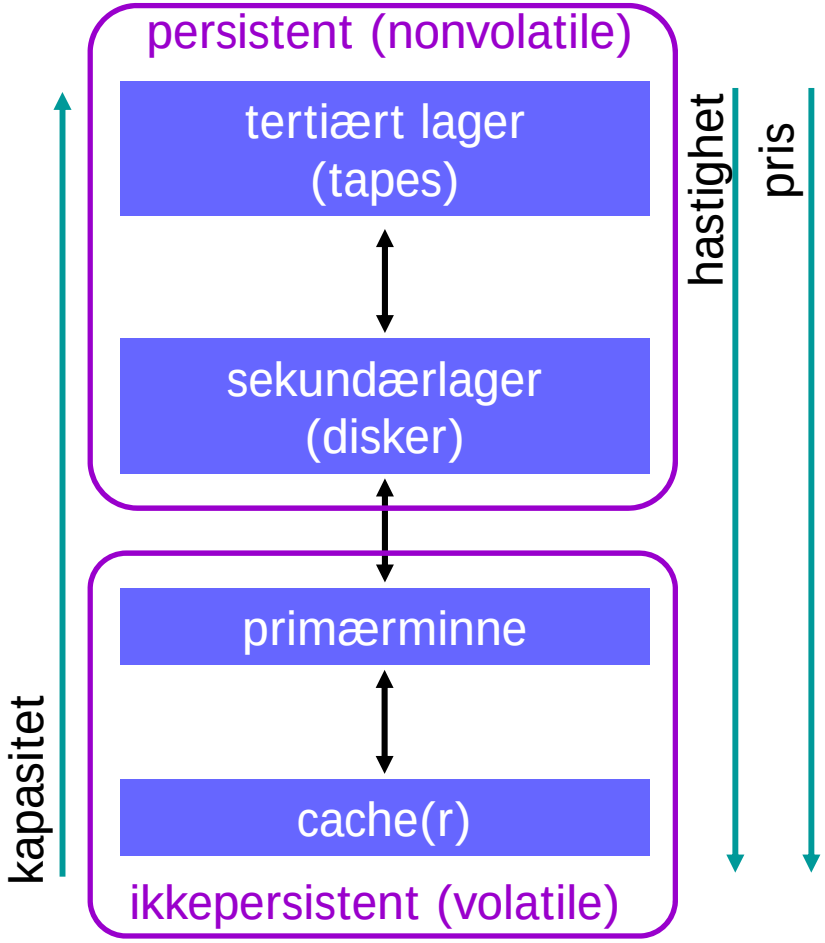
UNIVERSITETET
I OSLO

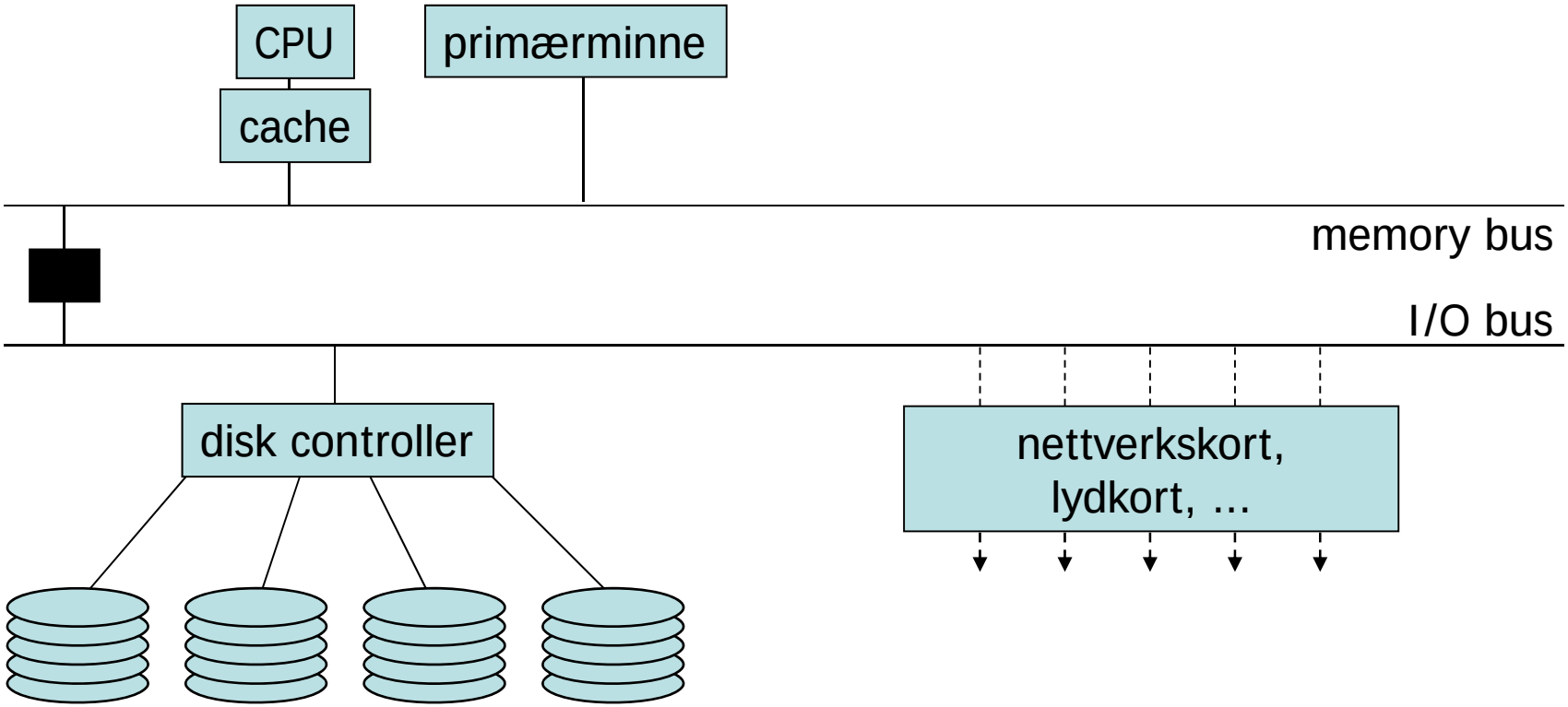
Effektiv bruk av sekundærlager

Minnehierarkiet

- Vi kan ikke aksessere disken hver gang vi trenger data (det tar tid!)
- Store datamaskinsystemer har derfor flere forskjellige komponenter hvor data kan lagres med
  - ulik kapasitet
  - ulik hastighet
- Komponenter med mindre kapasitet har typisk lavere aksesstid og høyere kostnad pr. byte
- Høyere nivåer tjener som sikkerhets-kopi av data på lavere nivåer
- Det laveste nivået ligger nærmest CPUen



Hovedarkitekturen



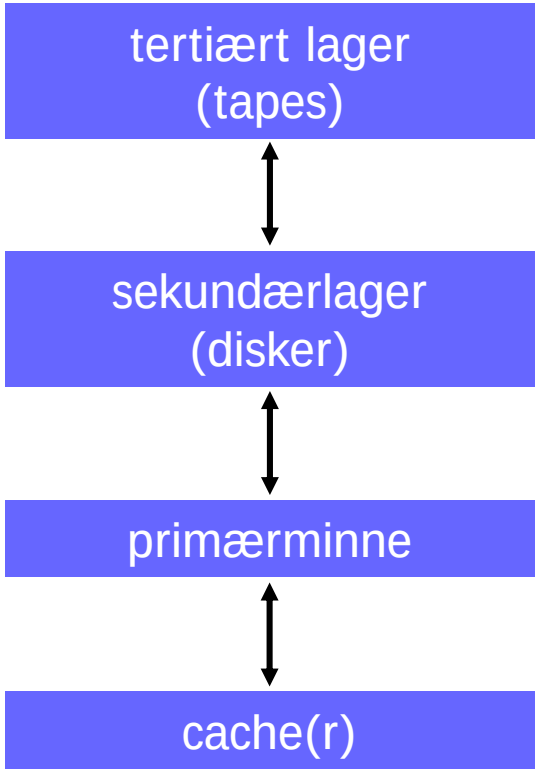
1 kilobyte	= 1Kb	= 2^{10} bytes	= 1024 bytes	≈ 10^3 bytes
1 megabyte	= 1Mb	= 2^{20} bytes		≈ 10^6 bytes
1 gigabyte	= 1Gb	= 2^{30} bytes		≈ 10^9 bytes
1 terabyte	= 1Tb	= 2^{40} bytes		≈ 10^{12} bytes
1 petabyte	= 1Pb	= 2^{50} bytes		≈ 10^{15} bytes

