Indeksering

Konvensjonelle indekser

B-trær og hashing

Flerdimensjonale indekser

Trestrukturer

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

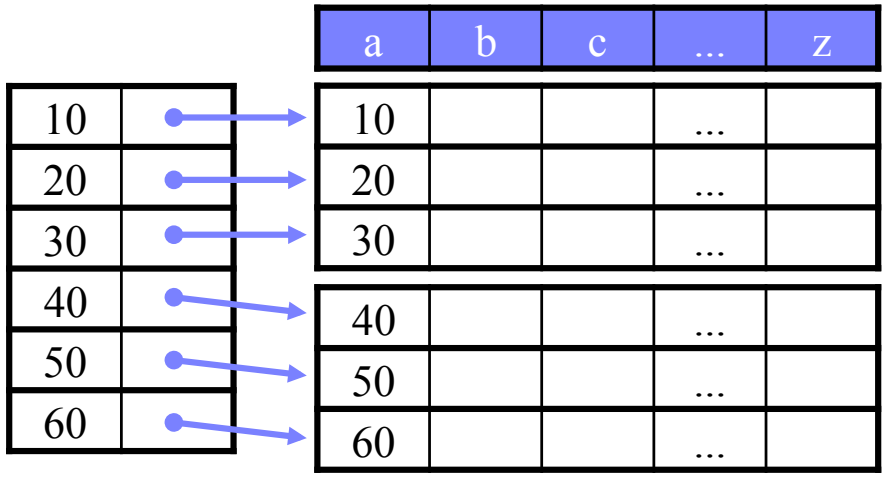
- Tett (dense) vs. tynn (sparse)
- Primærindeks
  Datafilen er sortert fysisk på søkenøkkelen
  Maks én datapost pr. søkenøkkelverdi
- Clusterindeks
  Datafilen er sortert fysisk på søkenøkkelen
  Tillatt med flere dataposter med samme søkenøkkelverdi
- Sekundærindeks
  Datafilen er ikke sortert etter den tilhørende søkenøkkelen

Indekser, oversikt

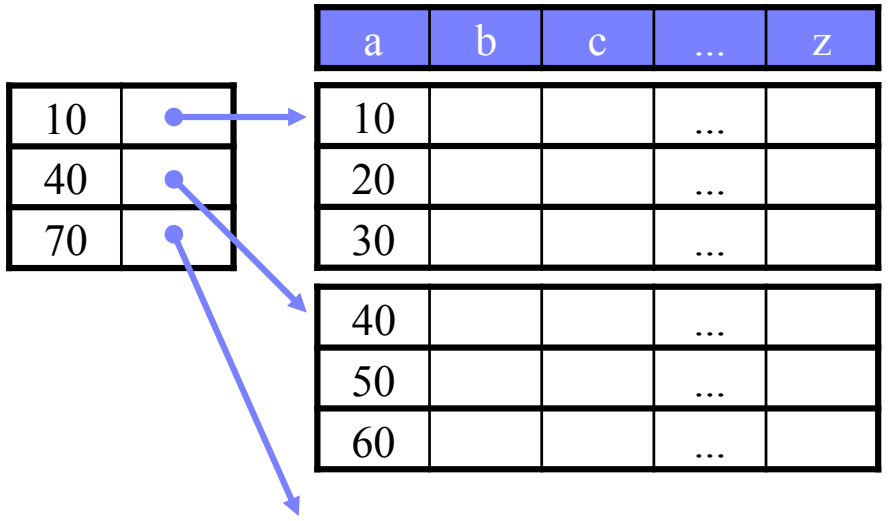
Datafil	**Søkenøkkelen er en kandidatnøkkel**	**Søkenøkkelen er ikke en kandidatnøkkel**
sortert på søkenøkkelen	primærindeks tett eller tynn	clusterindeks tett eller tynn
ikke sortert på søkenøkkelen	sekundærindeks tett	sekundærindeks tett

Primærindeks: Tette og tynne indekser

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



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

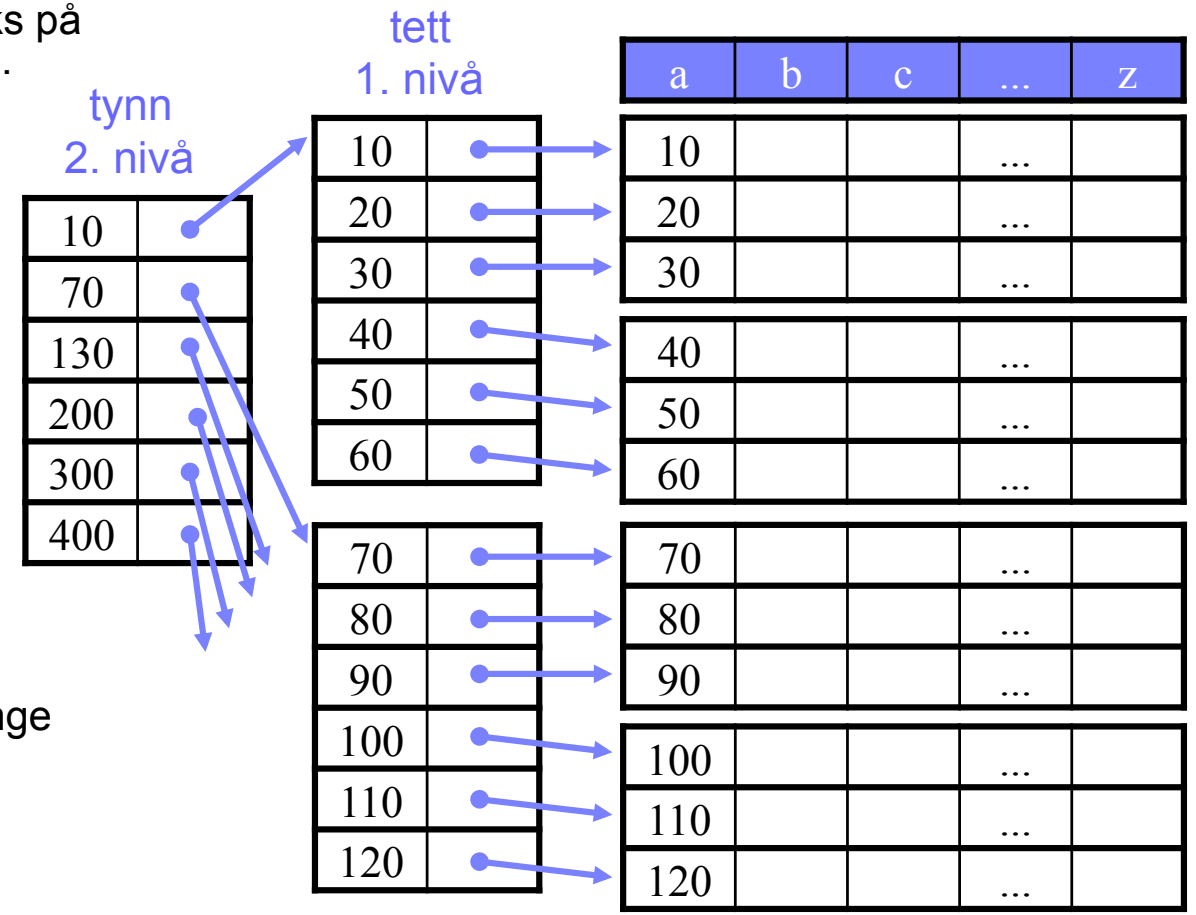


Et lite regnestykke

- 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:
  - $\lceil \log_2(1954) \rceil + 1 = 11 + 1 = 12$ blokkaksesser (maks),
    dette tar 12 * 5,6 ms = 67,2 ms
  - 3205 ganger raskere enn uten indeks!
- Med tynn indeks og binærsøk:
  - $\lceil \log_2(151) \rceil + 1 = 8 + 1 = 9$ 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

