UNIVERSITETET
I OSLO

Indeksering

Konvensjonelle indekser
B-trær og hashing
Flerdimensjonale indekser
Treliknende strukturer
Hashliknende strukturer
Bitmapindekser

Indekser

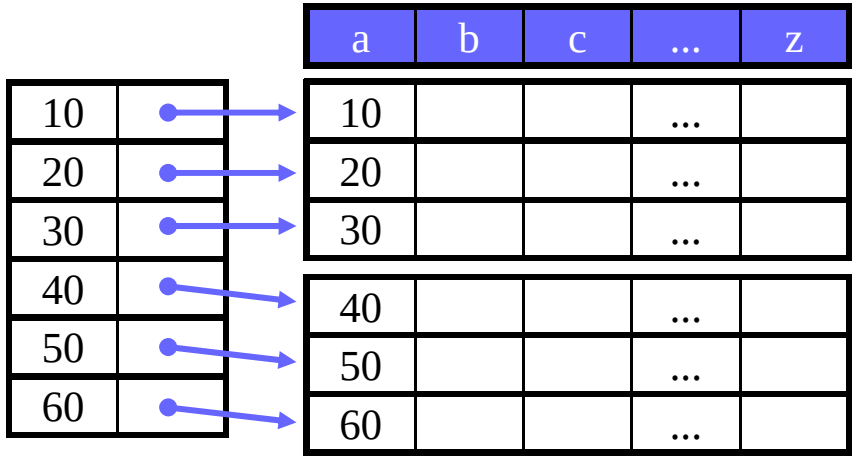
- En **indeks** på et attributt A er en datastruktur som gjør det lett å finne de elementene som har en bestemt verdi for A (**søkenøkkelen**).
- Indeksen er **sortert** på søkenøkkelen.
- For hver verdi av søkenøkkelen har indeksen en liste med pekere til de tilsvarende postene.
- Flere indekser på samme fil gir
  - raskere søking
  - økt kompleksitet – endringer må også oppdatere indeksene
  - økt lagerbehov

Ulike typer indekser

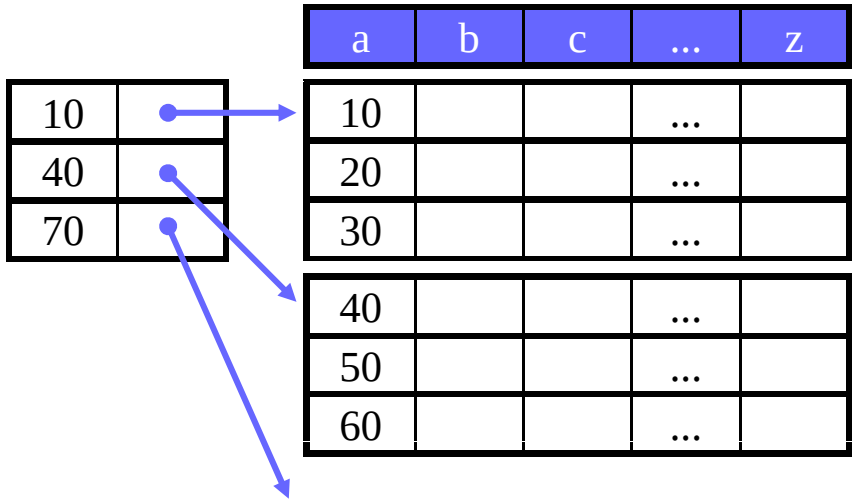
- Tett (dense) vs tynn (sparse)
- Primærindeks
- Clusterindeks
- Sekundærindeks

Tette og tynne indekser

- En **tett** indeks har ett oppslag for hver verdi av søkenøkkelen.



- En **tynn** indeks har ett oppslag for hver datablokk.



Et lite regnestykke

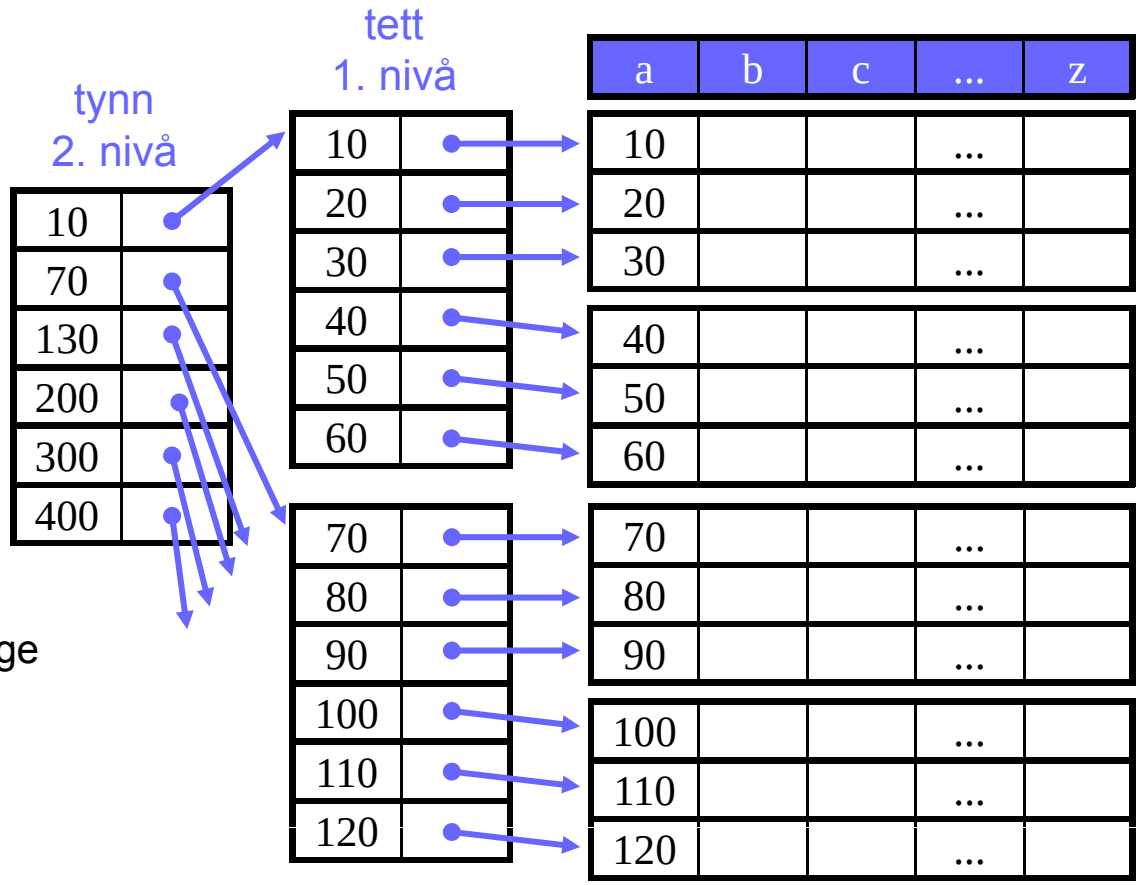
- Anta at vi har
  - 1.000.000 poster på 300B, 4B søkenøkkel, 4B pekere
  - 4KB blokkstørrelse, snitt 5.6 ms for å hente en blokk
  - 13.6 recorder pr. blokk, dvs. 76924 blokker med data
  - 512 indekser pr. blokk,
    dvs. 1954 blokker for en tett indeks og 151 blokker for en tynn
- Uten indeks:
  - 76924/2 = 38462 blokkaksesser i snitt,
    dette tar 38462 * 5.6 ms = 215.4 s
- Med tett indeks og binærsøk:
  - $\lceil \log2(1954) \rceil + 1 = 11 + 1 = 12$ blokkaksesser (maks),
    dette tar 12 * 5.6 ms = 67.2 ms
  - 3205 ganger raskere enn uten indeks!
- Med tynn indeks og binærsøk:
  - $\lceil \log2(151) \rceil + 1 = 8 + 1 = 9$ blokkaksesser (maks),
    dette tar 9 * 5.6 ms = 50.4 ms
  - 4272 ganger raskere enn uten indeks og 1.33 ganger raskere enn med tett indeks

Flernivåindekser

- En indeks kan oppta flere blokker.
- En flernivåindeks, dvs. en indeks på indeksen, kan øke effektiviteten.