Dataflyt i minnehierarkiet

- Hvis ønskede data ikke finnes på et nivå, må vi se etter dem på nivået over (langsommere, men med mer kapasitet)
- Hvis vi henter data fra et høyere nivå, må vi kanskje overskrive data som allerede finnes på inneværende nivå
- Det finnes mange algoritmer/strategier for å velge et passende offer for utskiftning
- Hvis vi endrer data, må vi også endre dataene på de høyere nivåene for å opprettholde en konsistent database
- Her er det to hovedstrategier:
  - *Delayed write*
    Data på høyere nivåer oppdateres «når det passer»
  - *Write-through*
    Data på høyere nivåer oppdateres umiddelbart; andre operasjoner må vente på oppdateringsoperasjonen

Cache og primærminne

- **Cacher:**
  - Ofte to nivåer:
    - on-board cache: på CPU-chipen (32 eller 64 Kb)
    - level-2 cache: på en annen chip (1024Kb)
  - Typisk aksesstid noen få ns (10^{-9} sekunder)
  - Overføringsenhet cache/primærminne: Cachelinje, 32 bytes
  - Hvis hver CPU har sin egen cache i et multiprosessorsystem, må vi bruke *write-through*
- **Primærminnet** (noen hundre Mb til flere Gb):
  - Primærminnet er *random access*
  - Typiske aksesstider fra 10 til 100 ns (10^{-8}-10^{-7} sekunder)
  - Overføringsenhet primærminne/sekundærlager: Datablokk/side (page)
  - Memorymanageren – minnehåndtereren – bruker vanligvis *virtuelt minne*



Virtuelt minne

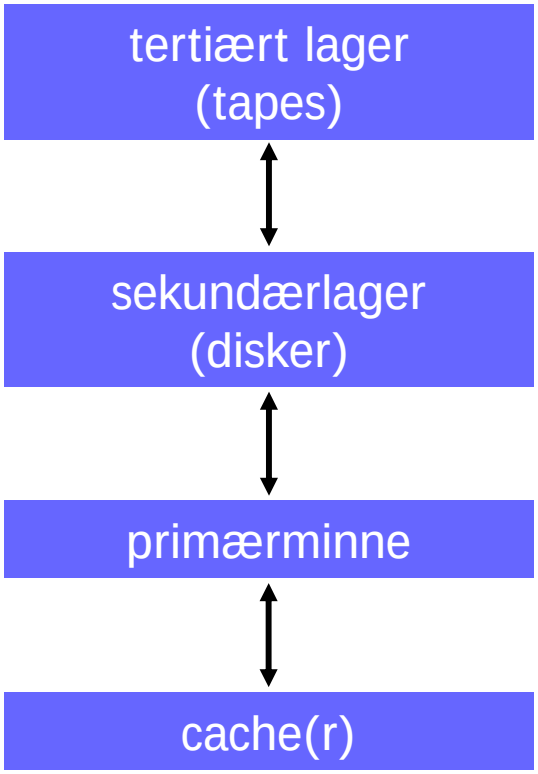
- Virtuelt minne lar prosessene bruke mer minne enn det som er fysisk tilgjengelig
- Typisk vil prosessene få lov til å bruke 32-bits virtuelle byteadresser, som gir et adresserom på 4Gb, noe som er mye mer enn det fysiske minnet hver enkelt prosess får lov til å bruke
- Operativsystemet holder orden på hvilke data som er i fysisk minne og hvilke som ligger på disk
- Denne teknikken kalles *paging*
- Paging er detaljert behandlet i operativsystemkurset INF3151 og vil ikke bli nærmere gjennomgått her

Minnehåndtering og DBSer

- Vanligvis prøver vi å holde databasen i primærminnet for å øke ytelsen
- Vi vil gjerne unngå å bruke *write-through*-strategien mellom primærminne og disk, men siden endringer i databasen skal være persistente, må alle oppdateringer skrives til disk og logges
- Databaser som er så små at de får plass i tilgjengelig virtuelt minne, kan, men trenger ikke, bruke operativsystemets minnehåndtering
- Større databaser må håndtere sine data direkte på disk, og et slikt DBMS må derfor ha sin egen minnehåndterer

Sekundær- og tertiærlager

- **Sekundærlager** (opptil noen få Tb):
  - Den viktigste typen er magnetisk harddisk; typisk aksesstid 10ms, så disker er rene sinker (ca. 10^6 ganger langsommere) i forhold til primærminnet
  - Brukes som persisent lager for data og som lager for programmets virtuelle minne
  - Ofte håndterer DBMS selv I/O mot disk
    - Oppgavene og problemene er i hovedsak de samme som i et ordinært filsystem
- **Tertiærlager** (flere Pb):
  - Brukes når systemet er for stort til å kunne lagres på disker
  - Aksesstider på flere sekunder, noen på flere minutter, dvs. minst 10^3 ganger langsommere enn disker
  - Det finnes mange typer, bl.a. ad-hoc tapelager, optiske jukebokser, tapesiloer
    - Tertiære lagre er ikke noe sentralt tema i dette kurset

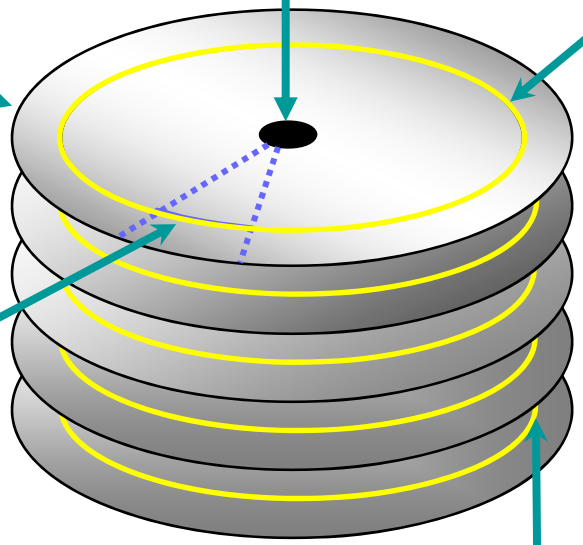


Disker

- Selv om det nå finnes mange persistente media som kan brukes som sekundærlager, er disker nær enerådende for større systemer (også for store databaser)
- På grunn av den store hastighetsforskjellen mellom disk og primærminne (en faktor på ca. 10^6) er det viktig å minimalisere antall diskaksesser
- Sekundærlageret er laveste nivå av persistente data

  Det er derfor viktig at ikke diskfeil fører til tap av data (Overføring til tertiærlager gjøres vanligvis med *delayed write*, slik at tertiærlageret ikke alltid er oppdatert)
- I vår håndtering av disker har vi to hovedoppgaver:
  - Sikring mot feil
  - Effektiv bruk

Fysiske disker (terminologi)



Platters
Sirkulære plater dekket med magnetiserbart materiale, sørger for bestandig (non-volatile) lagring av bits; begge platesidene kan være magnetisert

Spindle
som platene spinner rundt

Tracks (spor)
Konsentriske sirkler på hver enkelt plate

Disk heads
Leser eller endrer magnetismen på bits som passerer under hodet. Hodene er festet på en arm som kan forflytte hodene radialt langs plateoverflatene

Sectors
Segmenter av et spor, adskilt av ikke-magnetiserte *gaps*
Gaps brukes til å identifisere begynnelsen på en sektor