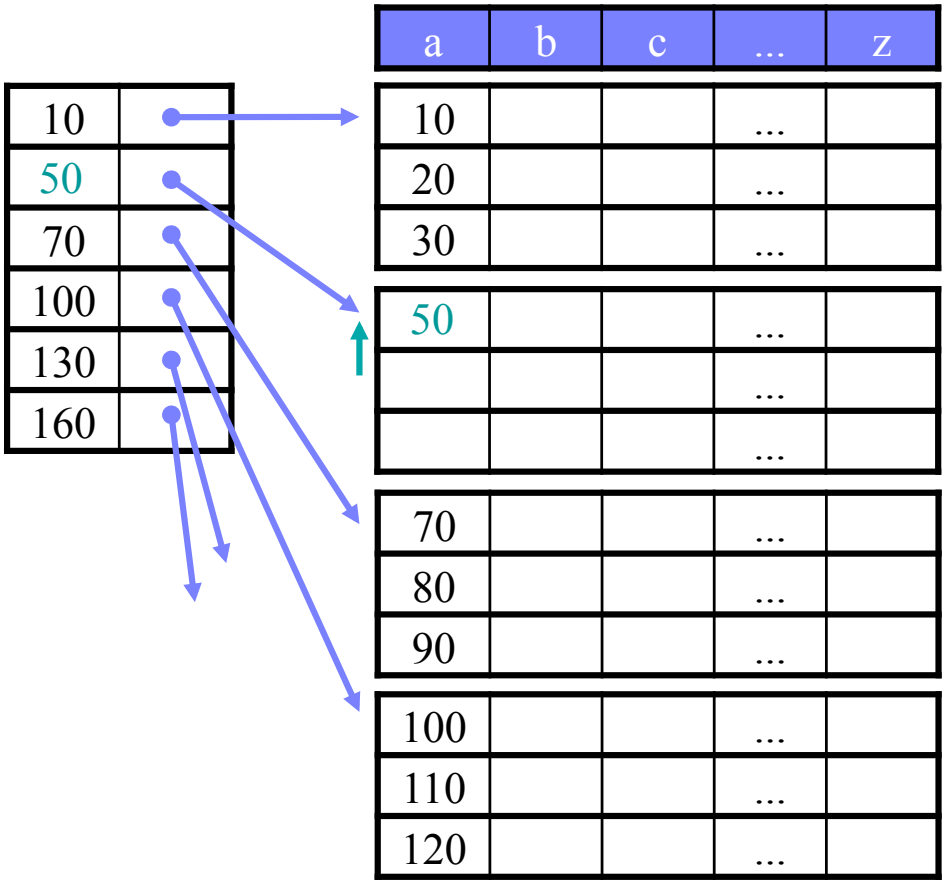
Flernivåindekser

- En indeks kan oppta flere blokker.
- En flernivåindeks, dvs. en indeks på indeksen, kan øke effektiviteten.
- Fortsettelse av regnestykket:
  - Trenger bare 1954 / 512 = 4 blokker for 2. nivå
  - ⌈log2(4)⌉ + 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.



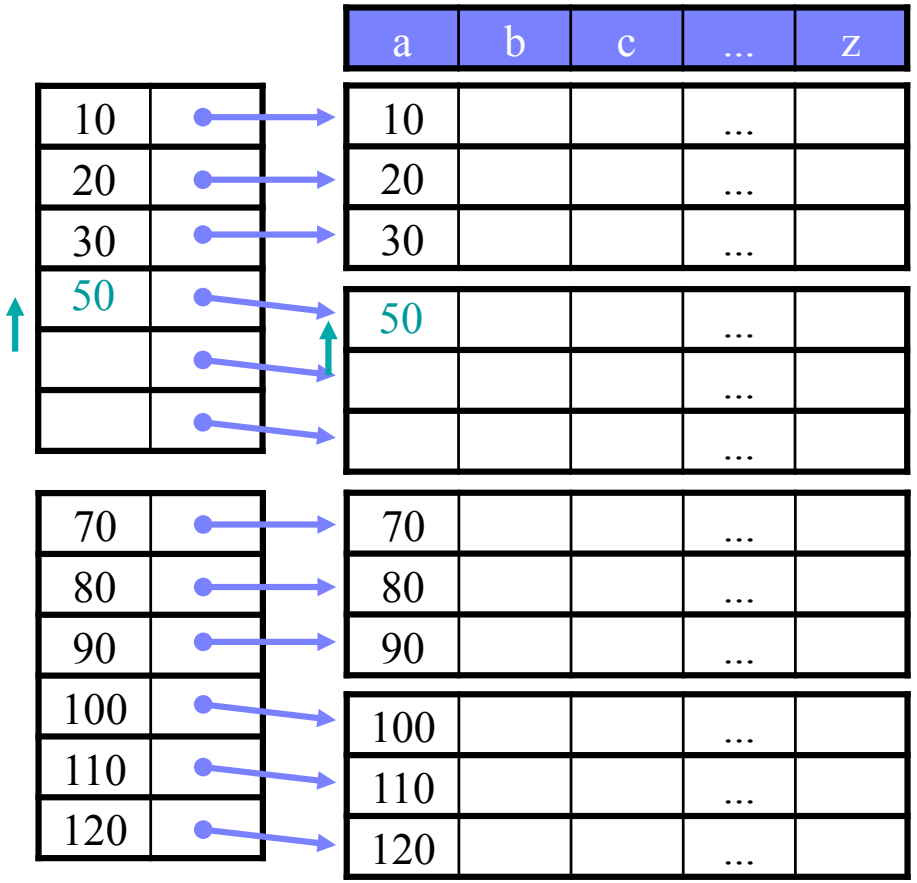
Eksempel: Sletting ved tynn indeks

- 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.



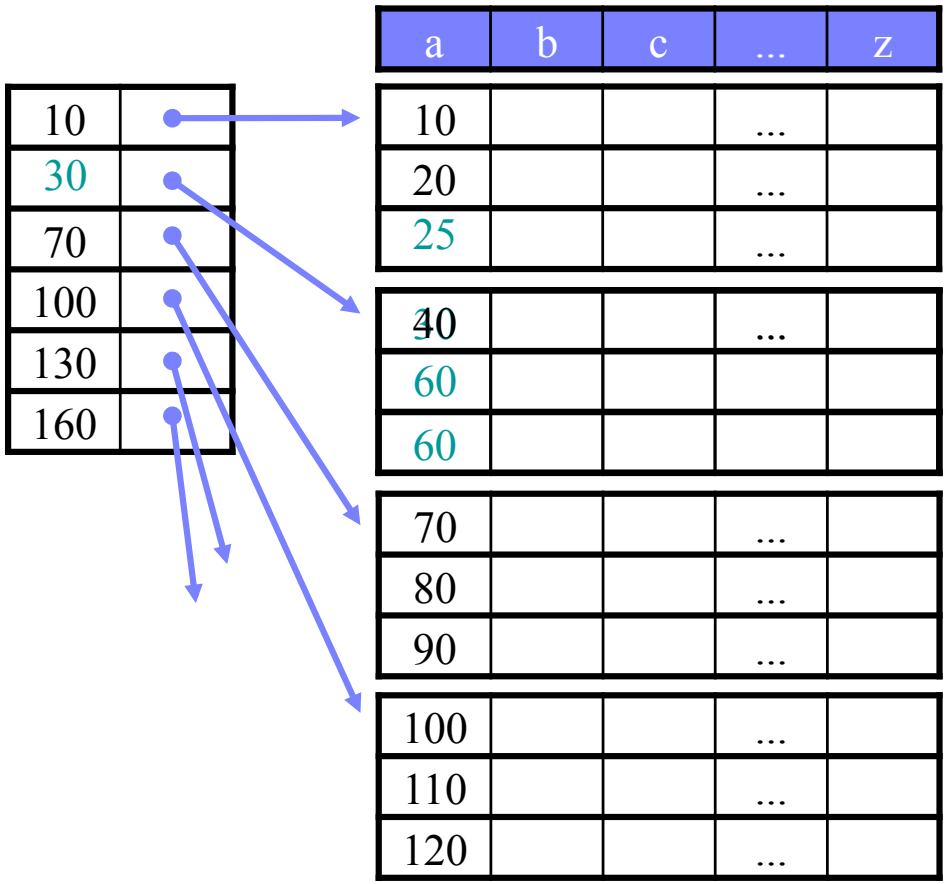
Eksempel: Sletting ved tett indeks

- 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.



