen von Einträgen in den Index Stellen wir uns vor, dass in obigem Beispiel die Indexeinträge mit den Hashwerten 0000 und 0111 eingefügt werden sollen. Da wir nur Platz für zwei Indexeinträge in einem Behälter haben, müssen wir den ersten Behälter aufteilen. Es ergibt sich dann die Situation wie in Abb. 4.

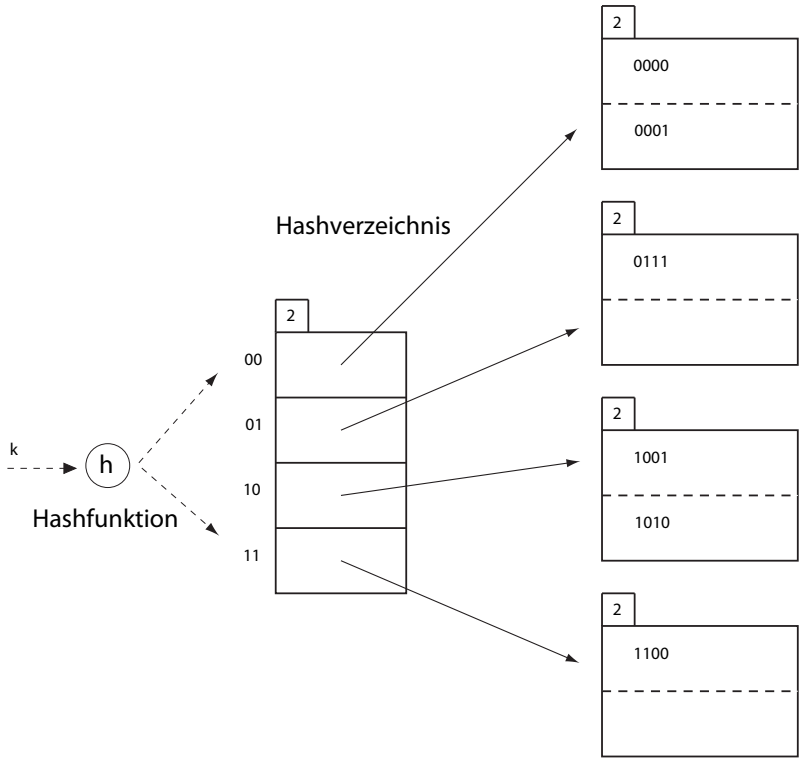

Abbildung 4: Einfügen in den erweiterten Index

Wenn nun auch noch der Indexeintrag mit dem Hashwert 1000 eingefügt werden soll, dann ist nicht genug Platz im Behälter für 10. In diesem Fall muss man nun das Hashverzeichnis erweitern, man verwendet nicht nur 2, sondern 3 Bits für das Hashverzeichnis. Im Ergebnis hat man die Situation in Abb. 5.

Eigenschaften Der wichtigste Vorteil des erweiterbaren hashbasierten Index besteht darin, dass man bei der Punktsuche aus dem Hashverzeichnis direkt zu dem Behälter gelangt, in dem der gesuchte Datensatz sich befindet (wenn es ihn gibt).

Der Nachteil besteht darin, dass (wie man im obigen Beispiel sieht)