Cylinders
Samhørende spor på de enkelte platene sies å danne en sylinder

Kortfattet diskteknologihistorie

- Første disk var IBM350, introdusert 4.9.1956, med kapasitet 4,4 Mbyte, hastighet 1200 RPM, aksesstid nesten ett sekund, vekt 1,2 tonn
- Winchesterteknologi (disk og diskarm i samme støvtette boks) kom i 1973 (IBM3340; vaskemaskinstørrelse)
- De første disker til PC-er kom i 1981 (5 Mb)
- To teknologier for diskkommunikasjon skilte seg ut:
  - SCSI («scussi») – Small Computer System Interface
    Eksempel: Seagates Barracuda- og Cheetah-disker
  - ATA – Advanced Technology Attachment
    Eksempel: IBMs disker

«State of the Art» diskteknologi ved årsskiftet 2007/2008

- Hitachi (tidligere IBM) leverer 1 Tb disk med SATA (Serial ATA) til bordmaskiner
- Seagate har gått over til SAS (Serial Attached SCSI)
- Det kan se ut til at SATA har et forsprang i konkurransen med SAS
- UiO/USIT satser på kabinetter med fjorten 500 Mb SATA-disker organisert som RAID 10
  (Vi kommer tilbake til RAID-teknologien)

  Kabinettene er koblet sammmen i et Fiber Channel SAN (Storage Area Network)

Diskkapasitet

- Diskstørrelsen er avhengig av
  - Antall plater, og om platene brukes på begge sider
  - Antall spor pr. overflate
  - Gjennomsnittlig antall sektorer pr. spor
    (moderne disker er *sonet*; dvs. at de ytre sporene har flere sektorer enn de indre)
  - Antall bytes pr. sektor
- Det er forskjell på total og formatert kapasitet

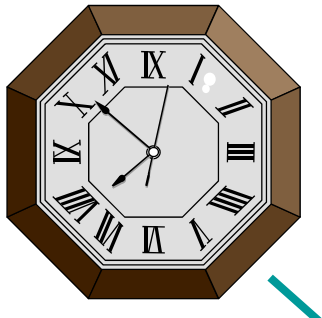
  Noe plass blir brukt til sjekksummer, reservespor o.l.

Aksessering av disk

- Hva må vi gjøre for å lese/skrive fra/til disk?
  - Plasser hodet over sylinderen (sporet) som inneholder datablokken vi vil lese eller skrive (den kan bestå av en eller flere sektorer)
  - Vent til første sektor i datablokken passerer under diskhodet
  - Les eller skriv datablokken mens sektorene passerer under diskhodet
- Tiden fra en prosess ber om å få lese en datablokk til blokken er i primærminnet, kalles diskens aksesstid

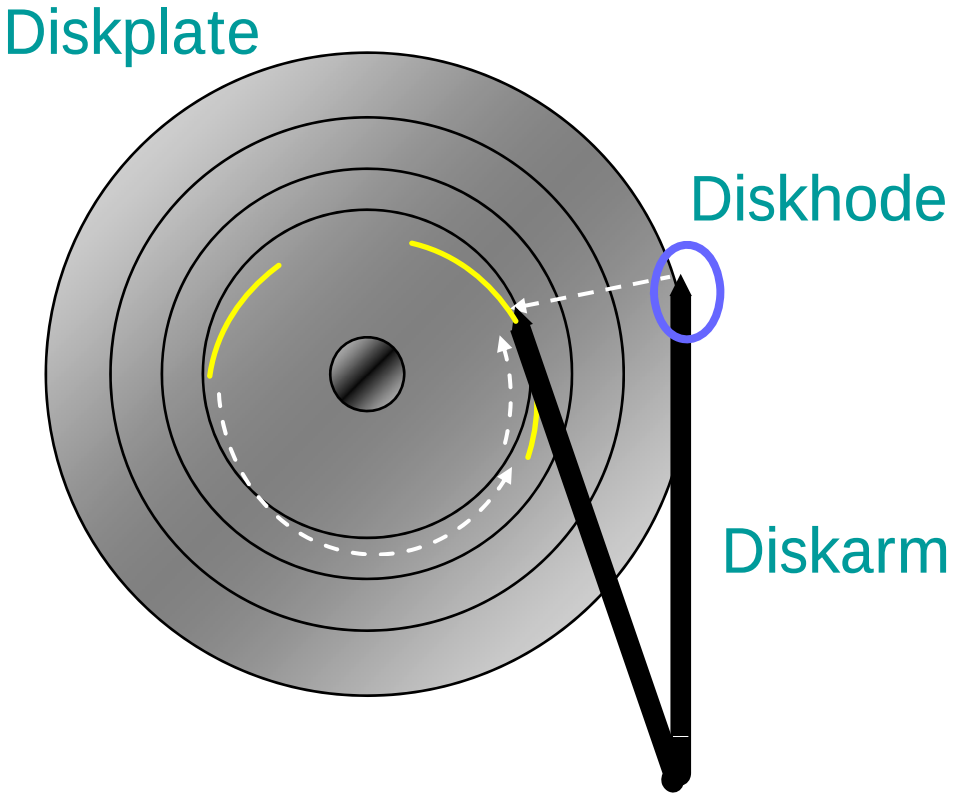
Diskers aksesstid – I

Jeg vil ha blokk X



blokk X i primærminnet

Total aksesstid =
søketid
+ rotasjonsforsinkelse
+ overføringstid
+ andre forsinkelser



Diskers aksesstid – II

scheduling latency

disk latency

- **Søketid** (**seek time**): Tiden det tar å flytte diskhodet til rett sylinder
  - Typiske søketider er 4 - 10 ms
- **Rotasjonsforsinkelse** (**rotational latency**): Tiden det tar for platene å rotere slik at første ønskede sektor er under diskhodet
  - Midlere forsinkelse er en halv omdreining
  - Typiske verdier er 2.00-8.33 ms (15000-3600 RPM)
- **Overføringstid** (**transfer time**): Tiden det tar for de ønskede data å passere under diskhodet
  - avhengig av datamengde, datatetthet og rotasjonshastighet – og om dataene strekker seg over flere spor (sylindre)
- **En rekke andre faktorer medfører tidsbruk**: CPU-tid for å be om og prosessere I/O; konkurranse om tilgang til diskkontroller, buss og minne; kontroll av at datablokker er korrekt overført (med mulige retransmisjoner), ...
  - Felles for alle disse er at de har neglisjerbar verdi sammenlignet med de øvrige faktorene (ns/µs versus ms)

Skriving og endring av blokker

- Skriveoperasjoner er analoge med leseoperasjoner
- En komplikasjon oppstår hvis skriveoperasjonen må verifiseres – vi må da vente en rotasjon og så lese og verifisere blokken
- Total skrivetid ≈ lesetid + tid for en rotasjon
- Blokker kan ikke endres direkte på disk:
  - Les blokken inn i primærminnet
  - Modifiser blokken
  - Skriv det nye innholdet tilbake til disk
  - (Verifiser skriveoperasjonen)
- Total modifiseringstid ≈ lesetid + tid til endring + skrivetid (tiden brukt på selve endringen er neglisjerbar)

Diskkontrollere

- Diskkontrolleren er en liten prosessor som:
  - Flytter diskhodene til riktig sylinder
  - Velger plate og overflate som skal brukes
  - Vet når riktig sektor er under hodet
  - Overfører data mellom primærminnet og disk
- Nyere kontrollere er selv små datamaskiner
  - Både disk og kontroller har nå egen buffer som reduserer diskaksesstiden
  - Data på ødelagte diskblokker/sektorer blir flyttet til et reserveområde på disken – operativsystemet (OS) vet ikke om dette, så en blokk kan ligge et annet sted enn hva OS tror
  - Dermed blir OS' diskadresser virtuelle

Effektiv bruk av sekundærlager

- Siden bruken av sekundærlageret er totalt dominerende når det gjelder effektiviteten av et DBS, er det her det er lønnsomt å optimalisere