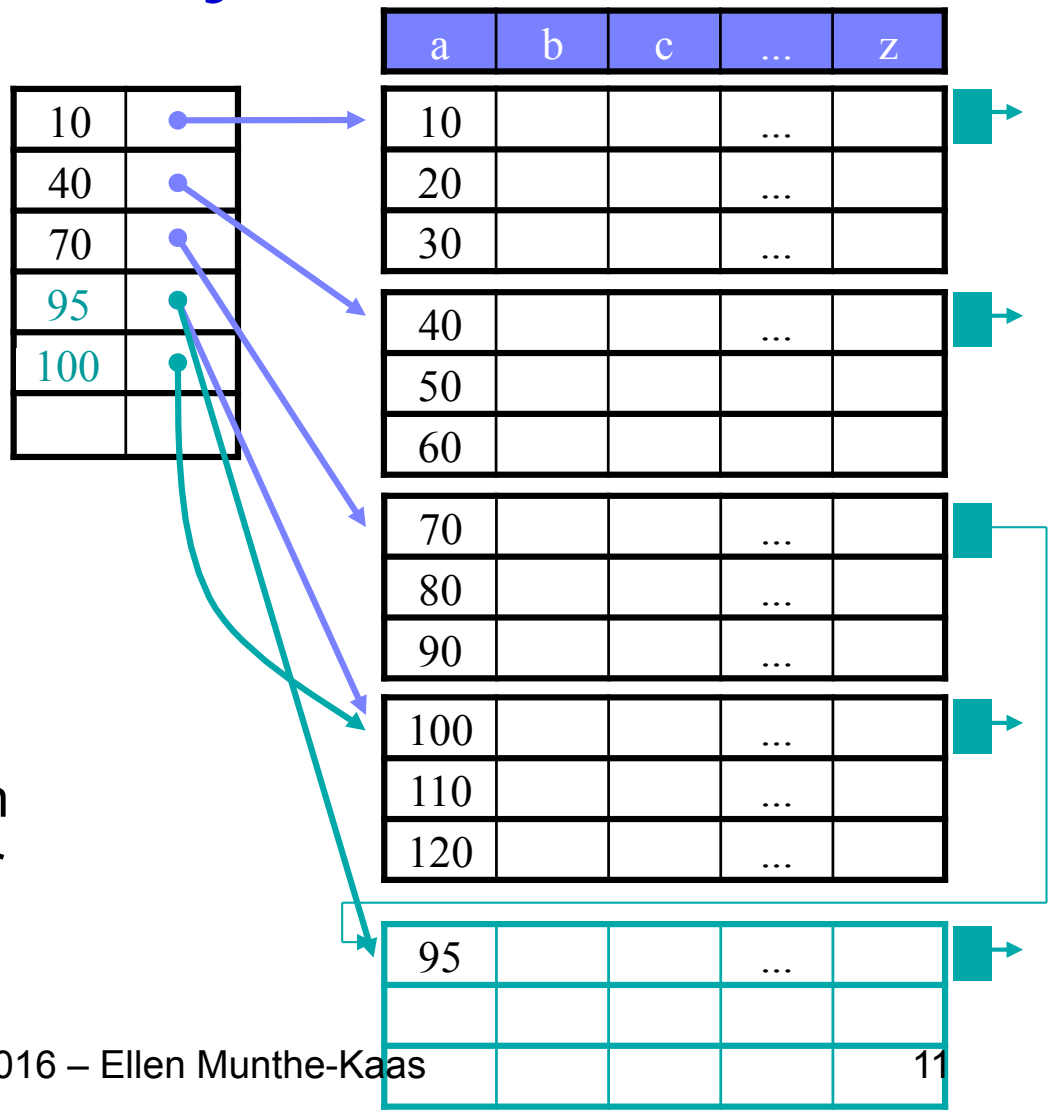
Eksempel: Innsetting ved tynn indeks

- 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.



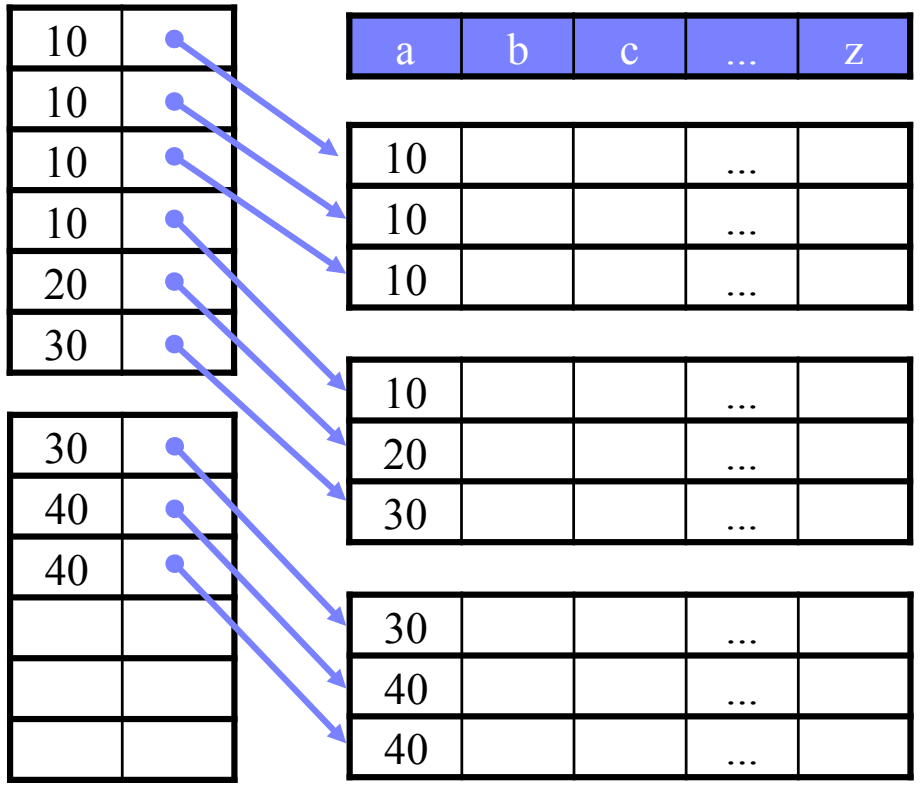
Eksempel: Innsetting ved tynn indeks

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



Clusterindekser - duplikate søkenøkler

- 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?

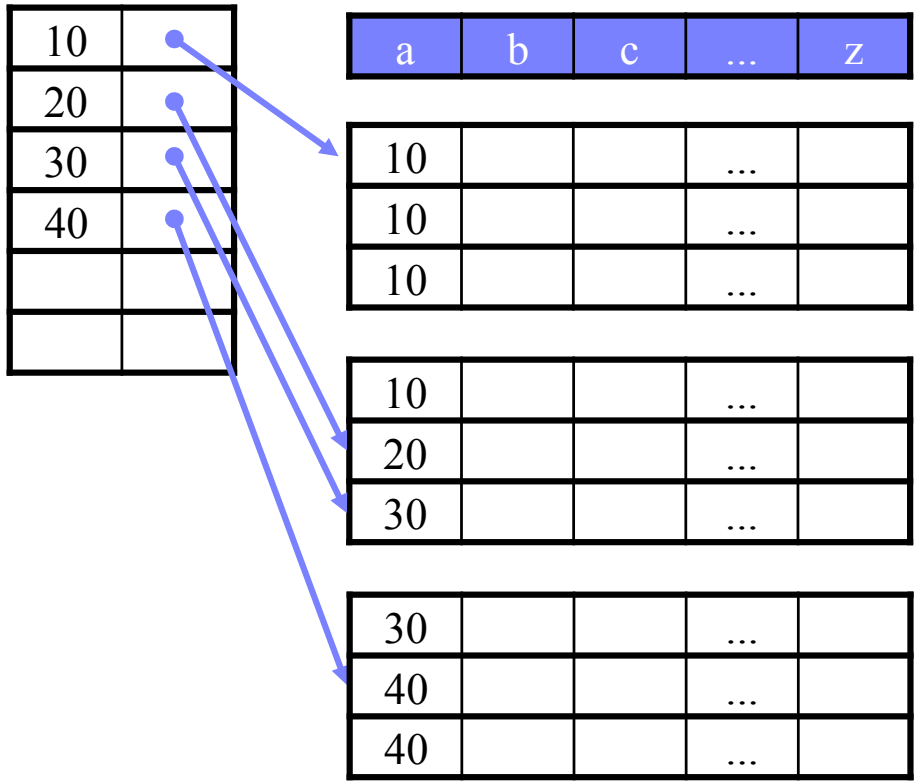


Clusterindekser: Eksempel – tett indeks

- Bare ett indeksfelt pr. unike søkenøkkel.