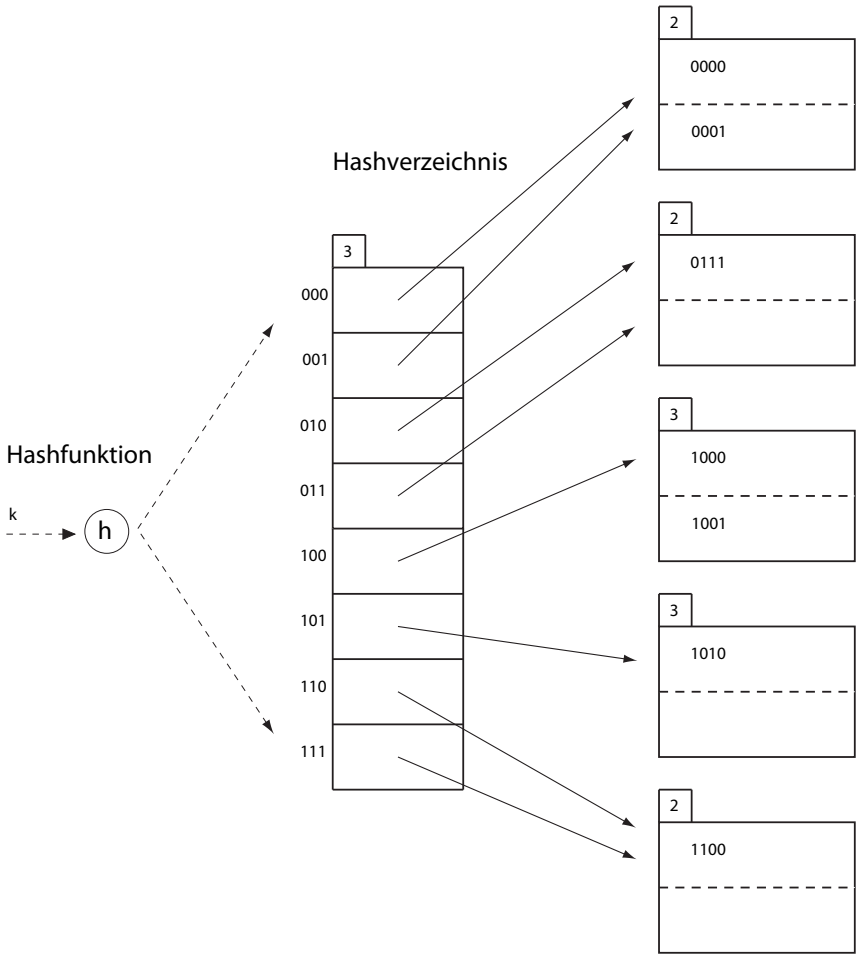

Abbildung 5: Einfügen mit Erweiterung des Hashverzeichnisses

die Erweiterung des Hashverzeichnisses in Schritten von 2-er-Potenzen passiert. Lineares Hashing ist eine Technik, die langsamere Erweiterung ermöglicht:

Lineares Hashing

Beim linearen Hashing wird kein Hashverzeichnis verwendet, sondern man hängt immer neue Behälter an die Hashdatei an. Damit dies funktioniert, verwendet man eine Familie von Hashfunktionen $h_0, h_1, h_2, \ldots$ mit folgenden Eigenschaften:

1. Jede Funktion h_{i+1} hat einen doppelt so großen Wertebereich wie die Funktion h_i.
2. Die durch die Hashfunktion h_i gegebene Partitionierung der Werte in Behälter wird durch h_{i+1} verfeinert: die Behälter bezüglich h_i zerfallen in Paare von Behältern unter h_{i+1}.

Ein Beispiel für eine solche Familie von Hashfunktionen ist:

$$h_i(k) = h(k) \cdot mod(2^i \cdot n) \quad (i = 0, 1, 2, \dots)$$

mit einer gegebenen initialen Hashfunktion $h(k)$ und einer initialen Größe der Hashtabelle n.

Häufig nimmt man n als eine Zweierpotenz, so dass jeweils eine bestimmte Anzahl von Bits des Hashwerts entscheidend ist.

Zunächst wird die Hashfunktion h_0 verwendet. Dies tut man solange, bis der Platz in den Behältern ausgeht. Zunächst nimmt man dann einen Überlaufblock. Eine Aufteilen eines Behälters geschieht nun nach einer festgelegten Strategie. Etwa: Einfügen füllt einen Behälter zu x% oder der Überlaufbereich eines Behälters erreicht eine gewisse Größe. In diesem Falle teilt man zunächst den ersten Behälter und wendet auf ihn die Hashfunktion h_1 an.

Dieser Rehash verteilt die Werte des Behälters auf den aktuellen Behälter und einen neuen Behälter. Dieser „Bruder“ dieses geteilten Behälters wird hinten angehängt. Man merkt sich die Position des als nächstes zu teilenden Behälters.

D.h. man hat immer zuerst Behälter, für die man h_{i+1} anwenden muss *vor* der Position der nächsten Aufteilung. Dann kommen die Behälter für die man h_i verwenden muss und schließlich die Brüder der zuvorderst aufgeteilten Behälter mit h_{i+1}.

Burkhardt Renz
TH Mittelhessen
Fachbereich MNI
Wiesenstr. 14
D-35390 Gießen

Rev 4.0 – 11. März 2019