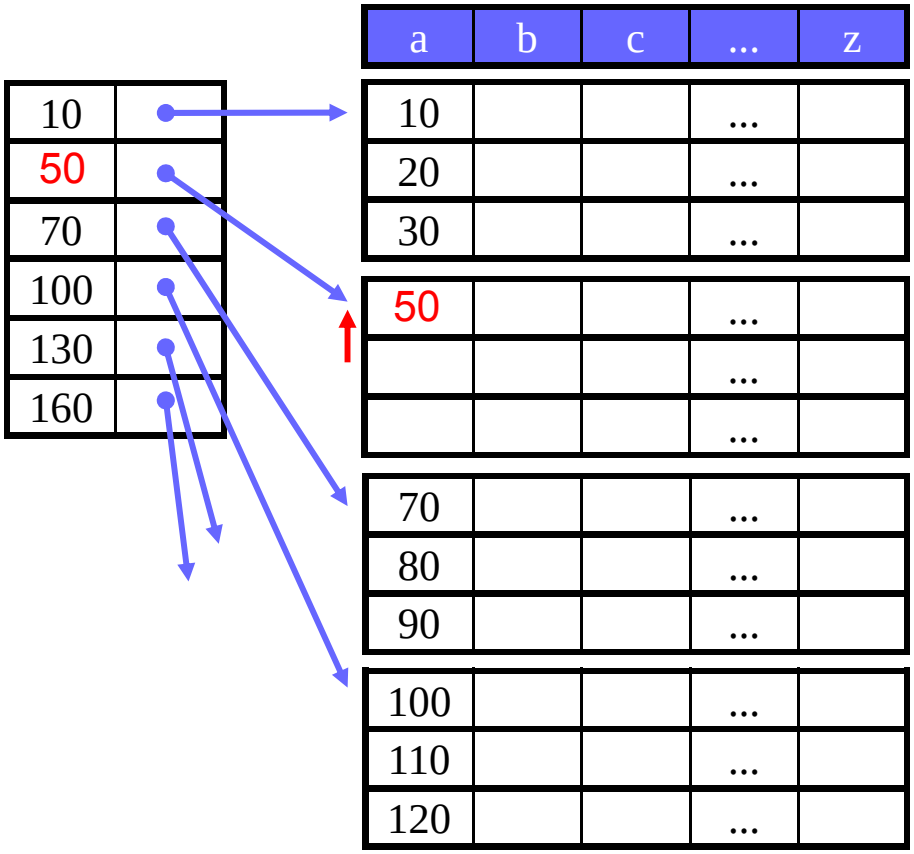
- Forts. regnestykket:
  - trenger bare 1954 / 512 = 4 blokker for 2. nivå
  - $\lceil \log_2(4) \rceil + 1 + 1 =$ 2 + 1 + 1 = 4 blokkaksesser, dette tar 4 * 5.6 ms = 22.4 ms
  - 2.25 ganger raskere enn en enkelt tynn indeks, 3 ganger raskere enn en tett indeks.

- Kan i prinsippet ha vilkårlig mange indekser.



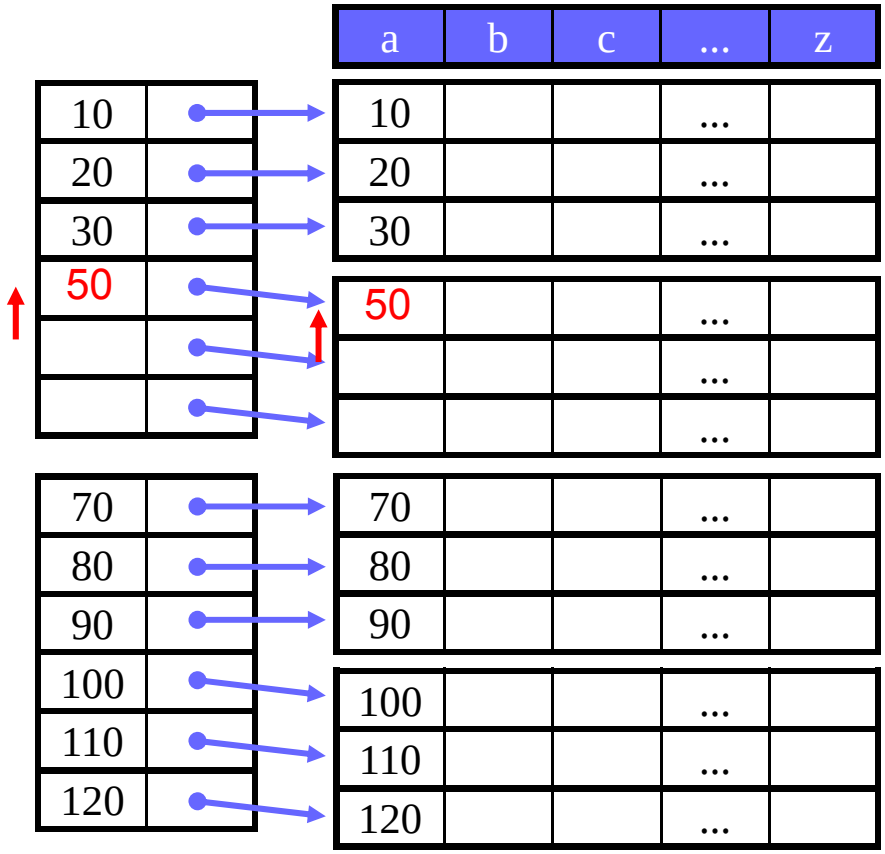
Eksempel: Sletting ved tynn indeks

- Slett post med a = 60
  - Ingen endring nødvendig i indeksen.
- Slett post med a = 40
  - Den første posten i blokken er blitt oppdatert, så indeksen må også oppdateres.



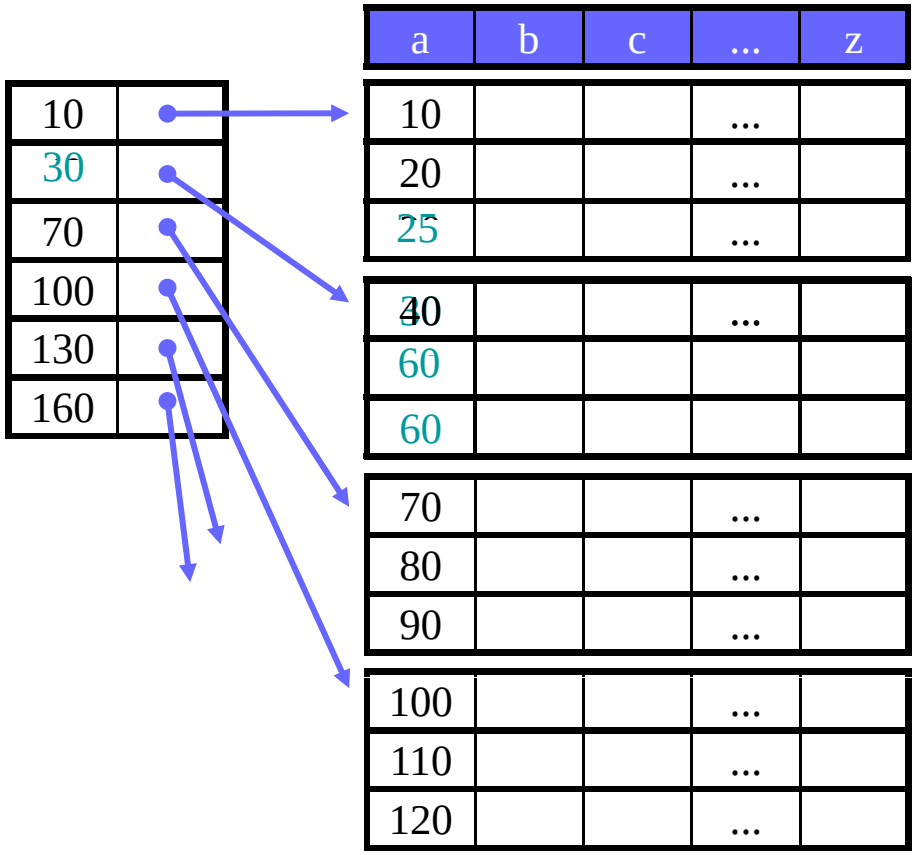
Eksempel: Sletting ved tett indeks

- Slett post med a = 60
- Slett post med a = 40
  - I mange tilfeller ønsker man å "komprimere" dataene i blokkene.
  - Man kan også komprimere hele datasettet, men vanligvis beholdes noe ledig plass for fremtidig innsetting.