  ☺ Mindre indeks – raskt søk.

  ☹ Mer komplisert å finne påfølgende poster.

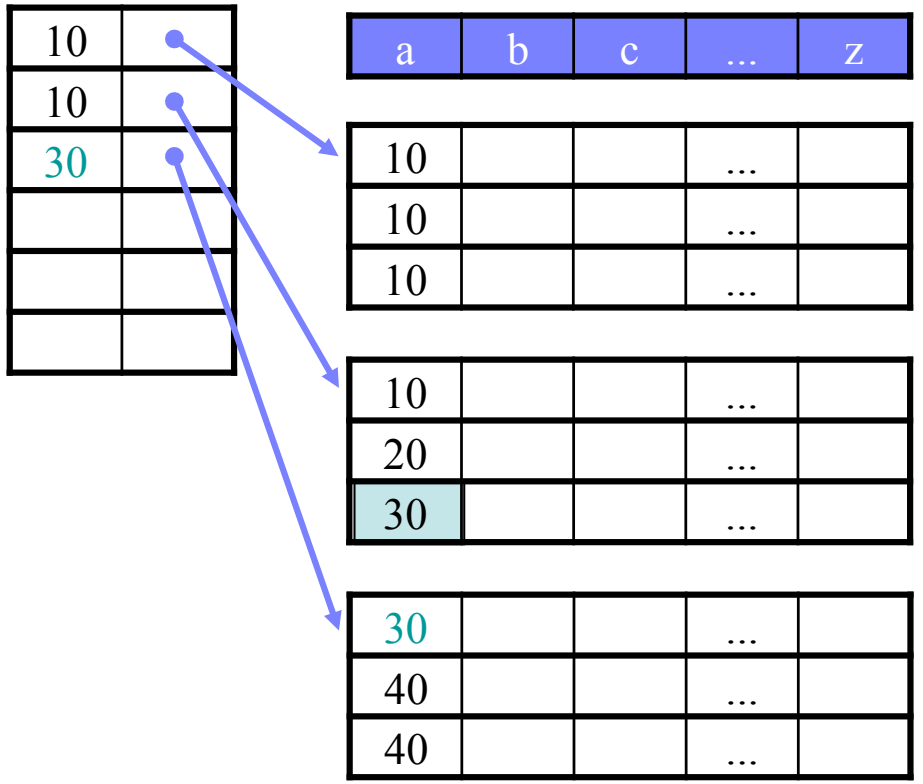


Clusterindekser: Eksempel – tynn indeks

- Indeksfeltene peker til første post i hver blokk

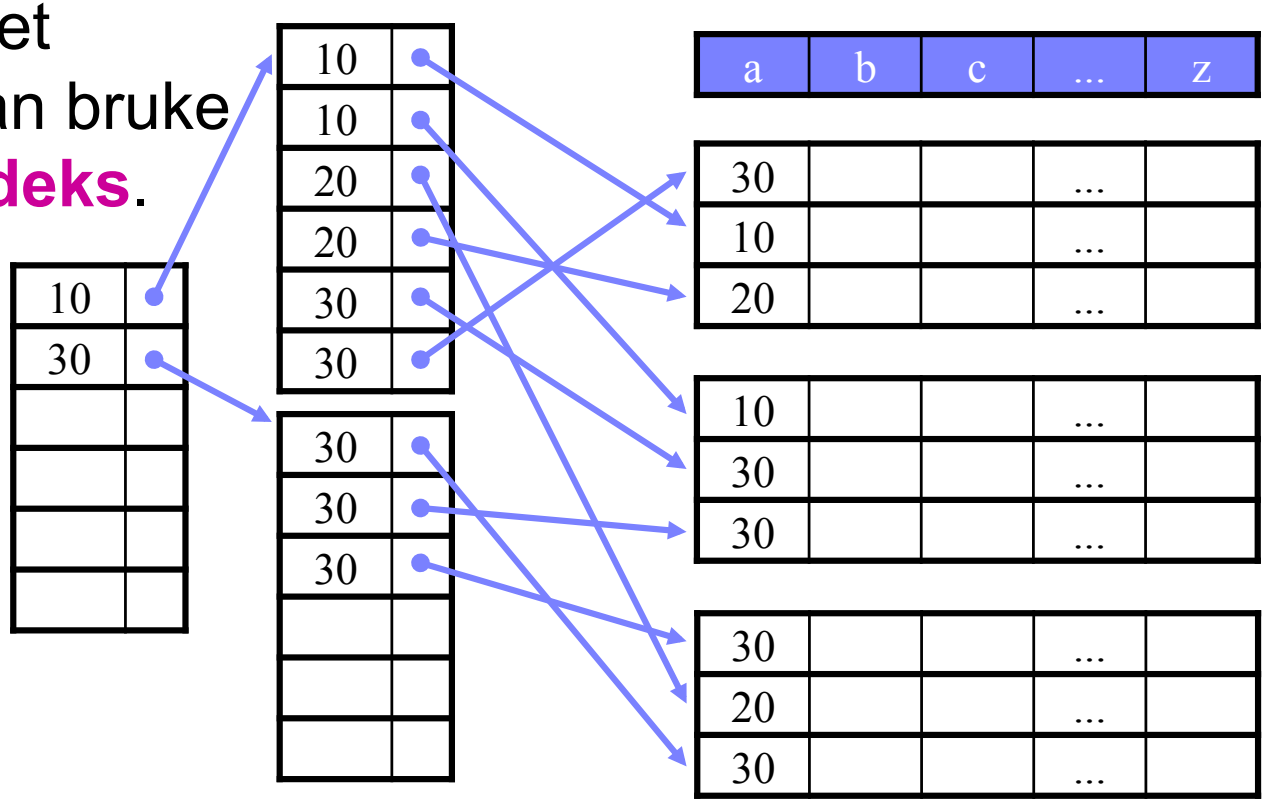
  ☺ Liten indeks – raskt søk

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



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.



