Lösungsvorschläge zur Klausur „1664 Implementierungskonzepte für Datenbanksysteme“

03. März 2012

Aufgabe 1 (Datenbanksysteme)

(a)

Im Kurstext werden in Abschnitt 1.2 folgende Anforderungen genannt:

- Datenunabhängigkeit
- Effizienter Datenzugriff
- Gemeinsame Datenbasis
- Nebenläufiger Datenzugriff
- Fehlende oder kontrollierte Redundanz
- Konsistenz der Daten
- Integrität der Daten
- Datensicherheit
- Bereitstellung von Backup- und Recovery-Verfahren
- Stellen von Anfragen
- Bereitstellung verschiedenartiger Benutzerschnittstellen
- Flexibilität
- Schnellere Entwicklung von Anwendungen

Hinweis für Korrektoren: Ab 10 Richtigen gibt es die volle Punktzahl.

(b)

Die Zugriffszeit setzt sich zusammen aus:

- der Suchzeit, die für die Positionierung des Schreib-/Lesekopfs auf der richtigen Spur benötigt wird
- der Latenzzeit, die für die Rotation des Schreib-/Lesekopfs auf den gewünschten Block gebraucht wird
- der Übertragungszeit, die tatsächlich für das eigentliche Schreiben oder Lesen der Daten nach der Positionierung des Schreib-/Lesekopfs benötigt wird

(c)

Unterschieden werden Haufendateien, sequentielle Dateien und Hash-Dateien.

In Haufendateien werden die Daten ungeordnet in chronologischer Folge jeweils ans Ende der Datei angehängt. Sie sind sehr speicherplatzeffizient und unterstützen schnelles Einfügen und schnelle Dateidurchläufe. Schwach sind Haufendateien bei der Suche nach einem Element und

dementsprechend beim Löschen des Elements, wobei der eigentliche Löschvorgang sehr schnell ist. Die Suche nach der zu ändernden Seite dauert jedoch recht lange.

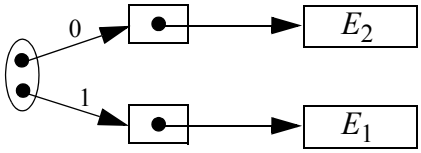
In sequentiellen Dateien werden die Daten nach den Werten eines Feldes des Datensatzes, dem sogenannten Ordnungsfeld, sortiert gespeichert. Die Speicherplatzeffizienz ist gut und Suchanfragen insbesondere mit Bereichsbedingungen werden optimal unterstützt. Allerdings sind sequentielle Dateien sehr langsam beim Einfügen und Löschen, da die Sortierung aufrechterhalten werden muss. In DBMS werden deshalb in der Regel Dateien nicht wirklich sortiert gehalten, da die B-Baumfamilie alle Vorteile einer sortierten Datei beinhaltet und zusätzlich das Einfügen und Löschen von Daten effizient unterstützt.

Hash-Dateien sind weniger speicherplatzeffizient als Haufendateien und sequentielle Dateien und bieten keinerlei Unterstützung für Suchen mit Bereichsbedingungen. Auch komplette Durchläufe sind durch die unvollständige Seitenbelegung, die zu mehr Seiten führt, langsamer. Einfügen und Löschen werden aber sehr gut und Suchen mit Gleichheitsbedingungen optimal unterstützt.

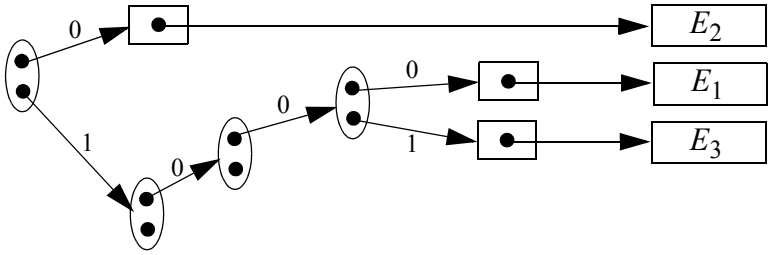
Aufgabe 2 (Dynamisches Hashing)

Nach dem Einfügen des Elements E_1 sieht der Index wie folgt aus:

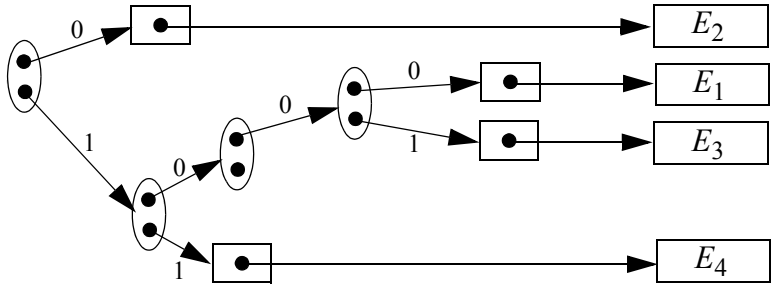
Beim Einfügen des Elements E_2 wird ein innerer Knoten angelegt:



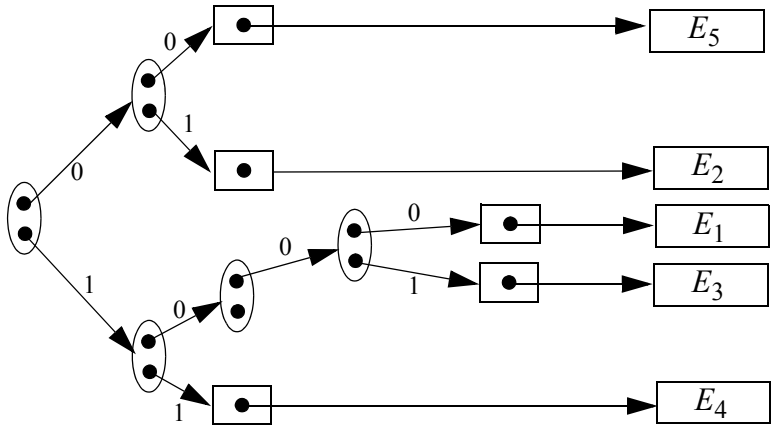
Die ersten drei Bits des Hashwertes für Element E_3 entsprechen den Bits des Hashwertes des Elements E_1. Daher vergrößert sich das Verzeichnis entsprechend:



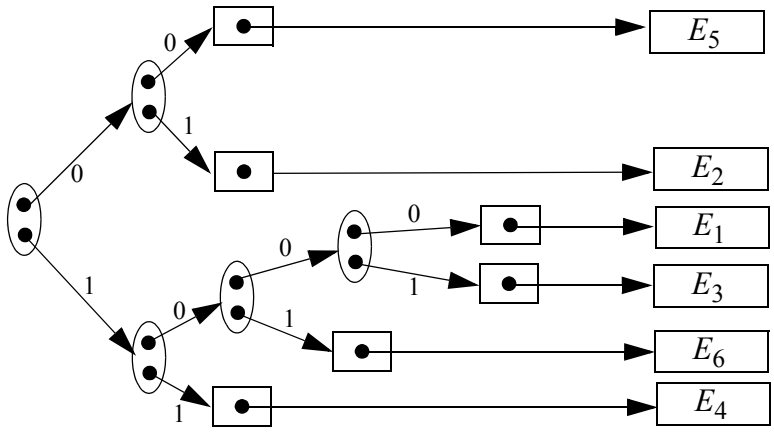
Nach Einfügen von E_4 hat der Index folgende Struktur:



Beim Einfügen von E_5 wird der Behälter, der E_2 enthält, geteilt:

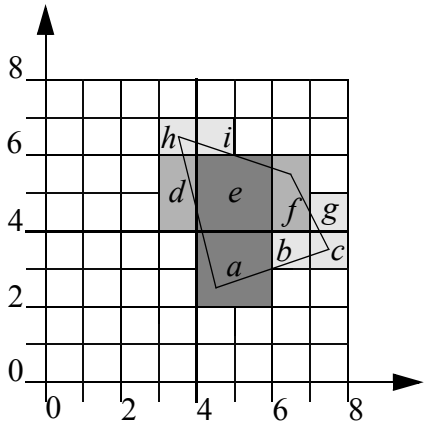


E_6 wird eingefügt, ohne das Verzeichnis zu verändern:



Aufgabe 3 (*z*-Ordnung)

(d) Das Viereck besteht aus folgenden *z*-Elementen:

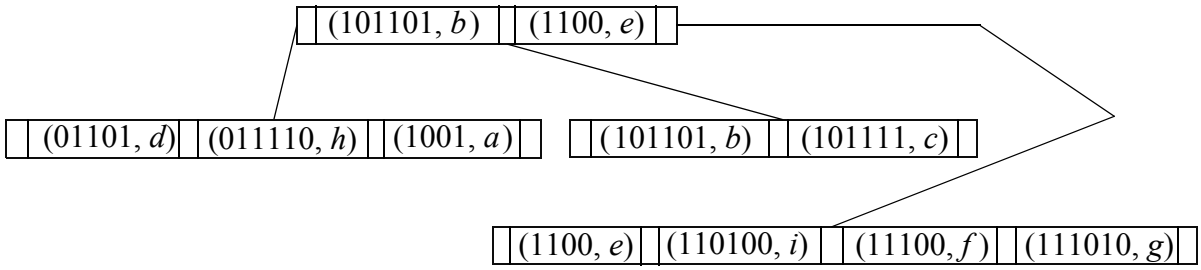


In der folgenden Tabelle stellen wir nun die *z*-Elemente durch Bitverschachtelung ihrer Zellkoordinaten dar. Jede Koordinate wird zusätzlich durch einen Index, der die Tiefe der Gitterhierarchie angibt, gekennzeichnet. Beispielsweise bezeichnet $(0_1, 2_1)$ die Zelle Nummer 2 auf der obersten Hierachieebene.

z-Element	Zellkoordinaten	Verschachtelter Bitstring
a	$2_2, 1_2 = 10, 01$	10 01
b	$6_3, 3_3 = 110, 011$	10 11 01
c	$7_3, 3_3 = 111, 011$	10 11 11
d	$3_3, 2_2 = 011, 10$	01 10 1
e	$2_2, 2_2 = 10, 10$	11 00
f	$6_3, 2_2 = 110, 10$	11 10 0
g	$7_3, 4_3 = 111, 100$	11 10 10
h	$3_3, 6_3 = 011, 110$	01 11 10
i	$4_3, 6_3 = 100, 110$	11 01 00

(e)

Der B^+-Baum hat folgende Struktur:



Hinweis für Korrektoren: Da die genaue Struktur des B^+-Baums von der Einfügereihenfolge der *z*-Elemente abhängt, akzeptieren wir jeden gültigen B^+-Baum, der alle *z*-Elemente enthält.

Das Anfragerechteck besteht aus vier *z*-Elementen, die die folgenden Ergebnisse liefern:

z-Element	Bitstring	Ergebnismenge
z_1	$(5_3, 5_3) \rightarrow (101, 101) \rightarrow 110011$	$\{e\}$
z_2	$(5_3, 3_2) \rightarrow (101, 11) \rightarrow 11011$	$\varnothing$
z_3	$(6_3, 3_3) \rightarrow (110, 11) \rightarrow 11110$	$\varnothing$
z_4	$(6_3, 5_3) \rightarrow (110, 101) \rightarrow 111001$	$\{f\}$

Somit erhält man als Ergebnis die *z*-Elemente *e* und *f*.

Aufgabe 4 (Join-Algorithmus)

Der Algorithmus funktioniert sehr ähnlich zum Sort-Merge-Join.

```
algorithm indexjoin(Iterator r, Iterator s)
  r.init(); s.init();
  ok_s := s.next(); // ok_s zeigt an, ob s noch Elemente enthält
  ok_r := r.next();
  sc := s;
  res := leere Relation;

  while ok_s and ok_r do
    while ok_s and s.key()=r.key() do
      t := concat(r.tuple(), s.tuple());
      append(res, t);
      ok_s = s.next();
    od
    s := sc;
    ok_s := true;
    while( ok_s and s.key() < r.key()) do
```

```
        ok_s = s.next();
    od
    sc := s;
    ok_r := r.next();
od
return res;
```

Aufgabe 5 Deckblatt

Hier erhalten Sie den Punkt, wenn Sie beide Klausurdeckblätter korrekt und vollständig ausgefüllt haben, also Namen, Matrikelnummer und Adresse korrekt eingetragen und genau diejenigen Aufgaben markiert haben, die Sie auch bearbeitet haben.