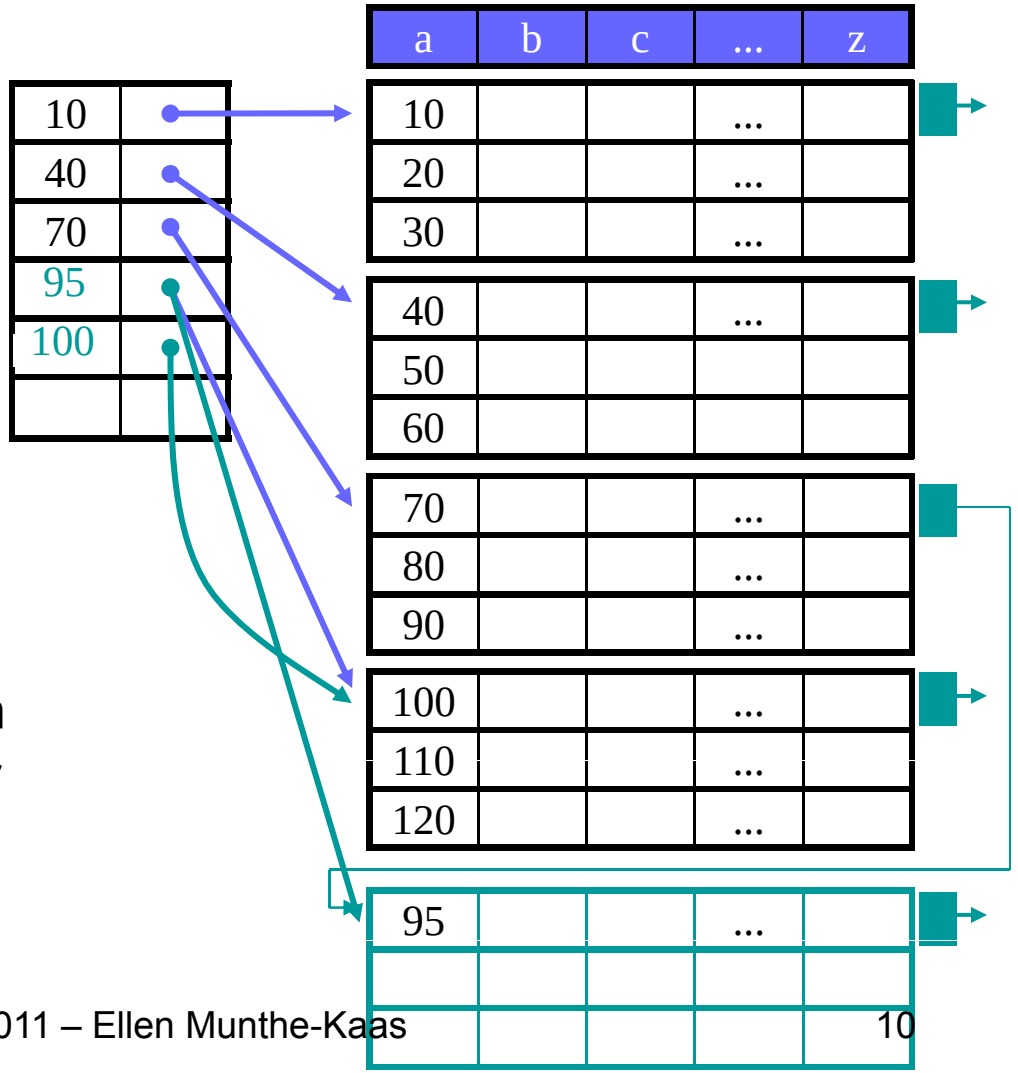
Eksempel: Innsetting ved tynn indeks

- Sett inn post med a = 60
  - Vi er heldige – ledig plass der vi trenger det.
- Sett inn post med a = 25
  - Må flytte posten med a=30 til neste blokk for å lage plass.
  - Den første posten i blokk to er endret, og indeksen må oppdateres.
  - Merk: Kunne også satt inn en ny/overflytsblokk.



Eksempel: Innsetting ved tynn indeks

- Sett inn post med a = 95
  - Ikke plass – sett inn ny/overflytsblokk
    - Overflytsblokk: Trenger ikke gjøre noe i indeksen (har bare pekere til hovedblokkene).
    - Ny blokk: Indeksen må oppdateres.
- Innsetting ved tette indekser gjøres på samme måte – men indeksen må oppdateres hver gang.

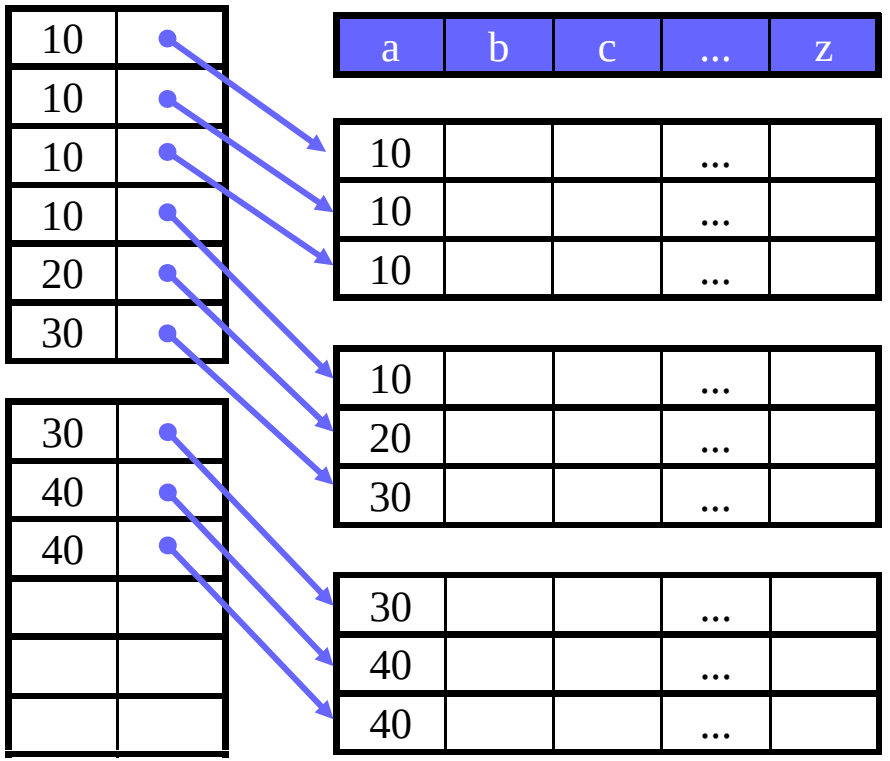


Tette versus tynne indekser

	Tett	Tynn
plass	Ett indeksfelt pr. post	Ett indeksfelt pr. datablokk
blokkaksesser	“mange”	“få”
postaksesser	Direkte aksess	Må lete innenfor blokken
exist-spørringer	Bruk indeksen alene	Må alltid aksessere blokken
bruk	overalt	Ikke på uordnede elementer
endringer	Må alltid oppdateres hvis postrekkefølgen endres	Oppdateres bare hvis den første posten i en blokk endres

Duplikate søkenøkler (clusterindekser)

- En clusterindeks kan brukes hvis filen er sortert selv om søkenøkkelen ikke er unik.
- Eksempel 1 – tett indeks:
  - Ett indeksfelt pr. post.
  - ☺ Lett å finne poster og hvor mange som finnes av hver.
  - ☹ Flere felt enn nødvendig?

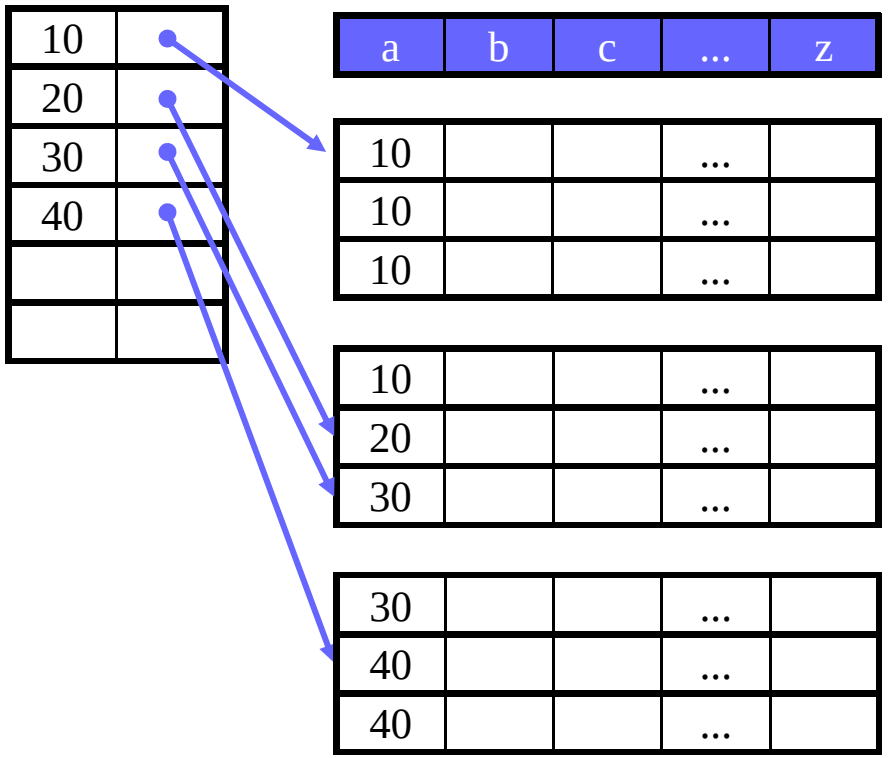


Clusterindekser: Eksempel 2 – tett indeks

- Bare ett indeksfelt pr. unike søkenøkkel.

  ☺ Mindre indeks – raskt søk.

  ☹ Mer komplisert å finne påfølgende poster.

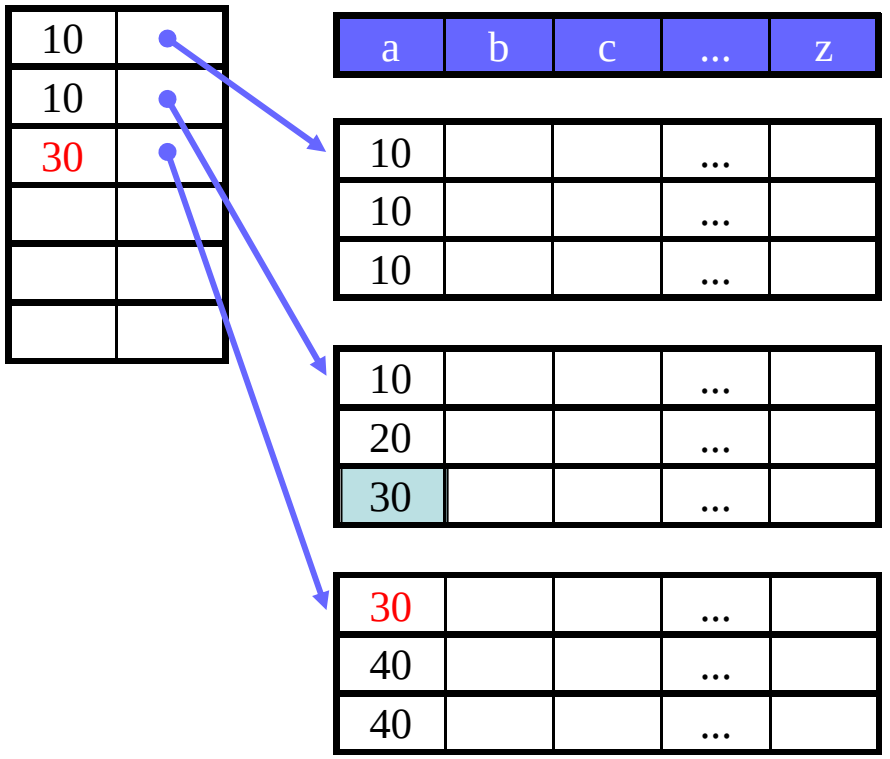


Clusterindekser: Eksempel 3 – tynn indeks

- Indeksfeltene peker til første post i hver blokk

  ☺ Liten indeks – raskt søk

  ☹ Vanskelig å finne poster.
  - F.eks. finn alle poster med a = 30.

