Naam:
Adres:
Postcode
Woonpl

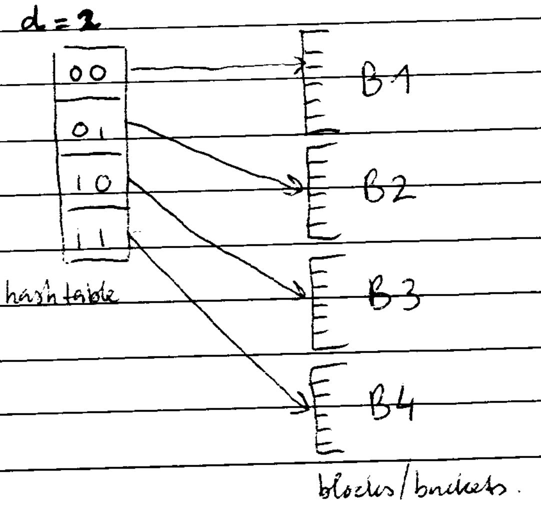
Studierichting: Informatica
Jaar van eerste inschrijving: 1996

Blad nr.:
Tentamen: Databases
Datum: 08-03-1999
Naam docent: Jongejan

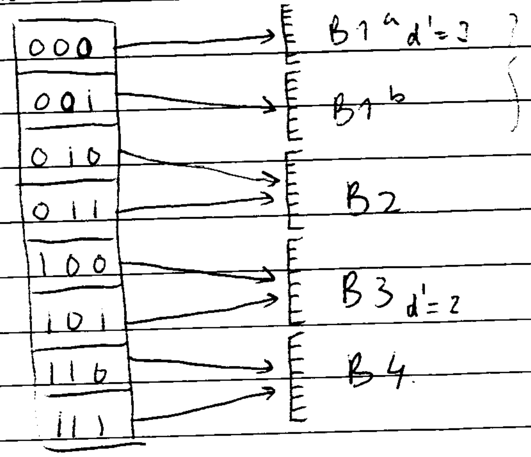
1. a. Bij extendible hashing wordt gebruik gemaakt van de binaire representatie van de hash-waarde die een hash-functie (bijv. $h(k) = k \bmod M$, bij M "buckets") oplevert. We kijken nu slechts naar de eerste d bits van deze bitstring, waarbij d de huidige "diepte" heet. We hebben een array van 2^d waarden, voor elke combinatie van d bits een entry. Ieder van deze entries wijst naar een bucket met records of naar een block op de disk waar de records staan. Wanneer door toevoegen van een record een bucket of block vol is, dan wordt deze gesplitst in tweeën, waarbij alle records op grond van hun $(d+1)$-de bit verdeeld worden over de oude en de nieuwe bucket of block. De diepte van de totale hash-table wordt met één verhoogd dus de diepte wordt nu $d+1$, en de array wordt twee keer zo groot. De andere buckets krijgen nu dus twee pointers naar zich toe omdat ze nog niet gesplitst zijn. In een plaatje:

(marge: ✗ Nee, er is zoiets [illegible] totale diepte)

(marge: Ja!)



nu is bv. bucket van 00 vol: $d := 3$



Bucket B1 is gesplitst.

De andere buckets kunnen nog gesplitst worden

Wanneer er meerdere pointers naar een block of bucket wijzen, dan hoeft, als de bucket vol is, natuurlijk niet de array uitgebreid te worden, omdat er al verdubbeld is. De pointers die nu naar dezelfde bucket wijzen kunnen dan naar verschillende buckets wijzen.

b. Single-level primary indexing.
Bij deze vorm van indexing wordt een index-file gemaakt met records van het type
<key field type, block pointer type>. De data-file moet hierbij geordend zijn op het key field type.
Voor elk block dat deel uitmaakt van de data file wordt een record opgenomen in de index file, met de key-value van het eerste record in dat block en een pointer naar (het adres van) dat block. De eerste record in een block is een zogenaamd anchor-record.
Primary indexing is een zogenaamd non-dense index methode, omdat niet voor elk record een index wordt opgenomen.
(zie verder laatste blad).

(2) a. Een minimal key voor R is $\{A\}$.
Namelijk, $\{A\}^+ = \{A,B,C,D,E\}$, want:
$A \to B$ levert $\{A,B\}$
$A \to D$ levert dan $\{A,B,D\}$
$D \to C$ levert dan $\{A,B,C,D\}$
en $AC \to DE$ levert $\{A,B,C,D,E\}$.
dus $\{A\}$ is een key van R. Aangezien $\{A\}$ maar één element heeft, is de key ook minimaal.

b. $A \to B$, $AC \to D$, $AC \to E$, $A \to D$, $D \to C$, $B \to D$.
$D \in \{A\}^+$, dus C kan weg → $D \in \{A\}^+$, dus kan weg.
↳ als we deze FA weglaten