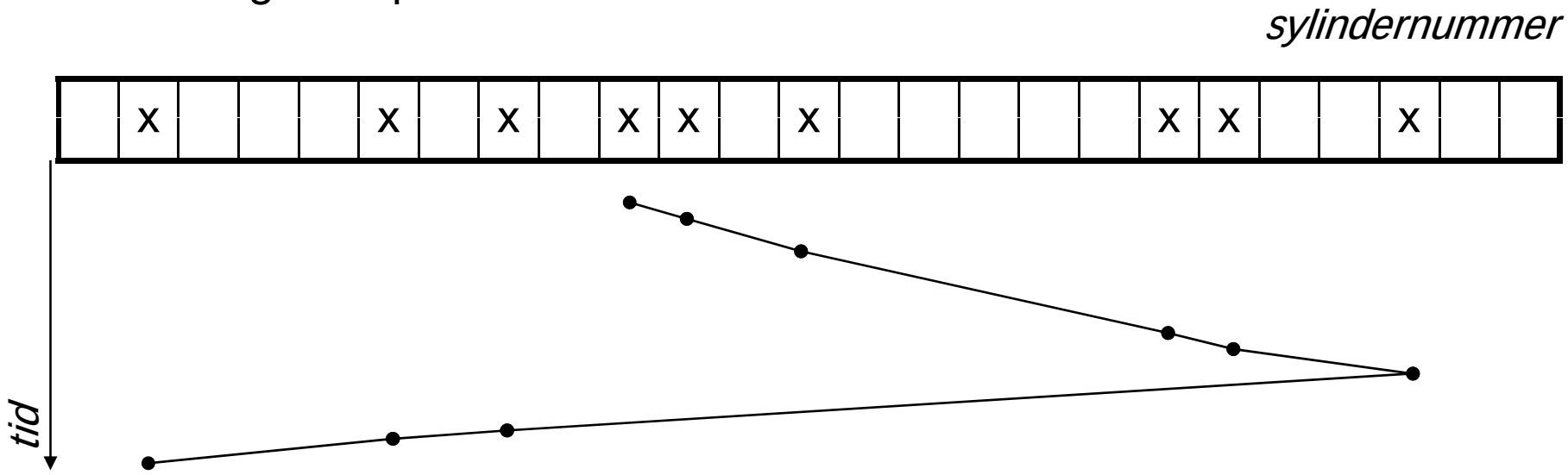
- Men det er mange måter å optimalisere på

Betydningen av blokkstørrelsen

- Forutsatt en tilfeldig plassering av blokkene på disken vil en dobling av blokkstørrelsen halvere antall diskaksesser
  - Total overføringstid blir den samme
  - Søketid og rotasjonsventetid blir halvert
- Men for store blokker er heller ikke bra
  - Hver blokk bør ligge innenfor ett spor (i hvert fall innen en sylinder)
  - Små dataelementer vil bare fylle brøkdelen av en blokk
- Valg av blokkstørrelse er avhengig av datastørrelse og aksessmønster
- Ny teknologi med større og raskere disker gjør at trenden er større blokkstørrelser

Søkestrategier

- Søketid er den dominerende faktor for total disk I/O tid
- Det finnes flere algoritmer, bl.a.:
  - First-Come-First-Serve
  - Shortest Seek First: Diskkontrolleren velger hvilken prosess den vil betjene avhengig av hodets posisjon og ønsket blokks posisjon (minimaliser hodebevegelsen)
  - Elevatoralgoritmen: La hodet gå fra innerste til ytterste sylinder og stopp hvis det passerer en ønsket blokk, snu når siste blokk i denne retningen er prosessert



Multippel bufring (prefetching)

- Hvis vi kan forutse aksessmønsteret på disken, kan vi øke effektiviteten ved å bruke flere bufre
- Det enkleste er dobbelbufring:
  - Les data inn i buffer nr 1
  - Prosesser data i buffer nr 1
    samtidig som data leses inn i buffer nr 2
  - Prosesser data i buffer nr 2
    samtidig som data leses inn i buffer nr 1
  - osv.

Beregningsmodeller

- **RAM-modellen** (RAM = Random Access Model):
  - Antar at alle data får plass i primærminnet
  - Tiden det tar å aksessere et data item er uavhengig av hva innholdet i itemet er eller hvor det ligger i minnet
- **I/O-modellen**:
  - Når ikke alle data kan få plass i primærminnet samtidig
  - Antall blokkaksesser (skriving/lesing) er det som avgjør hvor god en algoritme er
    (Tidsbruk i primærminnet er neglisjerbar)

Diskfeil – I

- Moderne disker har en MTF (mean time to failure) på over 50 år (da er ca. 50 % av diskene ødelagt), men
  - Mange disker feiler i løpet av noen måneder (produksjonsfeil)
  - Hvis det ikke er noen produksjonsfeil, vil disken trolig virke i mange år
  - Gamle disker har igjen en høyere risiko for feil pga. slitasje, støv o.l.

Diskfeil – II

Vi deler diskfeil inn i tre grupper:

- **Opprettbare** (intermittent failures)
  Midlertidige feil som kan rettes ved å lese blokken på nytt
  (f.eks at støv på disken gir en bit-feil)
- **Uopprettbare** (media decay/write failures)
  Permanente feil hvor bits er ødelagte
  (f.eks skader/sår i den magnetiske overflaten)
- **Diskkræsj**
  Hele disken er permanent uleselig

Sjekksummer – I

- Diskblokker lagres med noen ekstra bit, kalt **sjekksum**
  Sjekksummer gjør systemet robust mot opprettbare diskfeil
- De brukes til å validere en lest eller skrevet blokk:
  - Les blokk + lagret sjekksum
  - Beregn sjekksummen på lest blokk
  - Sammenlign lest og beregnet sjekksum
- Hvis lest og beregnet sjekksum er forskjellige,
  les blokken på nytt og beregn på nytt
  - Lykkes leseoperasjonen, så returner riktig innhold
  - Hvis antall leseforsøk går over en gitt grense,
    returner feilmeldingen “bad disk block”

Sjekksummer– II

- Sjekksummer brukes bare for å *oppdage* feil
- Det er mange måter å beregne sjekksummer på
- 1-bit paritet er den enkleste: tell ener-bits i blokken
  - Likt antall gir paritetsbit 0
  - Odde antall gir paritetsbit 1

  Stor fare for å ikke oppdage feil
- 8-bit paritet: ett paritetsbit pr. bit i en byte:
  Tell 1-ere i mest signifikante bit, ... , minst signifikante bit
  Faren for å overse feil reduseres med faktoren $1/2^7$
- Polynomiske koder – CRC (cyclic redundancy check) :
  - Generer en sjekksum med modulo-2 aritmetikk (XOR)
- Flere andre ....

Håndtering av diskkræsj

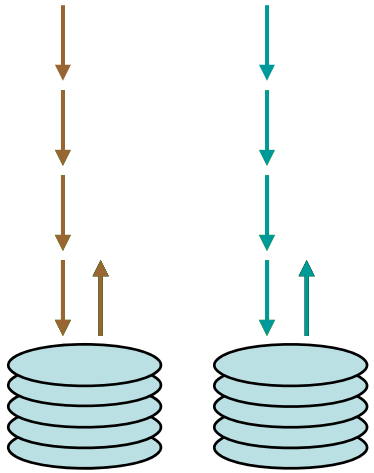
- Det er ingen annen (praktisk) måte å gjenopprette data på etter et diskkræsj enn å ha en kopi på et annet medium som magnetbånd eller en speilet disk
- Samme teknikker som brukes til å håndtere diskkræsj, brukes også til å håndtere uopprettbare diskfeil

- Det finnes en rekke strategier for å redusere faren for tap av data ved permanente diskfeil
  - De fleste baserer seg på en utvidet paritetssjekk
  - De vanligste går under betegnelsen RAID-strategier (RAID = Redundant Arrays of Independent Disks)

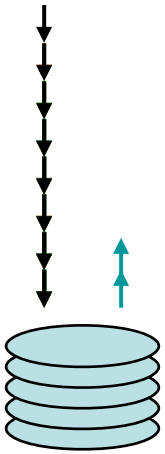
Striping

- Dette er en teknikk for å øke lese- og skrivehastigheten til og fra disk
- I stedet for å skrive alle data til samme disk, splittes dataene og skrives i parallell til n disker
- Dette reduserer selve skrive/lesetiden med en faktor n (søke- og rotasjonsforsinkelsen blir omtrent som før)