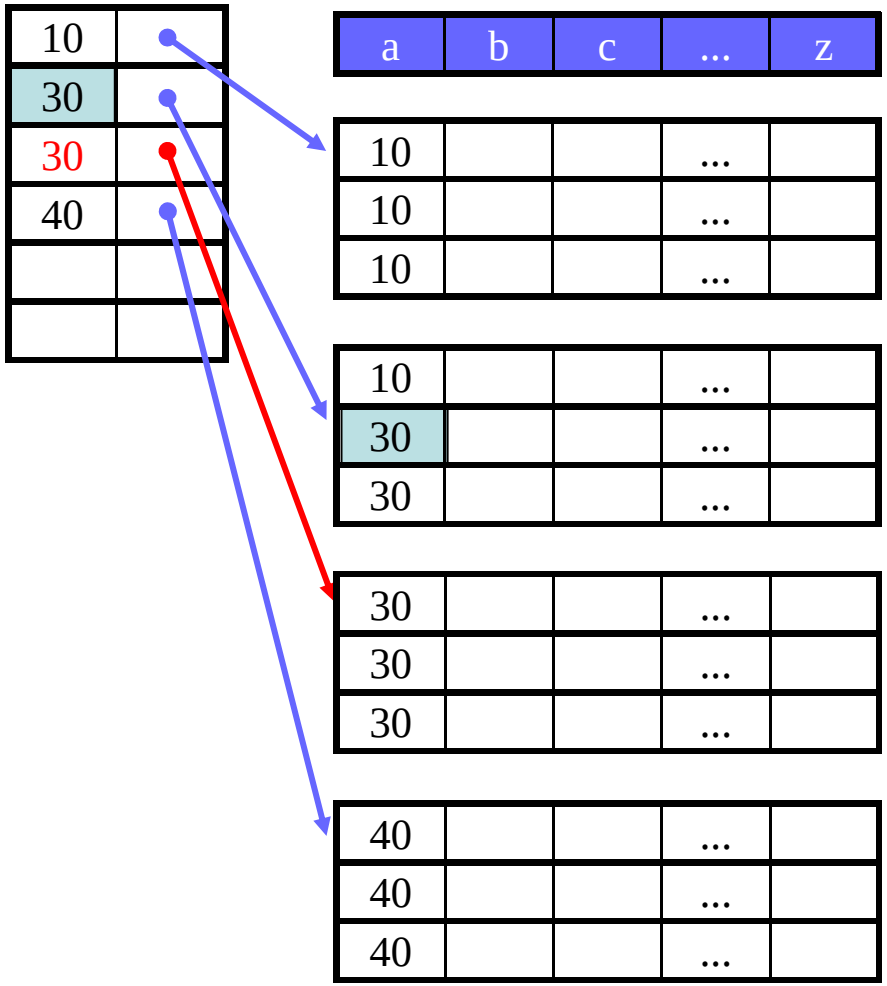


Clusterindekser: Eksempel 4 – tynn indeks

- Indeksfeltet er første *nye* post i hver blokk.

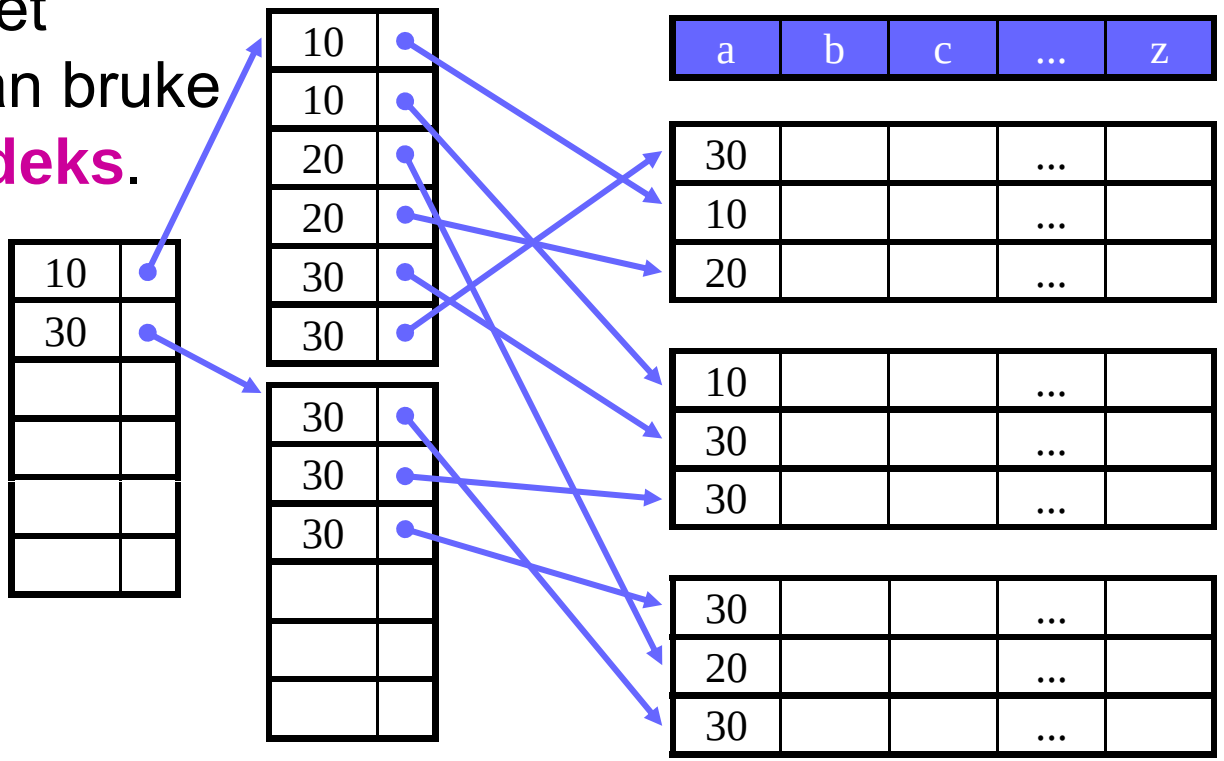
  ☺ Liten indeks – raskt søk.

  ☹ Vanskelig å fjerne poster.
  - Kan vi i det hele tatt fjerne den andre forekomsten med a = 30?



Usorterte filer – sekundærindekser

- Hvis filen er usortert (eller sortert på et annet attributt), kan man bruke en **sekundærindeks**.
  - Sortert på søke-nøkkelen – raskt søk.
  - 1. nivå er alltid tett – høyere nivåer er tynne.
  - Duplikater tillates.

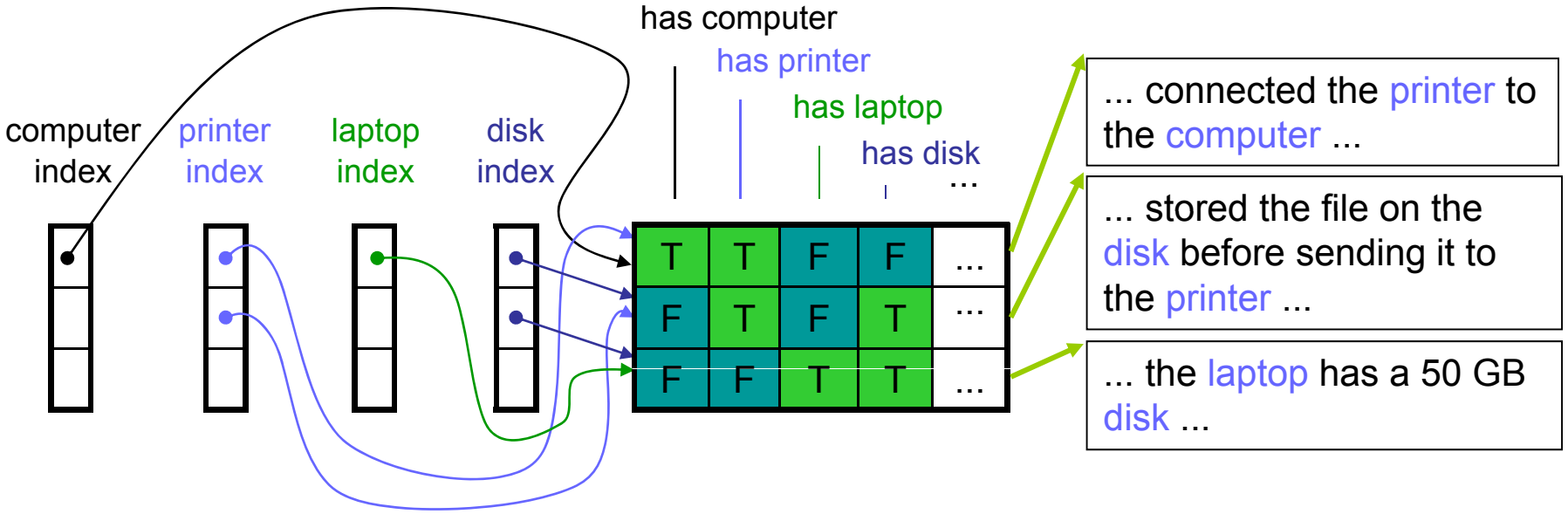


Inverterte indekser

- Hva hvis man ønsker å søke på elementer innenfor et attributt?
  - `SELECT * FROM R WHERE a LIKE '%cat%'`
  - Søke etter dokumenter som inneholder visse nøkkelord, f.eks. søkemotorer som Google, Altavista, Excite, Lycos, AllTheWeb osv.