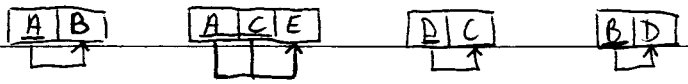
We houden over:
$A \to B$, $AC \to E$, $D \to C$, $B \to D$. $A \to E$ ipv. $AC \to E$
Hier kan niets worden weggelaten, dus deze is minimaal.

c. We hebben al een minimal cover. We nog alles samenvoegen tot relational schema's
$R = \{\{AB\}, \{ACE\}, \{DC\}, \{BD\}\}$.
In twee van deze schema's zit in ieder geval A, een key van R, dus nu hebben we een lossless-join dependency preserving decompositie. Is deze ook echt in 3NF?

Studierichting:
Jaar van eerste inschrijving:
Tentamen: Databases
Datum:
Naam docent:

(vervolg 2c)

In een plaatje:



Deze R is zelfs in BCNF, dus zeker in 3NF. 8

(3)

```
1  $FIND ANY Passagier;          Any? is dat efficient?
2  while DB-STATUS=0 do begin
3     ~~$GET Passagier;~~
4     ~~if NOT Passagier.Betaald~~
5     ~~$FIND FIRST~~
6     $GET Passagier;
7     if NOT Passagier.Betaald then begin
8        Vlucht.Naam := 'New York';
9        $FIND FIRST Vlucht WITHIN Boeking USING Naar;
10       while DB-STATUS=0 do begin
11          $GET Vlucht;
12          Piloot.Naam := 'Janssen';
13          $FIND OWNER WITHIN Vliegt ~~USING Naam~~;   (er is naam + woonpl)
14          if DB-STATUS=0 then begin
15             writeln(Passagier.Naam);
16          end;
17          $FIND NEXT Vlucht WITHIN Boeking USING Naar;
18       end;
19    end;
20    $FIND DUPLICATE Passagier USING Schild;   ?   (duplicaat van wat?)
21 end;
```

- Allereerst zoeken we alle passagiers die nog niet betaald hebben (regels 1, 2, 6, 7 en 20).
- Van elke van deze passagiers kijken we of ze een vlucht geboekt hebben naar 'New York' (regels 8, 9, 10, 11 en 17).
- Van al hun geboekte vluchten kijken we of er één (of meer) bij zijn die worden gevlogen door piloot 'Janssen' (regels 12, 13, 14).
- Als dit zo is, dan drukken we de passagiersnaam af (regel 15).

(5) (SELECT t.PNR AS PROJNR, t.MNR AS MEDNR, t.MNAAM AS MEDNAAM
FROM (PM NATURAL JOIN MW) t
WHERE t.ROL = 'PL')
UNION
(SELECT p.PNR AS PROJNR, m.MNR AS MEDNR, m.MNAAM AS MEDNAAM
FROM PM p, MW m
WHERE COUNT (SELECT d.* FROM PM d
WHERE d.PNR = p.PNR AND
d.MNR = m.MNR) > 1.

(6) Wat we moeten doen, is de tabellen MW en AFD samenvoegen via een natural join en deze m.b.v. de Part-functie opdelen in verschillende tabellen, per afdeling. Wanneer de som van alle salarissen op een afdeling hoger is dan het budget op deze afdeling, dan moeten we de gegevens van de werknemers en de afdeling in de resultaattabel plaatsen. Het omrekenen van het salaris naar een 40-urige werkweek gaat als volgt:
(salaris * 40) / (aantal uren per week dat de werknemer werkt).
De query wordt dan:

$\lambda v \in UMA : \cup \{ \{ (MNR;\ t(MNR)),\ (MNAAM;\ t(MNAAM)),$
$(SAL40;\ (t(SAL) * 40) / t(AUW)),$
$(ANR;\ t(ANR)),\ (ANAAM;\ t(ANAAM)) \mid t \in G \} \mid$
$G \in Part(v(MW) \bowtie v(AFD),\ \{ANR\}) \wedge$
$(\Sigma\, t \in G : t(SAL)) > (\Sigma\, t \in G \| \{BUDGET\} : t(BUDGET)) \}$

(vraag 1b) Wanneer nu gezocht wordt op de primary key, dan geldt dat wanneer de gezochte key $k \geq k_i$ en $k < k_{i+1}$, dan zit k in het block waarnaar gewezen wordt door pointer P_i. (Als we de records zien als $< k_i, P_i >$.) Wanneer de index gesorteerd is (en dat is zo) kan m.b.v. binary search de gezochte pointer gevonden worden. Over het algemeen is dit een snellere manier om een record te vinden dan m.b.v. binary search over de data file, aangezien in de meeste gevallen (als de records niet net zo groot zijn als de blocks, of nóg groter) minder block accesses nodig zijn.

8