Tette versus tynne indekser

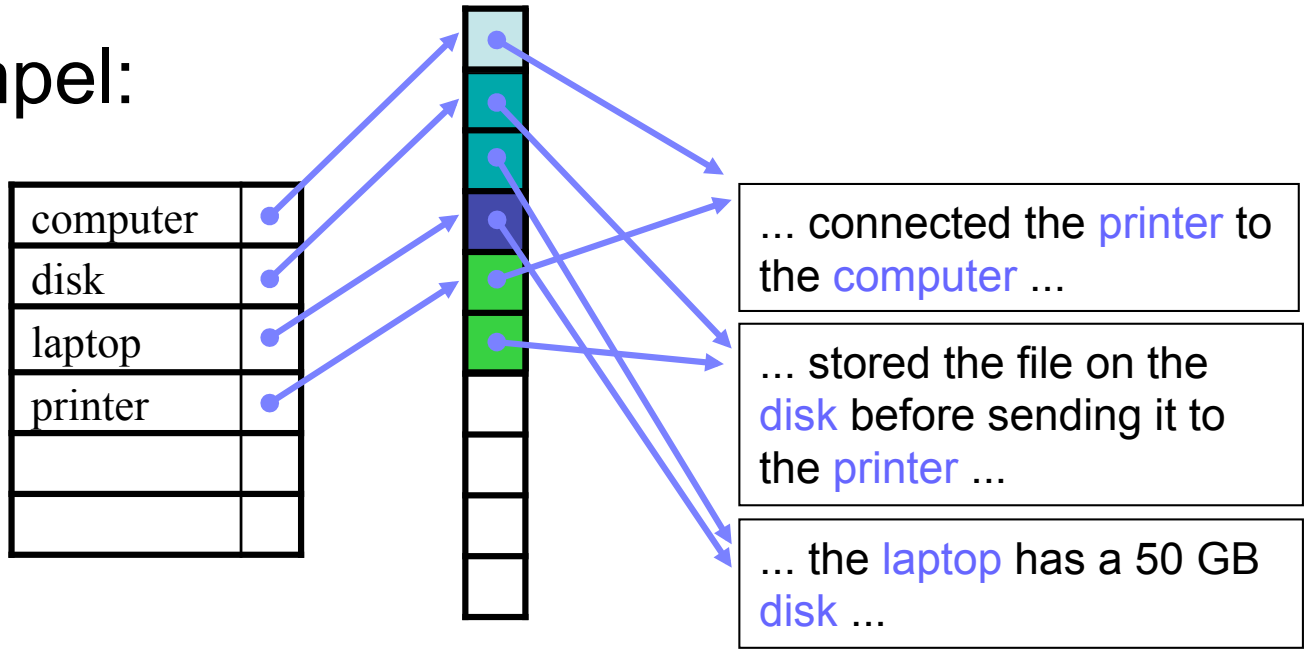
	tett	**tynn**
plass	ett indeksfelt pr. post	ett indeksfelt pr. datablokk
blokkaksesser	“mange”	“få”
postaksesser	direkte aksess	må lete innenfor datablokken
exists-spørringer	bruk indeksen alene	må alltid aksessere datablokken
bruk	overalt	ikke på usorterte elementer
endringer	må alltid oppdateres hvis postrekkefølgen endres	oppdateres bare hvis den første posten i en datablokk endres

Inverterte indekser

- Hva hvis man ønsker å søke på elementer innenfor ett 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.

Invertert indeks

- Eksempel:

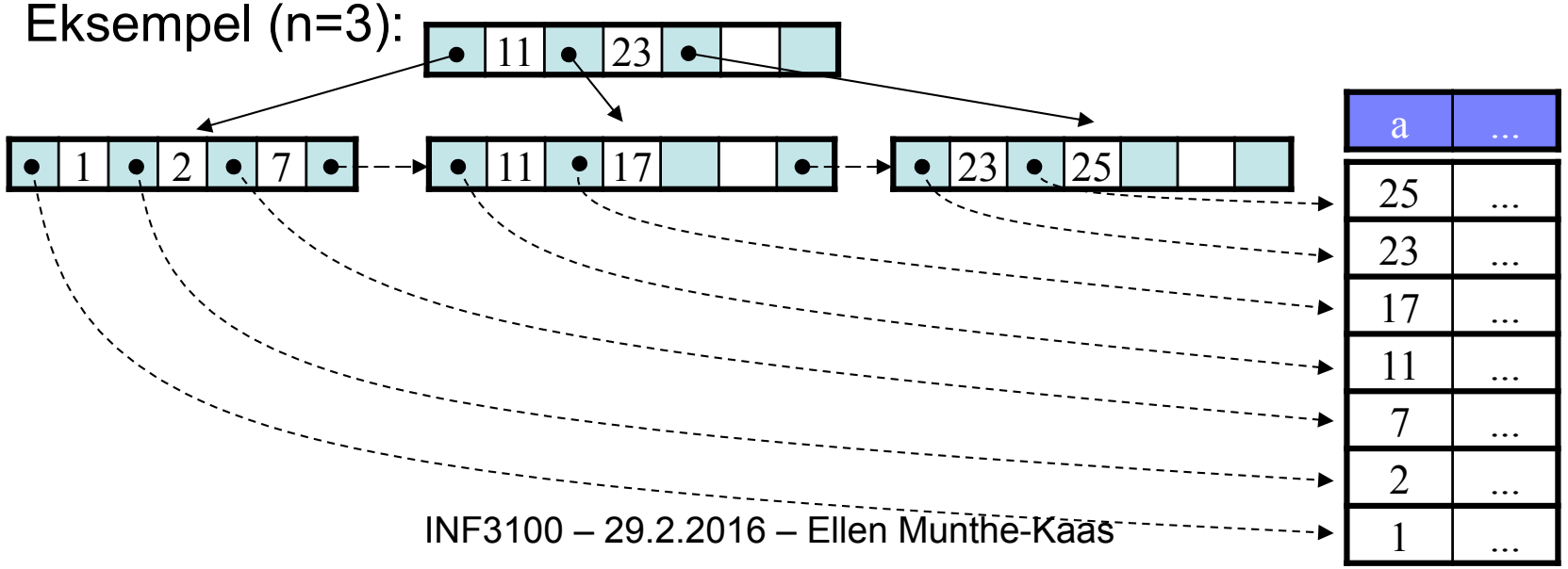


Navnet **invertert indeks** kommer fra følgende:

- **Direkte indeks**: Poster på formen (id1: dokument1), (id2: dokument2), ...
  (slå opp på en id, får tilgang til dokumentet)
- **Invertert indeks**: Poster på formen (computer: [id1]), (disk: [id2, id3]), ...
  (slå opp på et nøkkelord, får tilgang til alle relevante dokument-id'er)

B^+-trær

- Nodene er blokker. Alle løvnoder er på samme nivå. Hver node har *n* søkenøkler og *n+1* pekere.
  - Indre node: alle pekere er til subnoder.
  - Løvnode: n datapekere og 1 nestepeker.
- Alle noder må inneholde en viss mengde søkenøkler/pekere
  - Indre node: minst $\lceil (n+1)/2 \rceil$ pekere til subnoder.
  - Løvnode: minst $\lfloor (n+1)/2 \rfloor$ datapekere.
- Eksempel (n=3):