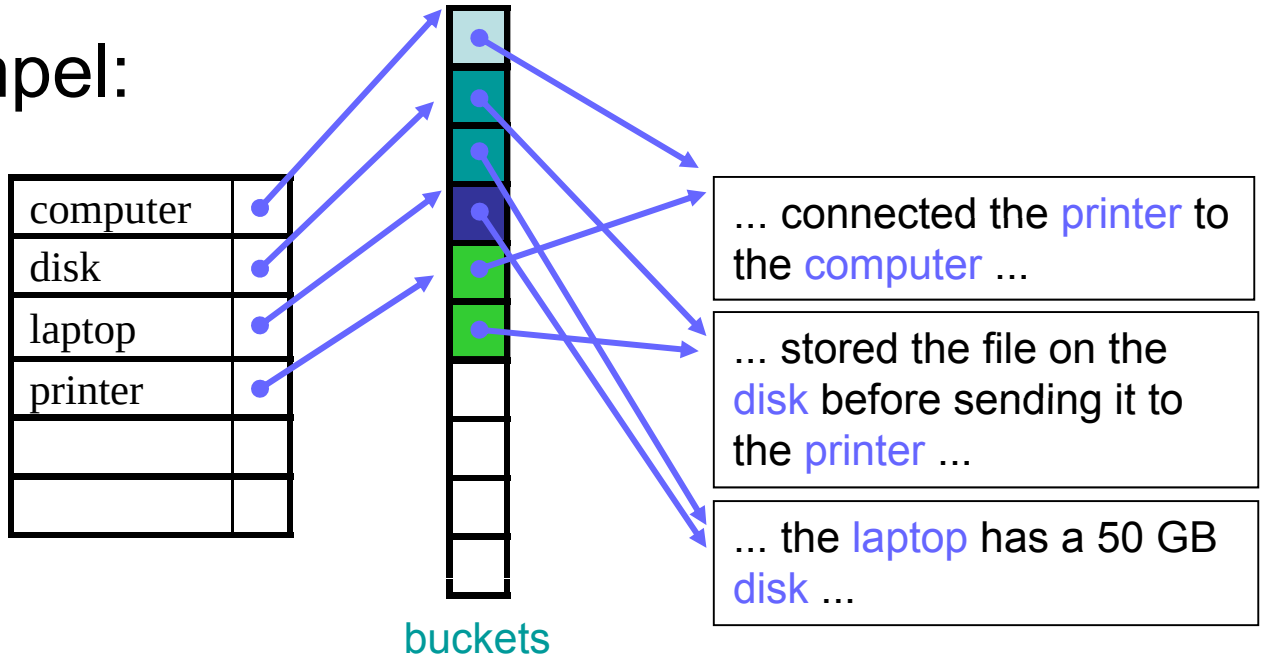
Alternativ 1: true/false-tabell

- Eksempel:
  tillatte nøkkelord – computer, printer, laptop, disk, ...



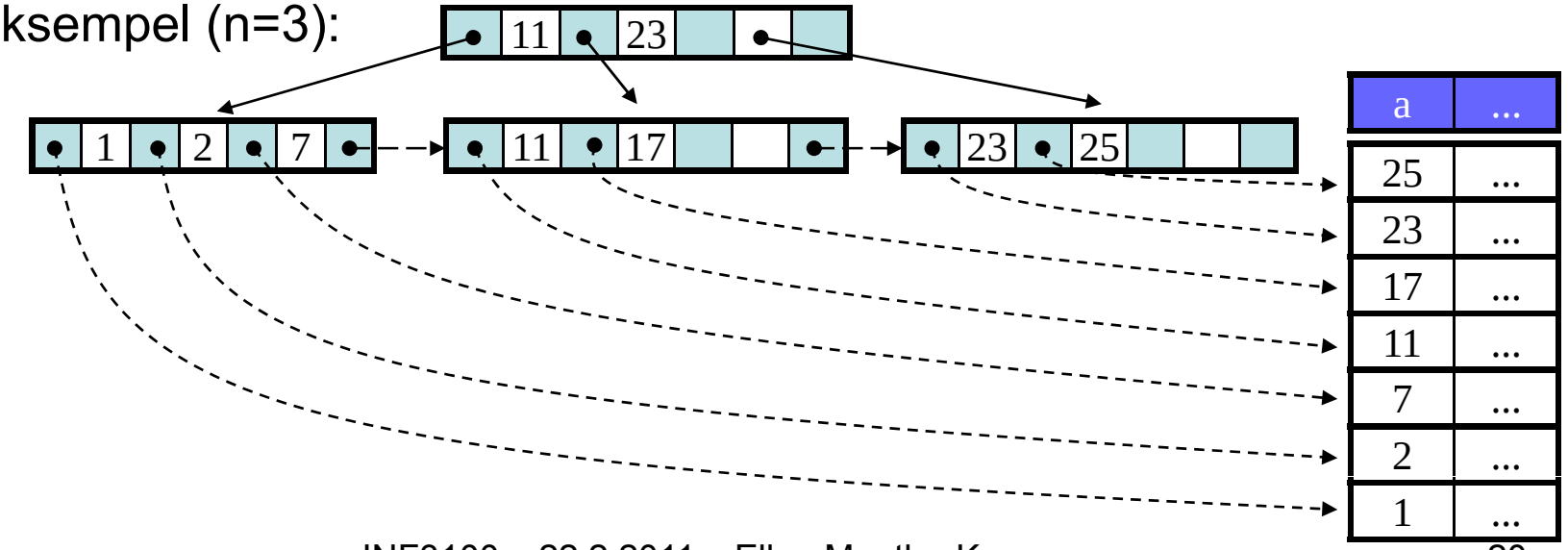
Alternativ 2: Invertert indeks

- Eksempel:



B^+-trær

- Hver node har *n* søkenøkler og *n+1* pekere.
  - Indre noder: alle pekere er til subnoder.
  - Løvnoder: n datapekere og 1 nestepeker.
- Ingen noder kan være tomme.
  - Indre noder: minst $\lceil (n+1)/2 \rceil$ pekere til subnoder.
  - Løvnoder: minst $\lfloor (n+1)/2 \rfloor$ datapekere.
- Eksempel (n=3):

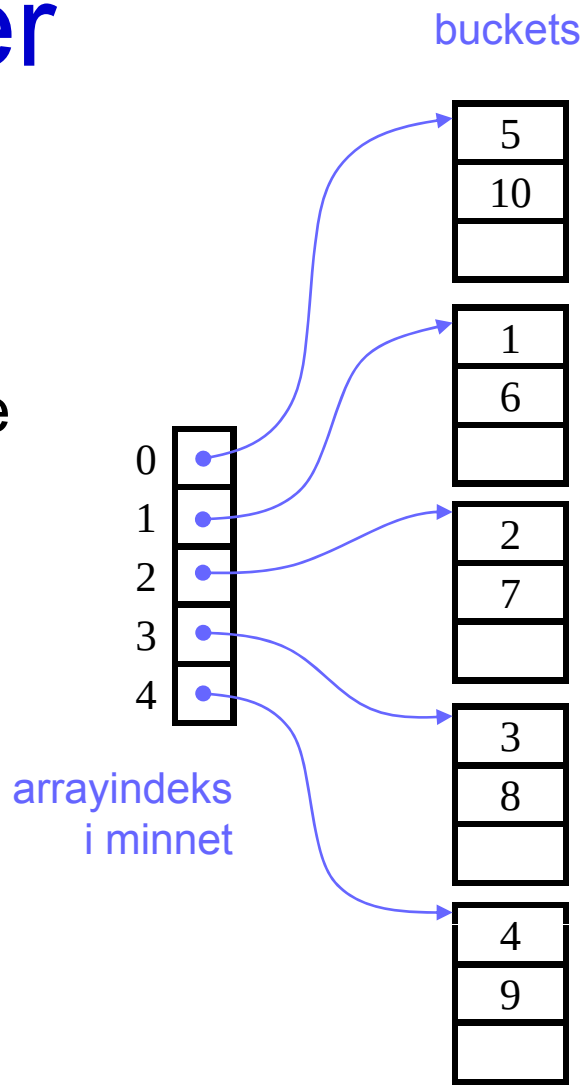


B^+-trær: effektivitet

- B^+-trær:
  - ☹ Et søk må alltid gå fra roten til en løvnode, dvs.
    antall blokkaksesser er lik høyden på treet pluss aksessering av selve postene.
  - ☺ Antall nivåer er vanligvis veldig lavt – typisk 3.
  - ☺ Intervallsøking går veldig raskt.
  - ☺ Ved stor *n* er det sjelden nødvendig å splitte/slå sammen noder.
  - ☺ Disk I/O kan reduseres ved å holde indeksblokkene i minnet.
- Eksempel: 4B søkenøkler, 8B pekere, 4KB blokker (ingen headere)
  - Hvor mange verdier kan lagres i hver node?
    $4n + 8(n+1) \leq 4096$ ⇨ n = 340
  - Nodene er i snitt 75% fulle. Hvor mange poster kan et B^+-tre med 3 nivåer inneholde?
    $(340 * 75\ \%)^3 = 16581375 \approx$ 16.6 million poster