To disker:



En disk:



Speiling

- Speiling betyr at vi har flere identiske kopier av hver disk, som oftest to

- Fordeler:
  - Raskere svartider ved lesing
  - Overlever diskkræsj – feiltoleranse
  - Balanserer lasten ved å fordele leseoperasjonene likt mellom de speilede diskene

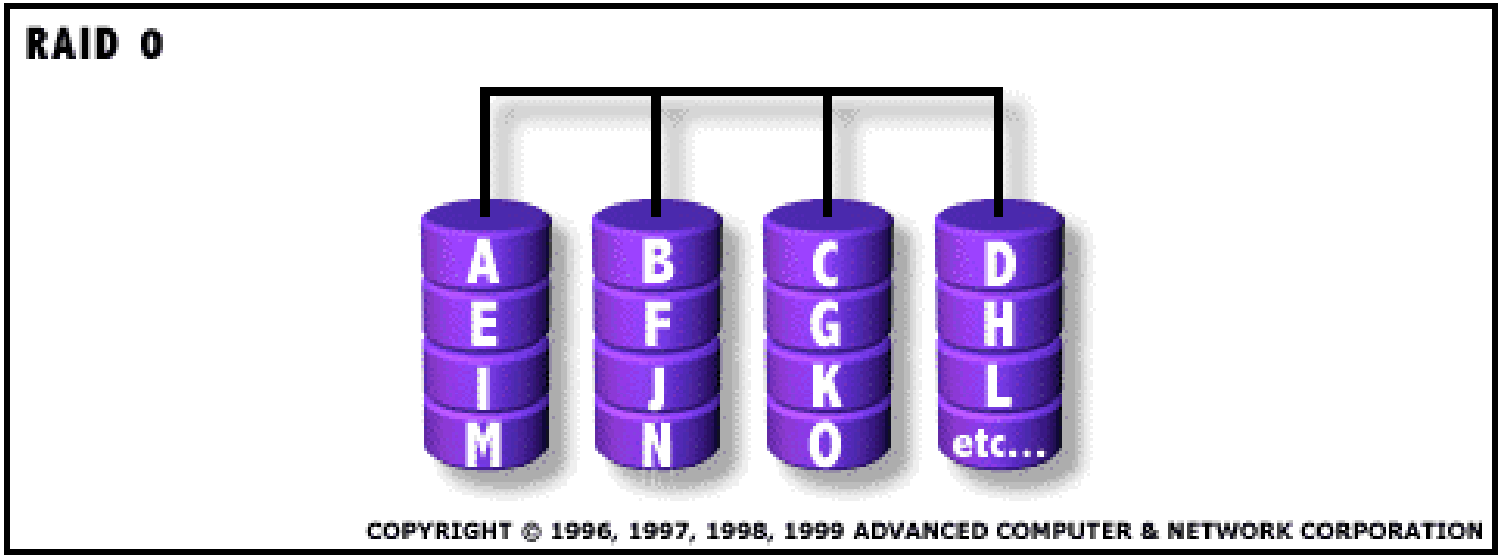
- Ulemper:
  - Øker lagringsbehovet

RAID (Redundant Arrays of Independent Disks)

- RAID level 0: non-redundant
- RAID level 1: mirrored
- RAID level 2: Hamming error correcting code (ECC)
  (Ingen kommersielle implementasjoner)
- RAID level 3: bit-interleaved parity
- RAID level 4: block-interleaved parity
- RAID level 5: block-interleaved distributed-parity
- RAID level 6: multiple redundancy

RAID 0 (ingen redundans) – I

- RAID 0: Stripet diskarray uten feiltoleranse
- Striping betyr at data brekkes opp i logiske blokker som hver skrives til en separat disk

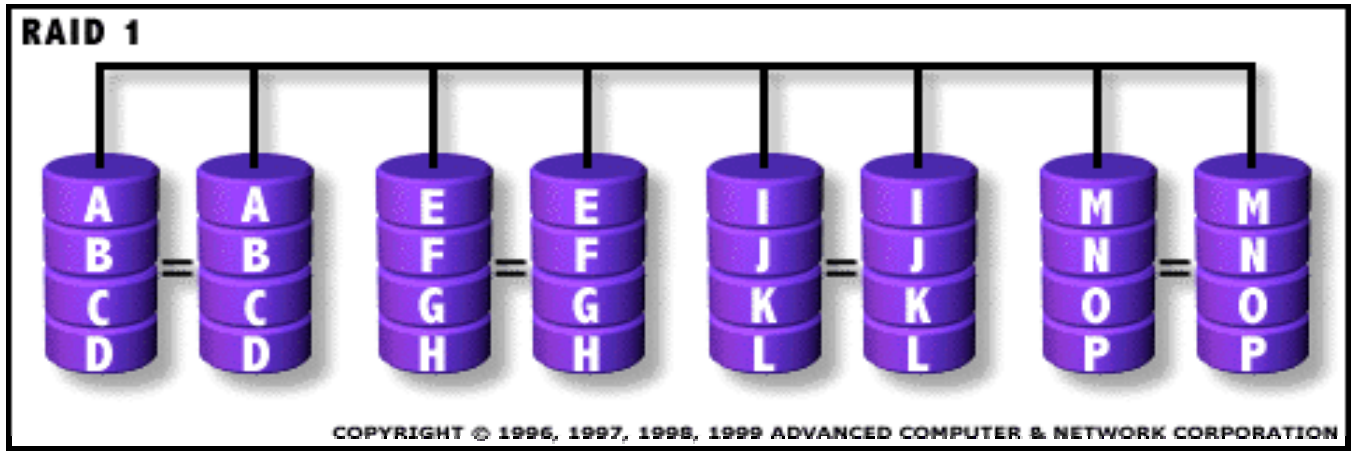


RAID 0 (ingen redundans) – II

- Fordeler
  - Ytelsen blir mye bedre ved å spre I/O på mange kanaler og disker
  - Den beste ytelsen får man når data stripes på flere kontrollere med bare en disk per kontroller
- Ulemper
  - Ikke en ekte RAID fordi den ikke er feiltolerant
  - Hvis en disk feiler, vil alle data i diskarrayen gå tapt
  - Må aldri brukes i feilkritiske situasjoner

RAID 1 (speiling)

- Data er duplisert og ligger på to disker



- Fordeler
  - En skriving eller to samtidige lesinger per speilet par
  - 100% redundans betyr at ingen rekonstruksjon trengs etter en diskfeil, bare en kopi til erstatningsdisken
- Ulempe: Ekstremt høy disk-overhead (100%)

RAID 2 og RAID 3 (Kursorisk)

- RAID 2: Hamming ECC
  - Alle databit skrives til én datadisk (ekstrem striping)
  - Alle dataord får sitt Hammingkodede ECC-ord lagret på egne ECC-disker
  - ECC-koden verifiserer korrekte data og retter feil på en enkelt disk
  - NB! Ingen kommersielle implementasjoner finnes
- RAID 3: Parallell dataoverføring med bit-paritet
  - Datablokkene stripes og skrives på datadiskene
  - Stripepariteten genereres ved skriving og lagres på en paritetsdisk
  - Pariteten sjekkes ved lesing

Modulo-2-summer

- En vanlig måte å lage en korreksjonsblokk (paritetsblokk) på, er å beregne modulo-2-summen av datablokkene (Modulo-2-sum blir ofte kalt XOR - eXclusive OR)
- Modulo-2-summen beregnes ved å la bit k i summen være
  - 1 – hvis et odde antall blokker har 1 i posisjon k
  - 0 – hvis et like antall blokker har 1 i posisjon k
- Eksempel