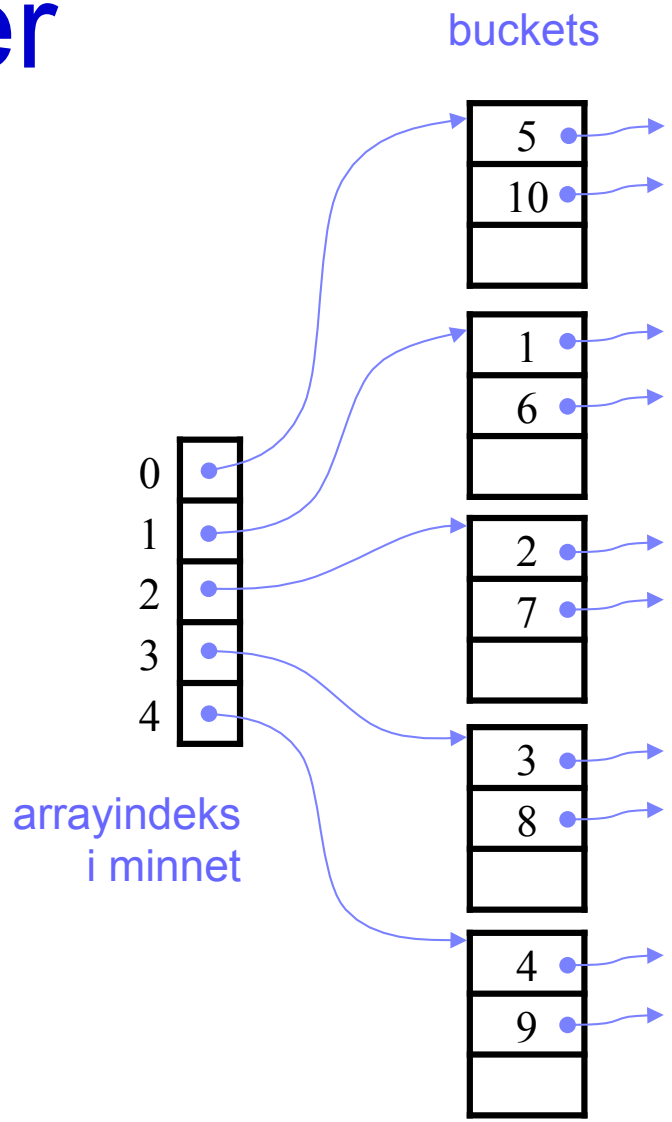


B^+-trær: effektivitet

- 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 (i datafilen).
  - ☺ Antall nivåer er vanligvis svært lavt – typisk 3.
  - ☺ Intervallsøking går raskt.
  - ☺ Ved stor *n* er det sjelden nødvendig å splitte/slå sammen noder.
  - ☺ Disk I/O kan reduseres ved å holde noen av indeksblokkene i minnet.
- Eksempel: 4B søkenøkler, 8B pekere, 4KB blokker (ingen headere)
  - Hvor mange verdier kan lagres i hver node?
    $4n + 8(n+1) \leq 4096$ ⇨ $n = \mathbf{340}$
  - Nodene er i snitt 75% fulle. Hvor mange poster kan et B^+-tre med 3 nivåer inneholde?
    $(340 * 75\ \%)^3 = 16581375 \approx$ **16,6 millioner** poster

Hashtabeller

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



Hashtabeller: effektivitet

- Ideelt er arraystørrelsen stor nok til at alle elementene for én hashverdi passer i én 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 for én hashverdi innenfor en bucketblokk hvis antall poster øker mens hashtabellen er statisk
- *Dynamiske* hashtabeller tillater arraystørrelsen å variere slik at det holder med én blokk pr. bucket.
  - utvidbar (extensible) hashing
  - lineær (linear) hashing

Sekvensielle vs hashindekser

- *Sekvensielle indekser* som f.eks. B-trær er gode på intervallsøk:
  select * **from** R **where** a > 5

- *Hashindekser* er gode når det søkes etter en spesiell nøkkel:
  select * **from** R **where** a = 5

Indekser i SQL

- Syntaks (DBMS-avhengig):
  - **create index** name **on** relationName (attribute)
  - **create unique index** name **on** relationName (attribute)

  - **drop index** name

- Merk: Ikke alle DBMSer tillater at man angir
  - type indeks, f.eks. B-tre, hashing osv.
  - parametre som loadfaktor, hashstørrelse osv.

Spørringer med flere betingelser

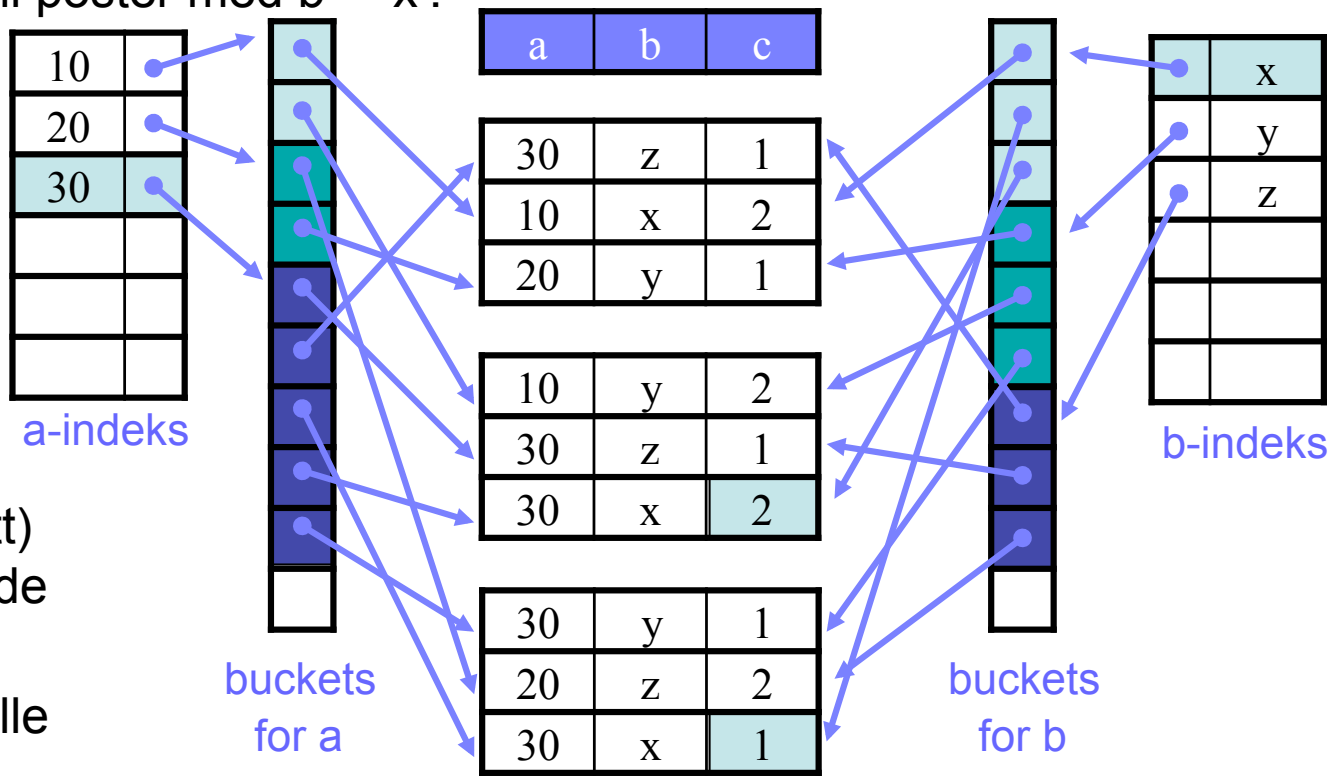
- **select** ... **from** R **where** a = 30 **and** b < 5
- Strategi 1:
  - Bruk en indeks, f.eks. på a.
  - Finn og *hent* alle postene med a = 30.
  - Søk gjennom disse postene for å finne poster med b < 5.

☺ Enkel strategi.

☹ Risikerer å lese mange unødvendige poster fra disk.