Hashtabeller

- Bruker en hashfunksjon fra søkenøkkelen til en arrayindeks med peker videre til hvilken bøtte (bucket) som eventuelt inneholder den aktuelle posten.
  - Arraystørrelsen er vanligvis et primtall
  - Viktig med en god hashfunksjon!
    - Rask
    - God fordeling av søkenøklene
- Eksempel:
  - Arraystørrelse B = 5
  - h(key) = mod(key, B)



Hashtabeller: effektivitet

- Ideelt er arraystørrelsen stor nok til at alle elementene for en hashverdi passer i en bucketblokk.
  - ☺ Da får vi signifikant færre diskoperasjoner enn med vanlige indekser og B-trær
  - ☺ Raskt søk etter spesifikk søkenøkkel
  - ☹ Flere poster kan føre til flere blokker pr. bucket
  - ☹ Dårlig på intervallsøk

Dynamiske hashtabeller

- Vanskelig å holde alle elementene innenfor en bucketblokk hvis antall poster øker mens hashtabellen er statisk
- *Dynamiske* hashtabeller tillater tabellstørrelsen å variere slik at det holder med en blokk pr. bucket.
  - utvidbar (extensible) hashing
  - lineær (linear) hashing

Les detaljer om disse selv!

Sekvensielle vs hashindekser

- *Sekvensielle indekser* som f.eks. B-trær er gode på intervallsøk:

```
SELECT * FROM R WHERE R.A > 5
```

- *Hashindekser* er gode når det søkes etter en spesiell nøkkel:

```
SELECT * FROM R WHERE R.A = 5
```

Indekser i SQL

- Syntaks[1]:
  - `CREATE INDEX` name `ON` relation_name (attribute)
  - `CREATE UNIQUE INDEX` name `ON` relation_name (attribute)
    → definerer en kandidatnøkkel

  - `DROP INDEX` name

- Merk: Kan ikke[1] angi
  - Type indeks, f.eks. B-tre, hashing osv.
  - Parametre som loadfaktor, hashstørrelse osv.

[1]DBMS-avhengig

Spørringer med flere betingelser

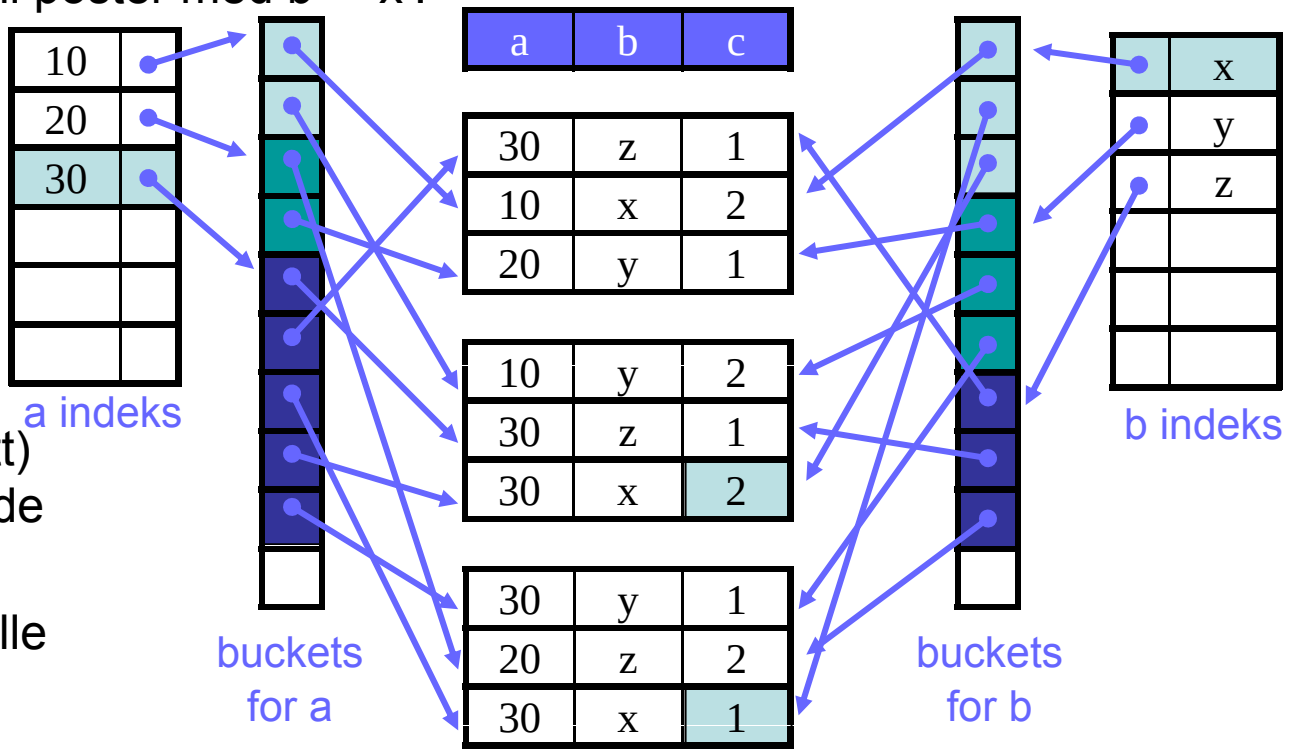
- `SELECT ... FROM R WHERE a = 30` **`AND`** `b < 5`
- Strategi 1:
  - Bruk en indeks, f.eks. på a.
  - Finn og *hent* alle postene med a = 30.
  - Søk gjennom disse postene for å finne dem med b < 5.

☺ Enkel strategi.

☹ Risikerer å lese mange unødvendige poster fra disk.

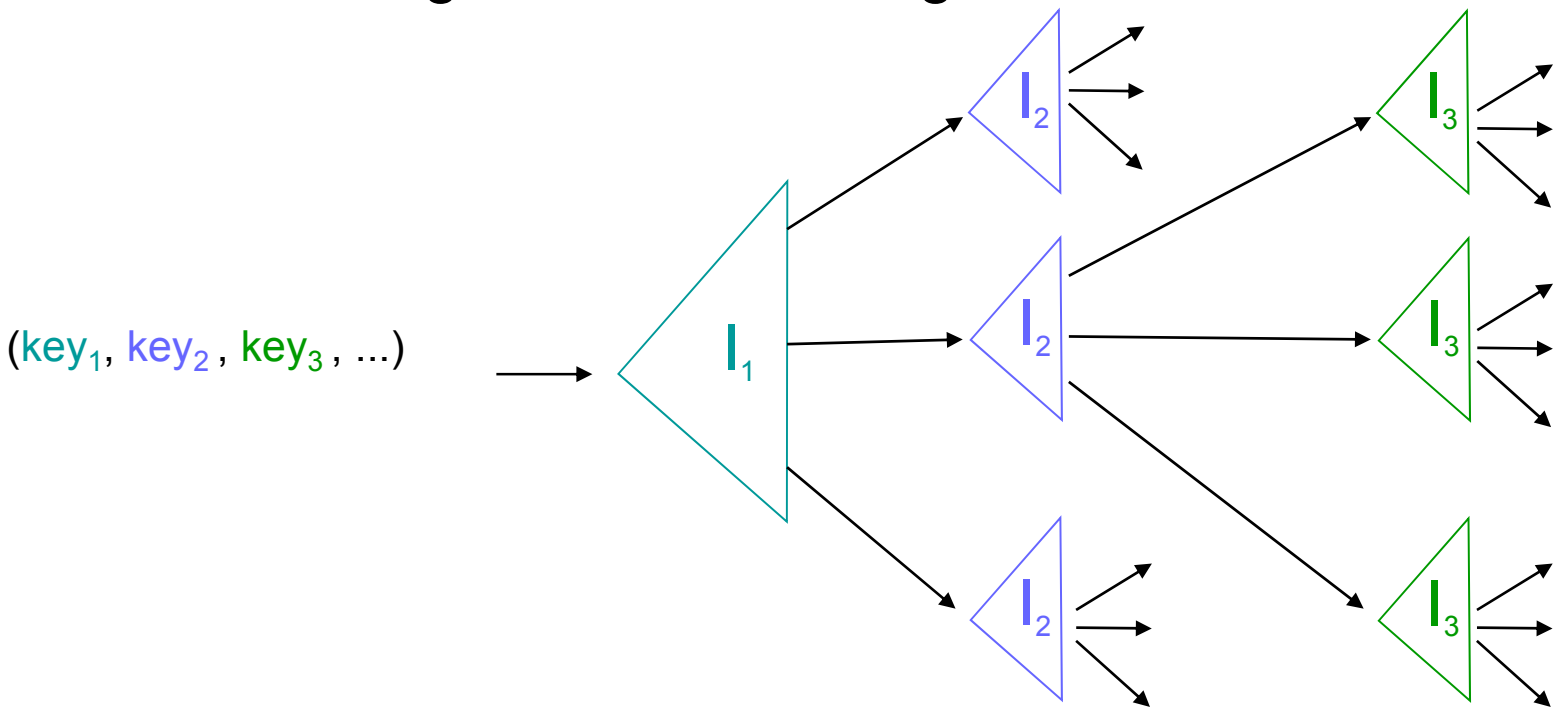
Flere betingelser: strategi 2

- `SELECT c FROM R WHERE a=30 AND b='x'`
  - Bruk to tette indekser, en for a og en for b.
  - Finn alle pekere til poster med a = 30.
  - Finn alle pekere til poster med b = 'x'.
  - Sammenlign (snitt) pekerne og hent de aktuelle postene.
  - Plukk ut de aktuelle attributtene.