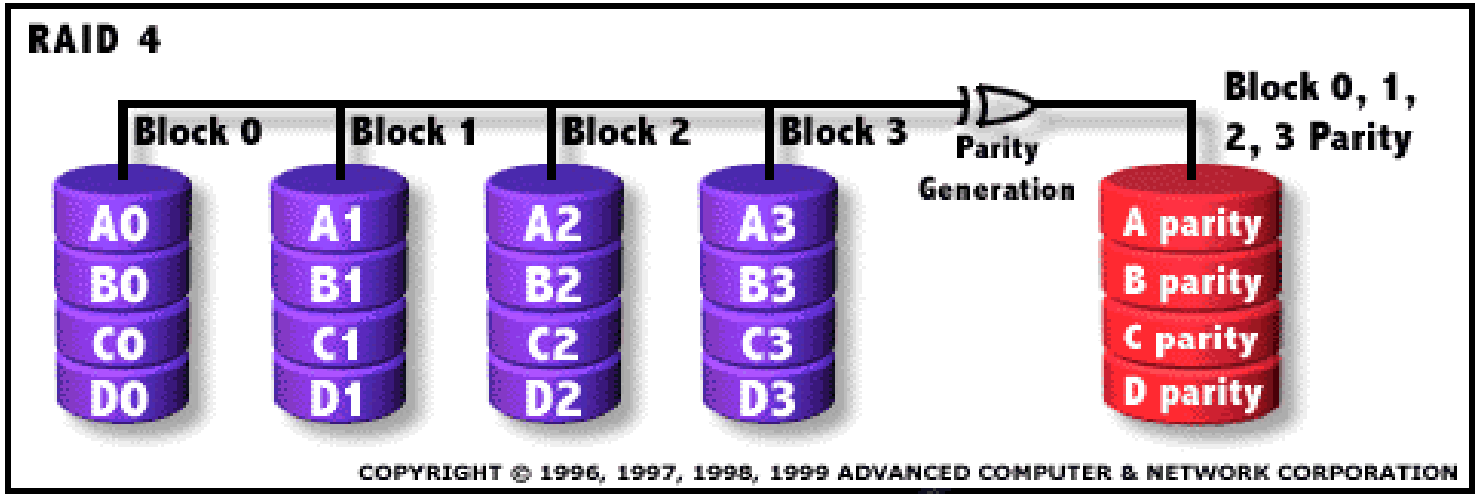
blokk 1	1	1	1	1	0	0	0	0
blokk 2	1	0	1	0	1	0	1	0
blokk 3	0	0	1	1	1	0	0	0
modulo-2-sum	0	1	1	0	0	0	1	0

Regneregler for modulo-2-summer

- La $\oplus$ betegne modulo-2-sum-operatoren
- Da gjelder:
  - Den kommutative loven: $x \oplus y = y \oplus x$
  - Den assosiative loven: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
  - 0 er identiteten: $x = 0 \oplus x = x \oplus 0$
  - $\oplus$ er sin egen invers: $x \oplus x = 0$

RAID 4 (Blokkvis paritet) – I

- Uavhengige datadisker med en felles paritetsdisk
- Hver datablokk skrives i sin helhet til én datadisk
- Pariteten beregnes ved skriving og lagres på en felles paritesdisk
- I figuren har vi at: A parity = A0 ⊕ A1 ⊕ A2 ⊕ A3



RAID 4 – II

- Ved skriving av en datablokk beregnes den nye verdien av den tilhørende paritetsblokken ved formelen:

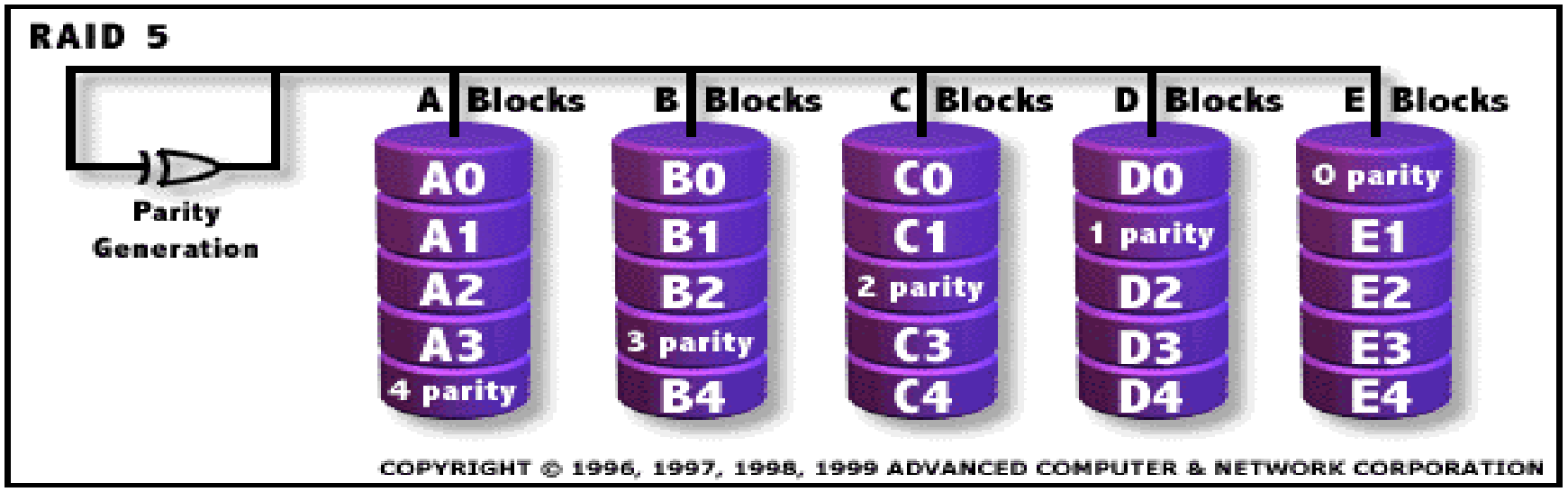
  gammel paritetsblokk ⊕ gammel datablokk ⊕ ny datablokk

- Hvis en enkelt disk kræsjer, det er det samme om det er en datadisk eller paritetsdisken, kan den regeneres ved å ta modulo-2-summen av alle de andre diskene

- En ulempe ved RAID 4 er at paritetsdisken kan bli en flaskehals:
  - Den må skrives ved hver oppdatering

RAID 5 (Blokkvis fordelt paritet)

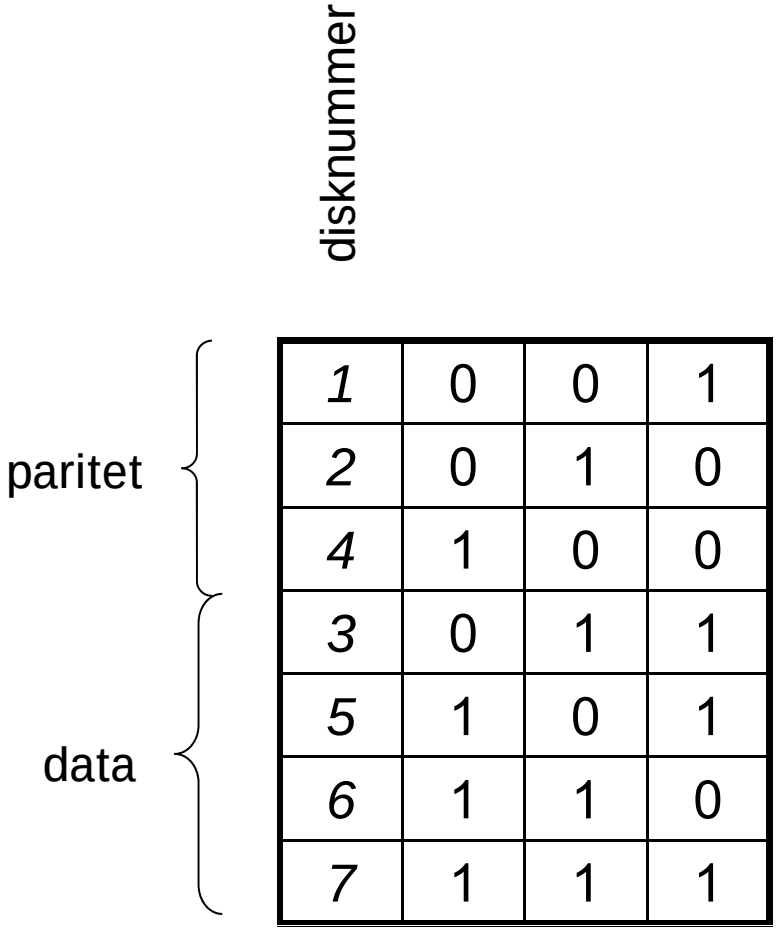
- Logisk identisk med RAID 4, men paritetsblokkene er fordelt på alle diskene, noe som fordeler lasten jevnt
- Anta at vi har n + 1 disker nummerert fra 0 til n
  Da lagres paritet på sylinder i på disk j hvis $j = i \bmod(n+1)$
  Da blir neste paritetssylinder sylinder i + 1 på disk j + 1
- RAID 5 er en mye brukt teknologi