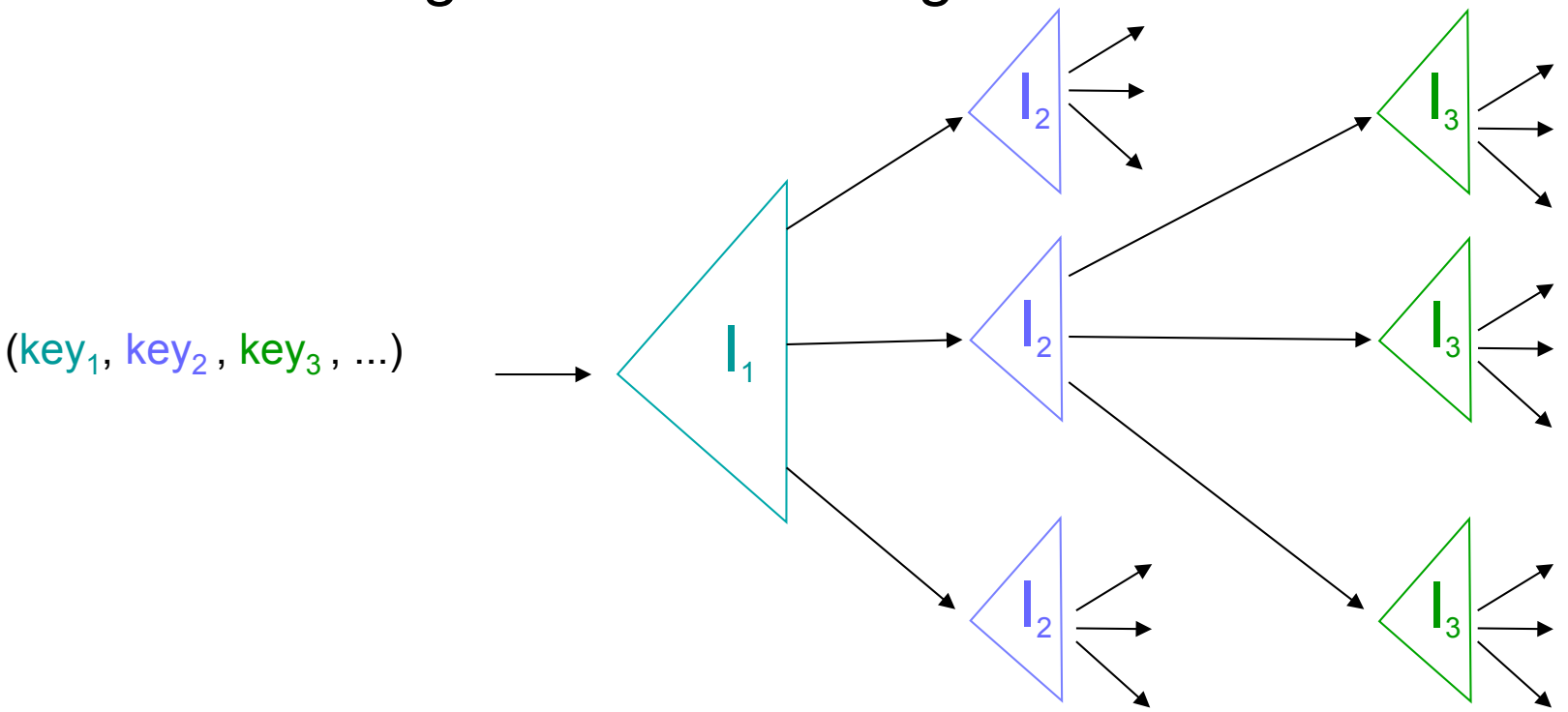
Flere betingelser: strategi 2

- **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.



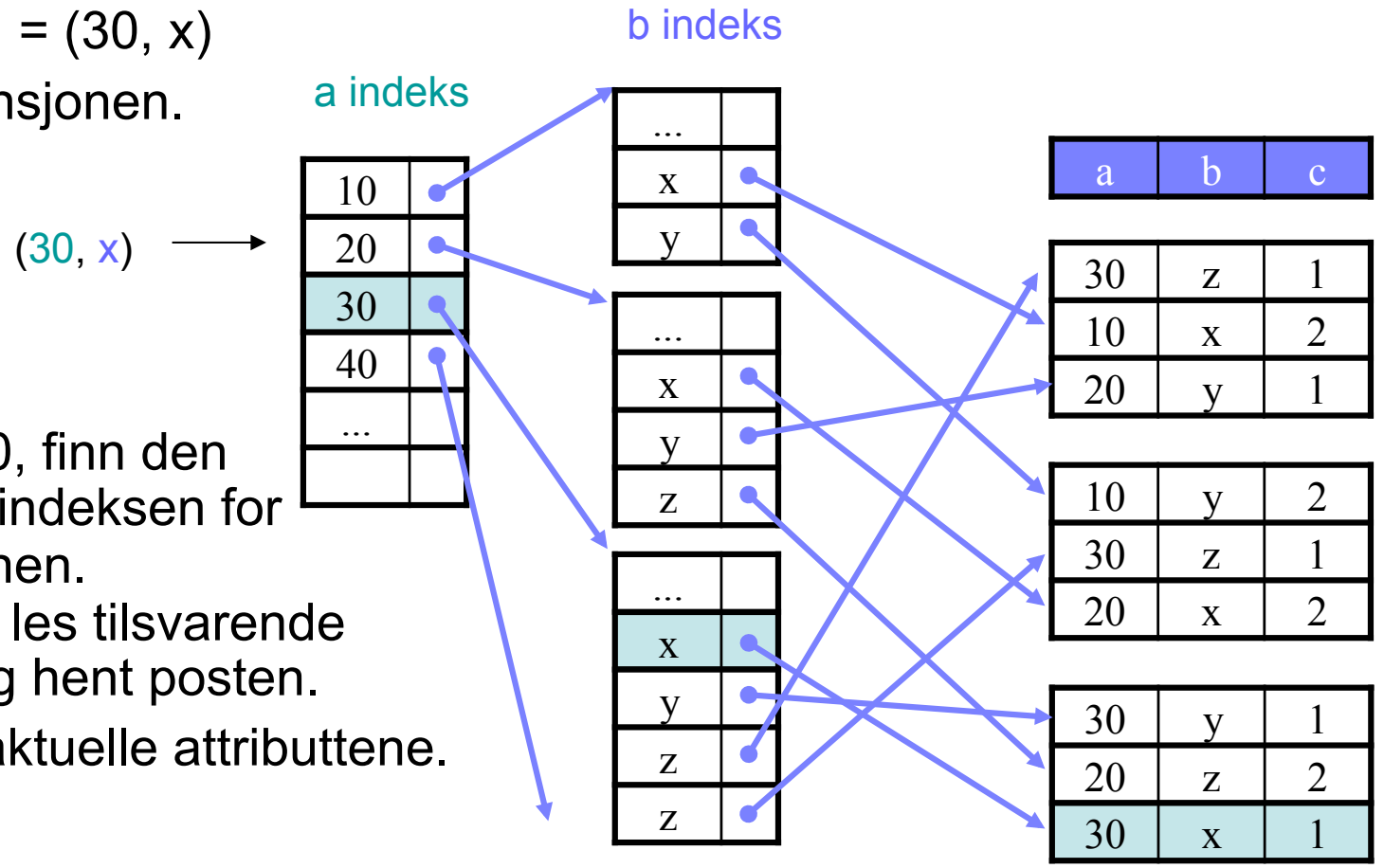
Flerdimensjonale indekser

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



Flerdimensjonale indekser: eksempel

- Eksempel, tett indeks:
  select c **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?

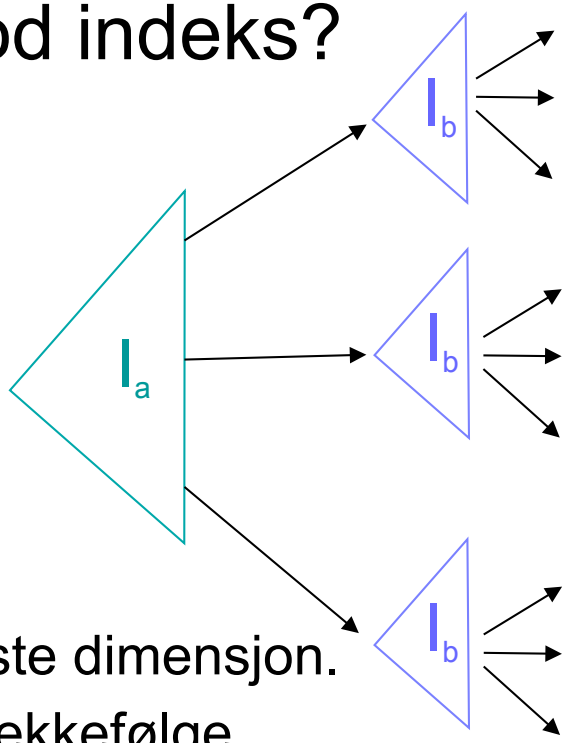
☺ Finn poster med a = 10 **and** b = ′x′

☺ 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′

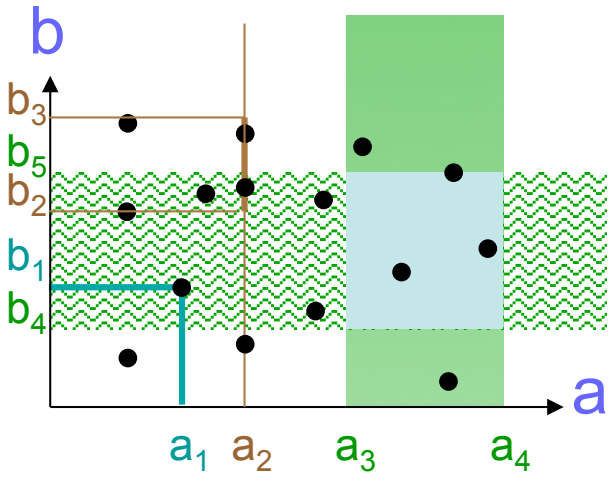


  - Risikerer å måtte søke i mange indekser i neste dimensjon.
  - Hadde vært bedre om dimensjonene endret rekkefølge.