Flerdimensjonale indekser

- En flerdimensjonal indeks kombinerer flere dimensjoner i samme indeks.
- En enkel trelignende tilnærming:



Flerdimensjonale indekser: eksempel

- Eksempel, tett indeks:

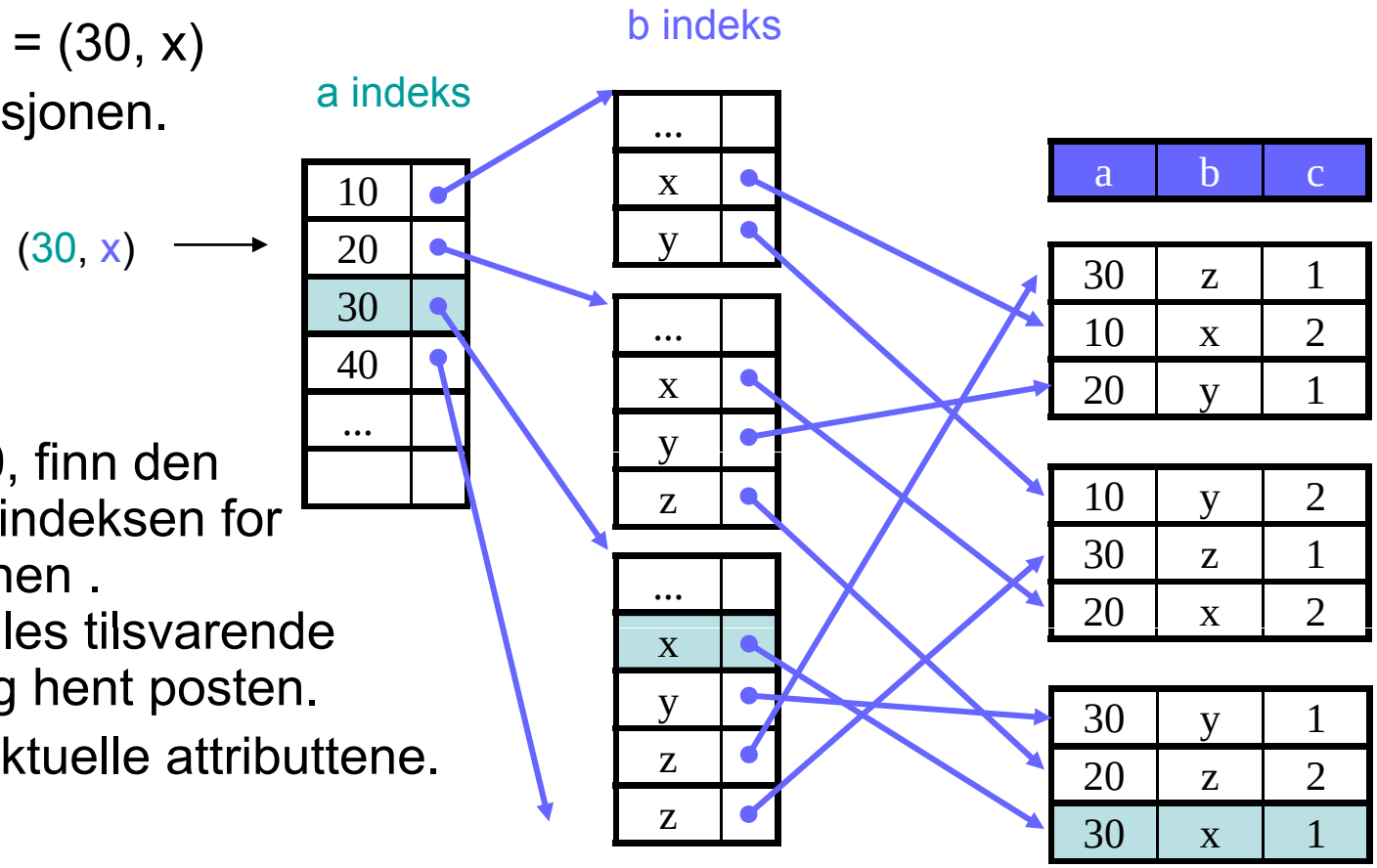
  `SELECT ... FROM R WHERE a = 30 AND b ='x'`

  – søkenøkkel = (30, x)

  – les a-dimensjonen.

  – søk etter 30, finn den tilsvarende indeksen for b-dimensjonen .
  søk etter x, les tilsvarende diskblokk og hent posten.

  – velg ut de aktuelle attributtene.



Flerdimensjonale indekser

- For hvilke spørringer er dette en god indeks?

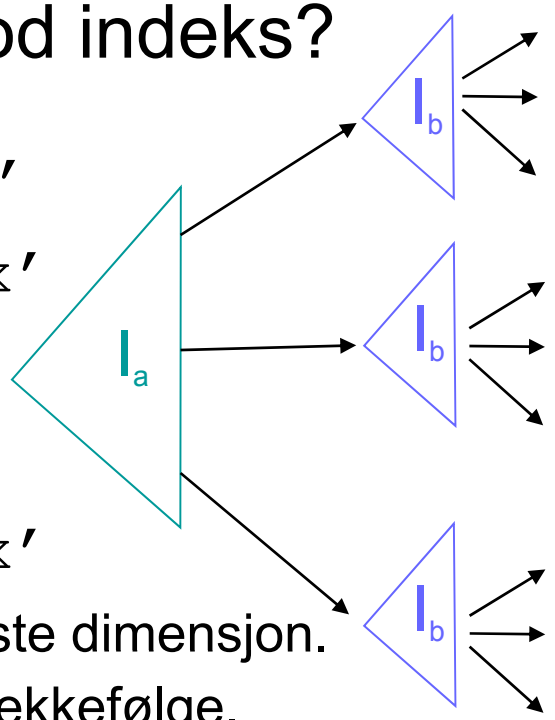
☺ Finn poster med `a = 10 AND b = 'x'`

☺ Finn poster med `a = 10 AND b ≥ 'x'`

☹ Finn poster med `a = 10`

☹ Finn poster med `b = 'x'`

? Finn poster med `a ≥ 10 AND b = 'x'`

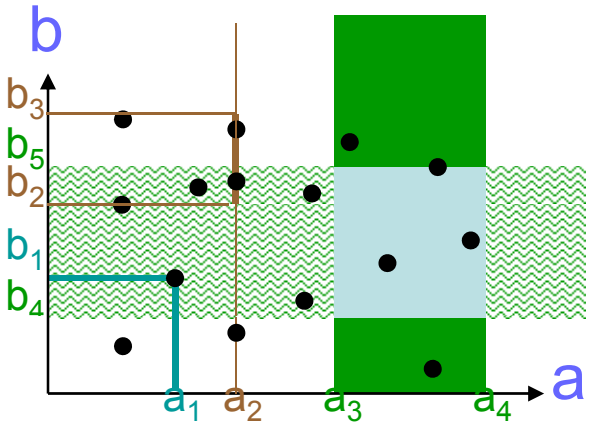


  - Risikerer å måtte søke i mange indekser i neste dimensjon.
  - Hadde vært bedre om dimensjonene endret rekkefølge.

- Det finnes mange andre tilnærminger...
  - Andre trelignende strukturer
  - Hashlignende strukturer
  - Bitmapindekser

Map View

- En flerdimensjonal indeks kan ha mange dimensjoner.
- Hvis vi holder oss til to (som i det forrige eksempelet), kan vi se på indeksen som et geografisk kart:



- Søking tilsvarer nå å søke i kartet etter
  - punkter: a_1 og b_1
  - linjer: a_2 og $<b_2,b_3>$
  - arealer: $<a_3 ,a_4>$ og $<b_4,b_5>$

Trestrukturer

- Det finnes mange trelignende strukturer som tilsvarer å lete etter kartarealer:
  - kd-trær
  - quad-trær
  - R-trær

Vi skal se på kd-trær som et eksempel, les om de andre i læreboken!

- Men alle disse må oppgi minst en av følgende egenskaper ved B-trær:
  - Balansering – alle løvnodene er på samme nivå.
  - Korrespondanse mellom trenoder og diskblokker.
  - Ytelse for oppdateringsoperasjoner.