RAID 6

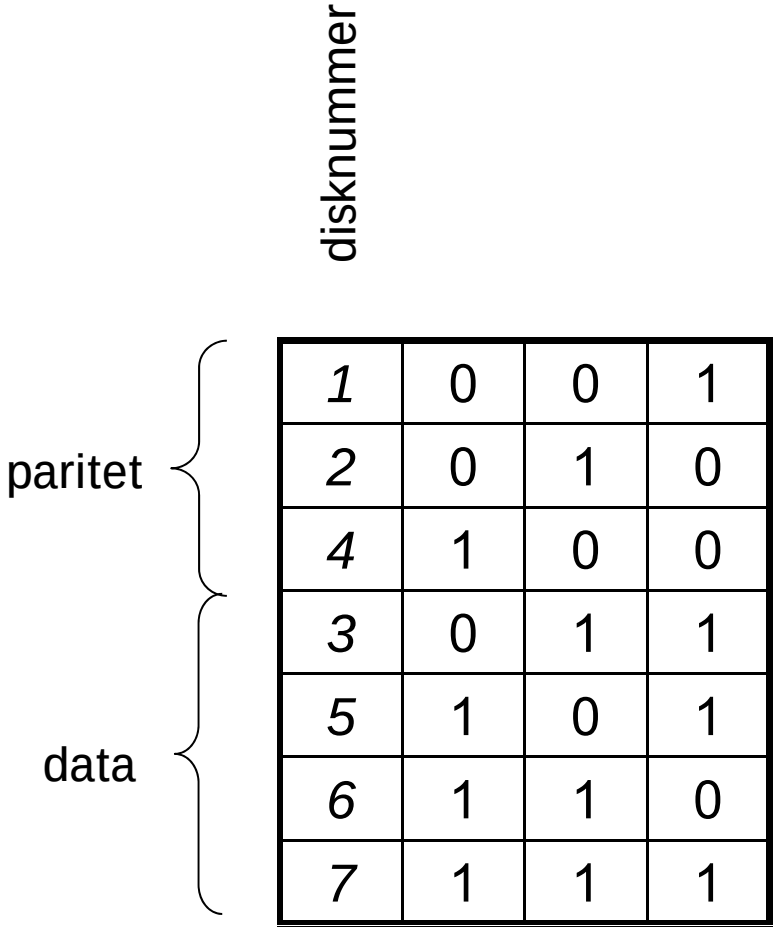
- Dette er egentlig en familie metoder som tåler at mer enn en disk kræsjer samtidig
- De mest avanserte bruker n logiske paritetsdisker (fysisk fordelt som i RAID 5) for å tåle n samtidige diskkræsj
- Vi skal se nærmere på Hamming-kodet RAID 6 som tåler 2 samtidige diskkræsj blant 2^k - 1 disker hvorav k er paritetsdisker (og 2^k - k -1 er datadisker)
- Diskene nummereres fra 1 til 2^k -1 og de identifiseres med sitt nummer som tolkes som et binærtall
- Diskene som bare har ett bit satt (toerpotensene), blir paritetsdisker, og de defineres som modulo-2-summen av de datadiskene som har dette bitet satt

RAID 6 – Hammingkodeeksempel – I



- Vi har 7 disker
- De tre diskene 1, 2 og 4 er paritetsdisker
- Disk 1 er modulo-2-summen av diskene 3, 5 og 7
- Disk 2 er modulo-2-summen av diskene 3, 6 og 7
- Disk 4 er modulo-2-summen av diskene 5, 6 og 7

RAID 6 – Hammingkodeeksempel – II



- Anta at disk 4 og 5 kræsjer
- Siden disk 5 trenges for å regenerere disk 4, må vi først regenerere disk 5
- Disk 1 er modulo-2-summen av diskene 3, 5 og 7, så disk 5 er modulo-2-summen av diskene 1, 3 og 7
- Nå kan vi regenerere disk 4 som modulo-2-summen av diskene 5, 6 og 7

RAID 10

- RAID 10 er en blanding av RAID 0 og RAID 1
- I hvert disk-array (kabinett) ordnes diskene i par som speiles (RAID 1)
- Dataene blir så stripet over parene (RAID 0)
- Dermed kombineres hastigheten i RAID 0 med sikkerheten i RAID 1
- Det er dagens store og billige disker som har gjort RAID 10 konkurransedyktig
- RAID 10 er i ferd med å bli den vanligste driftsformen for store datasentre

Lagring av data på disk

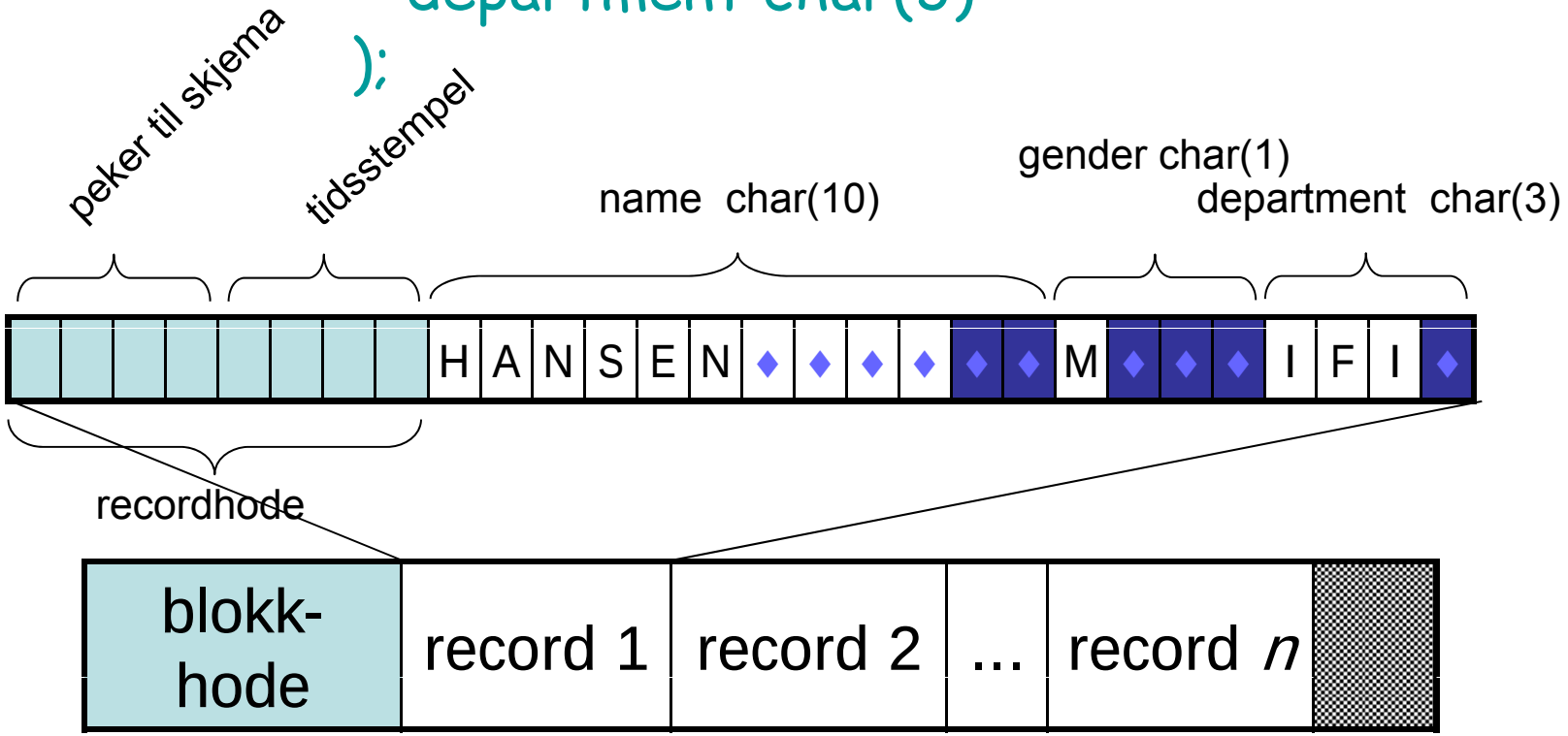
- Tupler/objekter representeres ved **recorder**
  - En record består av en mengde bytes som lagres samlet på en (eller flere) diskblokk(er)
  - Recorden har ett felt pr. attributt
  - Hver record har i tillegg et **recordhode** med administrativ informasjon om innholdet i recorden
  - Databasesystemet lagrer **recordskjemaer** – informasjon om attributter og datatyper, feltenes rekkefølge, ...
  - Hver diskblokk har et **blokkhode** med administrativ informasjon om innholdet i blokken
  - Samhørende diskblokker lagres fortrinnsvis nær hverandre på disk, f.eks. på samme eller tilstøtende sylindere