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

Map View

- Vi kan betrakte en flerdimensjonal indeks med to dimensjoner som et geografisk kart:

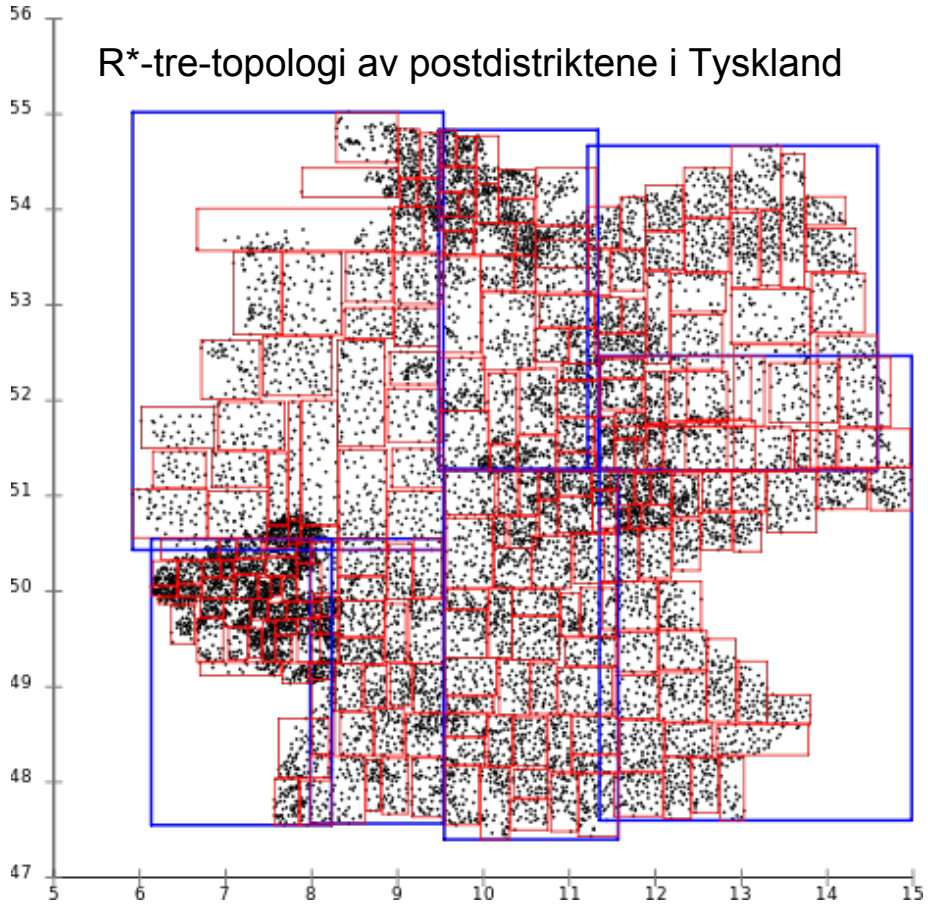


- Søking tilsvarer da å søke i kartet etter
  - punkter: a_1 og b_1
  - linjer: a_2 og $<b_2,b_3>$
  - arealer: $<a_3 ,a_4>$ og $<b_4,b_5>$

Trestrukturer

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

R*-trær



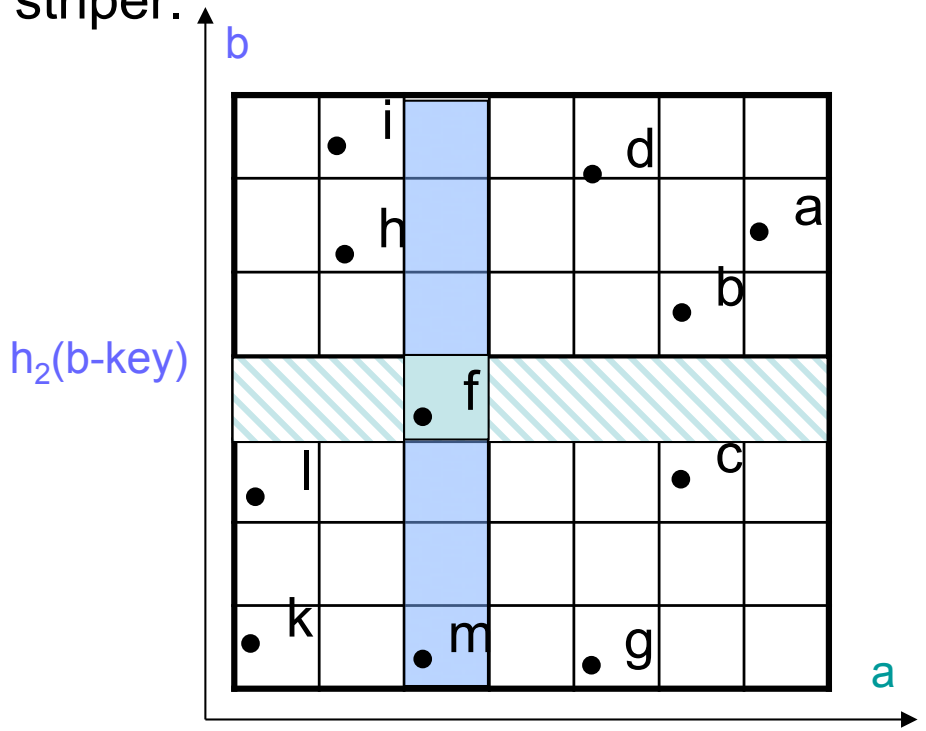
- Idé: Grupper geometriske objekter som ligger nær hverandre.
- Nodene er blokker. Alle løvnoder er på samme nivå.
  - Indre node: Det minste rektangelet som dekker alle objektene i subtreet
  - Løvnode: Et objekt
  - Alle indre noder må inneholde et visst antall pekere
- Søk (snitt, inneholdt i, nærmeste nabo) er enkelt
- Innsetting er utfordrende
  - Treet skal holdes fullstendig balansert
  - Rektangler bør ikke inneholde for mye tomrom
  - Rektangler med felles foreldernode bør ikke overlappe for mye
  - Kan måtte slette og gjeninnsette objekter for å få til bedre innplassering

Hash-lignende strukturer: gridfiler

- Gridfiler utvider tradisjonelle hashindekser til flere dimensjoner
  - Hasher verdier for hvert attributt i en flerdimensjonal indeks.
  - Hasher vanligvis ikke til *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
  - key1 = V_i **and** key2 = X_j
  - key1 = V_i
  - key2 = X_j
  - key1 >= V_i **and** key2 < X_j
- Gridfiler:
  - ☺ Bra for søking ved flere nøkler.
  - ☹ Bruker mye plass, krever en del organisering.

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 selv om recorden slettes
- 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	H
1	30	foo	65
2	30	bar	43
3	40	baz	84
4	50	fou	43
5	40	bar	65
6	30	baz	65

bitvektorer for F

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

bitvektorer for G

bar:	0	1	0	0	1	0
baz:	0	0	1	0	0	1
foo:	1	0	0	0	0	0
fou:	0	0	0	1	0	0

Karakteristika bitmapindekser

- 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