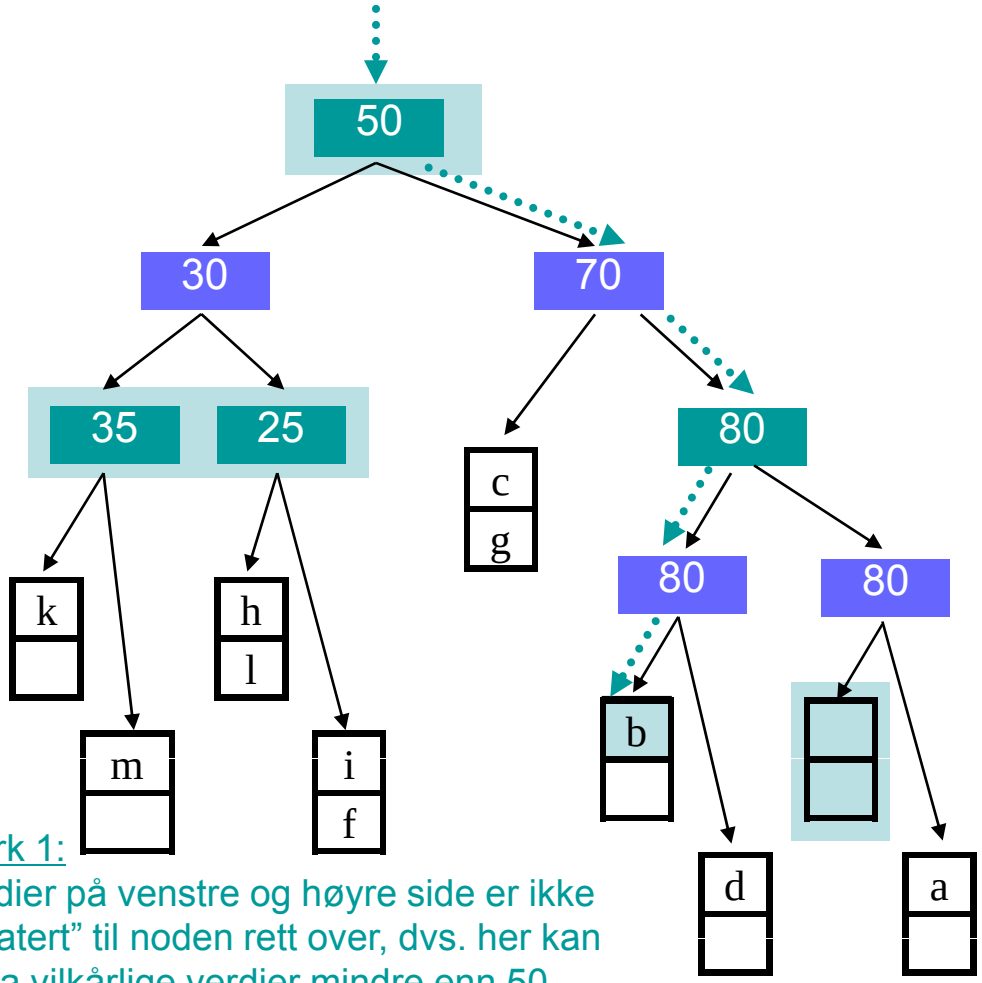
kd-trær

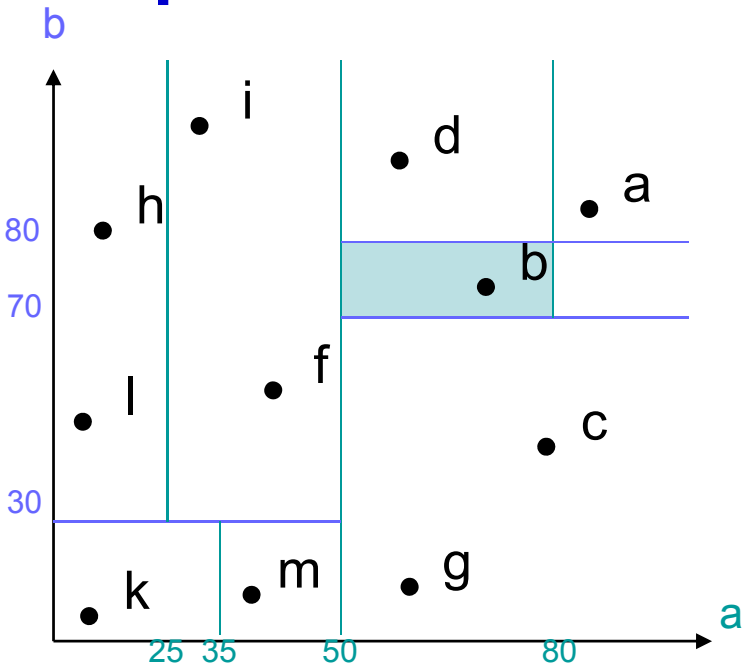
- Et kd-tre (k-dimensjonalt tre) er et binærtre hvor hver node inneholder
  - et attributt A
  - en verdi V som splitter resten av datapunktene i to
- Vi skal se på en variant hvor
  - indre noder bare inneholder V-verdien:
    - Venstre barn har verdier mindre enn V
    - Høyre barn har verdier større enn eller lik V
  - løvnoder er blokker med plass til et antall poster
- De ulike dimensjonene er flettet:

nivå 0	a-dimensjon
nivå 1	b-dimensjon
...	...
nivå n	a-dimensjon
nivå n+1	b-dimensjon
...	...

kd-trær: eksempel

Finn posten (a,b) = (70, 75)





Merk 1:
verdier på venstre og høyre side er ikke "relatert" til noden rett over, dvs. her kan vi ha vilkårlige verdier mindre enn 50.

Merk 2:
kan ha tomme blokker

kd-trær: noen "problemer"

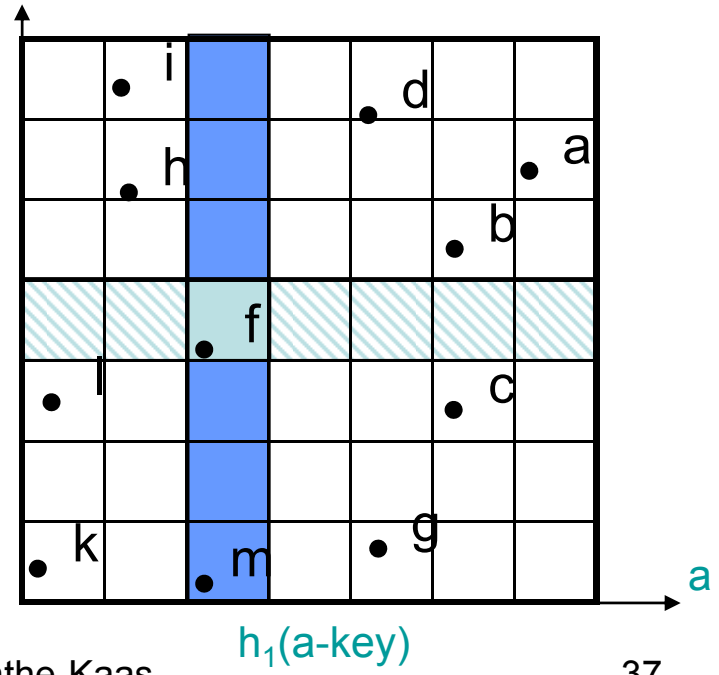
- Kan måtte sjekke begge grenene ut fra roten, f.eks. for betingelser på b-dimensjonen.
- Kan måtte sjekke begge grenene til et subtre ved intervallsøk, f.eks. ønsket intervall er <10,30> og nodeverdien er 20.
- Høyere trær sammenlignet med B-trær
  - Kostbart å lagre hver node i en egen blokk.
  - Mange flere blokkaksesser enn ved B-trær.
- Antall blokkaksesser kan reduseres hvis vi
  - splitter i mer enn to grener
  - grupperer indre noder i en blokk,
    f.eks. en node og alle etterfølgerne i et visst antall nivåer.

Hash-lignende strukturer: gridfiler

- Gridfiler utvider tradisjonelle hashindekser
  - Hasher verdier for hvert attributt i en flerdimensjonal indeks.
  - Hasher vanligvis ikke *enkeltverdier*, men *regioner* – h(key) = <x,y>
  - Gridlinjer partisjonerer området i striper.

- Eksempel (2 dimensjoner):
  - Finn post med (a,b) = (22, 31)
  - $h_1(22) = <a_x, a_y>$
  - $h_2(31) = <b_m, b_n>$

  ⇨ post f



Gridfiler

- Gridfiler kan raskt finne poster med
  - key 1 = V_i AND key 2 = X_j
  - key 1 = V_i
  - key 2 = X_j
  - key 1 $\geq V_i$ AND key 2 < X_j
- Gridfiler:
  - ☺ Bra for søking ved flere nøkler.
  - ☹ Bruker mye plass, krever en del organisering.

Les selv om partisjonerte hashfunksjoner!

Bitmapindekser

- Utgangspunkt: Alle recorder er tilordnet et uforanderlig, entydig tall
  - Nummering fra 1 til n
  - Nummeret kan betraktes som en record-ID og kan ikke gjenbrukes
- Velg ut feltet F som det skal lages en indeks på
  - For hver benyttet verdi v for F i en av recordene, opprett en bitvektor b_v med lengde n
  - Hvis record nr. i har F=v, la $b_v[i]=1$
  - Hvis record nr. i har F≠v, la $b_v[i]=0$

Bitmapindekser: eksempel

fil

record-nummer	F	G
1	30	foo
2	30	bar
3	40	baz
4	50	fou
5	40	bar
6	30	baz

bitvektorer for F

30:	1	1	0	0	0	1
40:	0	0	1	0	1	0
50:	0	0	0	1	0	0

Karakteristika

- Plassbehov:
  - Totalt antall bits er #records * #verdier
  - I verste fall trengs n^2 bits (men da har hver bitvektor bare én 1-bit)
  - Bitvektorene kan komprimeres; det er aldri mer enn n 1-bits totalt i bitvektorene
- Effektiv for
  - partial match queries (= angi verdier for noen felter, finn alle som har gitte verdier)
    - Beregn bitvis AND på tvers av bitmap-indeksene for de aktuelle attributtene
  - range queries (= angi intervaller for noen felter, finn alle som har verdier innen intervallene)
    - Beregn bitvis OR innen intervallene og bitvis AND mellom intervallene