Recorder med fast lengde

- Ett felt med en gitt lengde pr. attributt
  - Vanligvis settes lengden av et felt til 4n eller 8n for en passende n (mange maskiner arbeider mer effektivt hvis alle felter starter på minneadresser som er delelige med 4 eller 8)
- Recordhodet inneholder typisk
  - en peker til recordskjemaet
  - lengden på recorden
  - pekere til de enkelte feltene
  - tidsstempler
- En blokk inneholder vanligvis recorder fra bare ett skjema
- Blokkhodet inneholder typisk
  - lenker til andre blokker som hører logisk sammen med denne
  - info om relasjonen(e) recordene i blokken tilhører
  - pekere (**offsets**) til de enkelte recordene i blokken
  - tildsstempler

Eksempel: Record med fast lengde

```
create table employee (
   name          char(10),
   gender        char(1),
   department char(3)
);
```

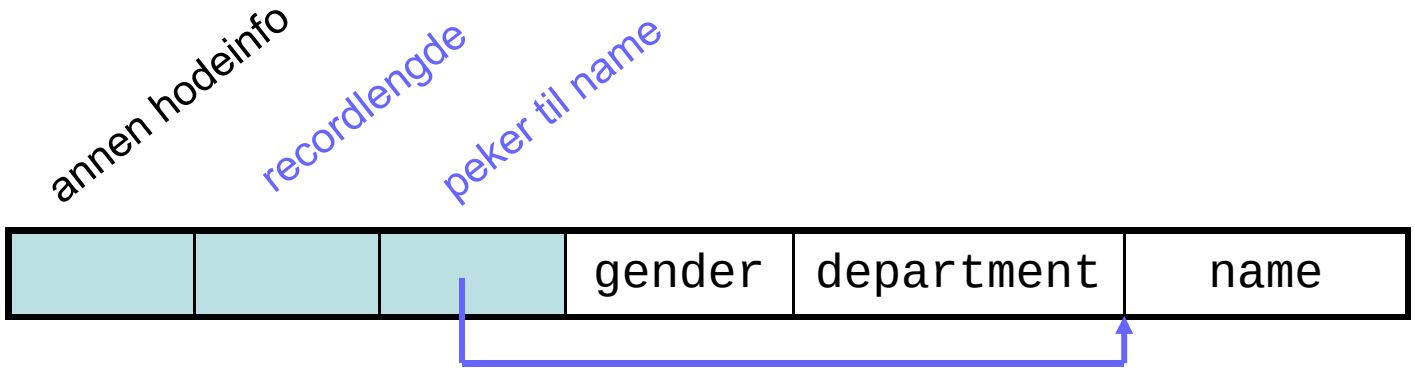


Recorder med variabel lengde

- Recorder med variabel lengde skyldes
  - attributter med variabel lengde
    f.eks. varchar
  - attributter med repeterende felter
    f.eks. multiset (SQL:2003)
  - BLOBer (binary large objects),
    f.eks. lyd, fotoer, film
  - recorder uten fast skjema
    f.eks. XML

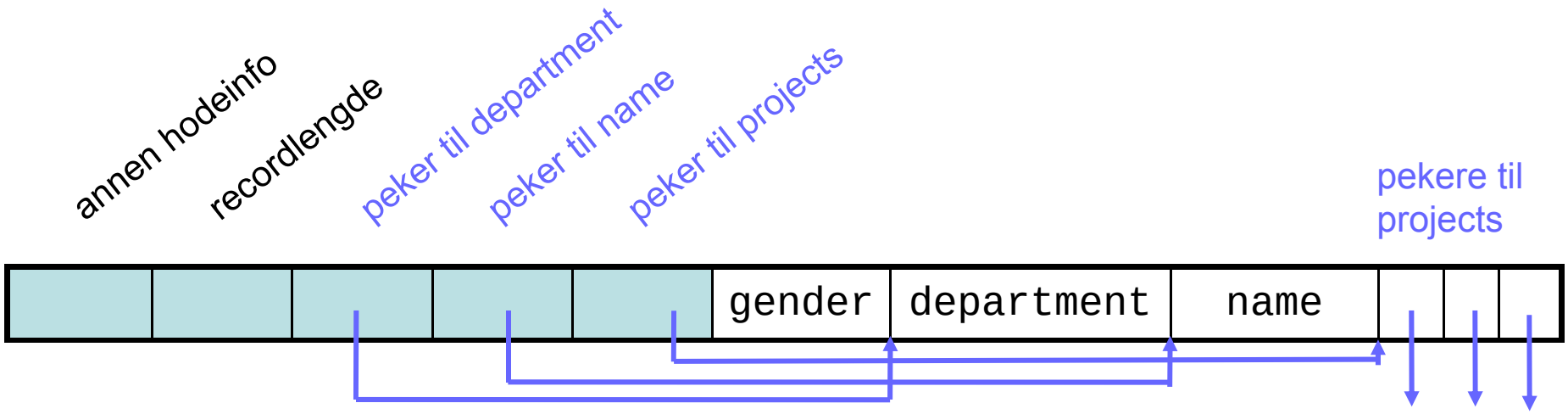
Eksempel: Attributter med variabel lengde

```
create table employee (
   name         varchar(50),
   gender       char(1),
   department char(3)
);
```



Eksempel: Repeterende felter

```
create table employee (
   name          varchar(50),
   gender        char(1),
   department varchar(20),
   projects     ref(ProjectType) multiset
);
```



Adressehåndtering

- Blokker og recorder adresseres
  - i primærminnet: ved virtuell byteadresse til første byte
  - i sekundærlageret: disk ID + fysisk plassering på disken
- I klient-tjenerarkitekturen:
  - Klientprosessen bruker **minneadresser** (dvs. konvensjonelt virtuelt adresserom)
  - Serverprosessen (DBSet) bruker **databaseadresser:**
    - fysiske adresser når blokken er i sekundærlageret
    - logiske adresser når blokken er i primærminnet
    - En **map table** relaterer logiske og fysiske adresser. Flytting eller sletting av en blokk gjøres ved én endring i tabellen; pekere til tabellen (logiske adresser) må ikke endres

Pekerhåndtering

- Pekere i recorder: En **transisjonstabell** oversetter logiske adresser til minneadresser når recorden er i primærminnet
- Pekerswizzling: Når blokker flyttes fra sekundærlager til primærminnet, oversettes pekere (**pointer swizzling**) fra databaseadresserom til virtuelt adresserom. Når blokken returneres til sekundærlager, omgjøres oversettelsen (**unswizzling**)
- Pinned block: For å hindre uønsket skriving tilbake til disk