

# Systemfeil og logging

# Korrekthet av data

- Vi ønsker at data alltid skal være riktige og nøyaktige
- Eksempel på tvilsomme data:

Ansatt

Navn	Alder
Hansen	56
Lie	2489
Olsen	1

# Integritets- eller konsistensregler

- Dette er predikater som data må tilfredsstill
- Eksempler
  - $X$  er kandidatnøkkel i relasjonen  $R$
  - $X \rightarrow Y$  holder i  $R$
  - $\text{Domain}(\text{farge}) = \{R(\text{ød}), G(\text{rønn}), B(\text{lå})\}$
  - Ingen ansatt får tjene mer enn det dobbelte av gjennomsnittslønnen
- Definisjon av konsistens:
  - Konsistent tilstand:  
Tilfredsstiller alle (statiske) integritetsregler
  - Konsistent database:  
Databasen er i en konsistent tilstand

# Avvik fra 100%-prinsippet – I

- 100%-prinsippet sier at informasjonsmodellen kan brukes som databaseskjema
- Det betyr at alle skranker i modellen blir til integritetsregler i skjemaet
- I teorien er dette bra, men i praksis kan det av og til være hensiktsmessig å tillate avvik
- Det betyr at integritetsreglene (de vi håndhever i databasen) ikke alltid trenger å uttrykke den fulle sannhet om UoD
- Merk:  
Integritetsreglene får aldri lov til å være strengere enn skrankene i datamodellen

# Avvik fra 100%-prinsippet – II

To eksempler på tvilsomme skranker:

- Hver gang lønnen oppdateres, gjelder at  
*ny lønn > gammel lønn*
- Hver gang en konto-forekomst fjernes, gjelder at  
*kontoens saldo = 0*

Det siste eksemplet kan vi «fikse» med enkle midler som å legge til et boolsk attributt:

Konto	Kto#	...	saldo	fjernet?
-------	------	-----	-------	----------

# Midlertidig inkonsistens – I

Observasjon:

Databaser kan ikke alltid være konsistente

Eksempel:  $a_1 + a_2 + \dots + a_n = \text{total}$  (*integritetsregel*)

Sett inn 1000 kroner på konto  $a_2$ :

$$a_2 \leftarrow a_2 + 1000$$

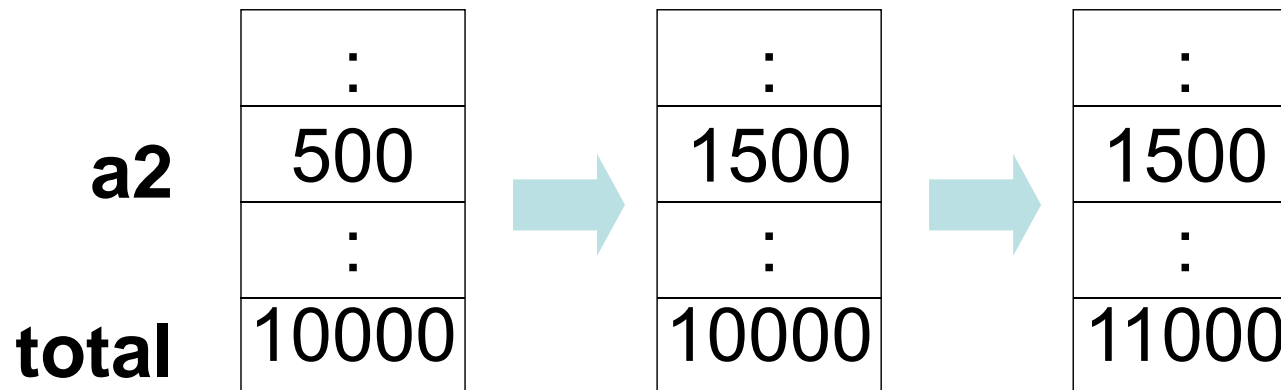
$$\text{total} \leftarrow \text{total} + 1000$$

*Integritetsregelen er midlertidig brutt*

# Midlertidig inkonsistens – II

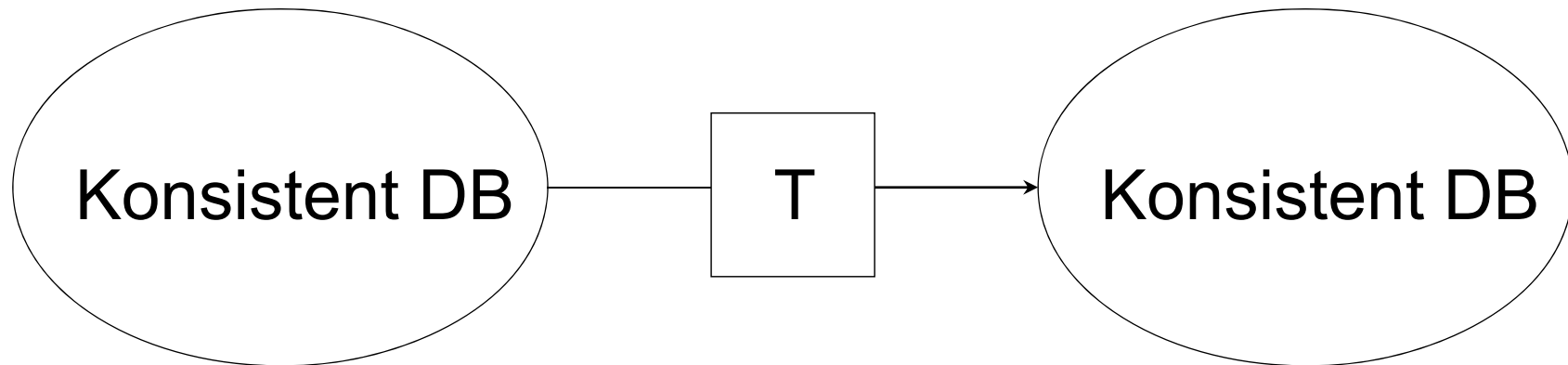
Eksempel:  $a_1 + a_2 + \dots + a_n = \text{total}$  (*integritetsregel*)

Sett inn 1000 kroner på  $a_2$ :  
 $a_2 \leftarrow a_2 + 1000$   
 $\text{total} \leftarrow \text{total} + 1000$



# Transaksjoner – I

En *transaksjon* er en samling aksjoner som bevarer konsistens





# Transaksjoner – II

- **En viktig antagelse**

Hvis T starter i en konsistent tilstand,  
og T eksekverer alene,  
så vil T avslutte med å etterlate  
databasen i en konsistent tilstand

- Korrekthet (uformelt):
  - Hvis vi stopper å utføre transaksjoner, etterlater vi DB i en konsistent tilstand
  - Alle transaksjoner opplever en konsistent DB

# Årsaker til inkonsistens og brudd på integritetsreglene

- Feil i en transaksjon
- Feil i DBMS
- Hardware-feil
  - F.eks.: Et diskkræsje endrer saldo på en konto
- Deling av data
  - F.eks.: T1: Øk lønnen til alle med tittel «programmerer» med 4,5%
  - T2: Forandre Tove Hansens tittel fra «programmerer» til «systemerer»

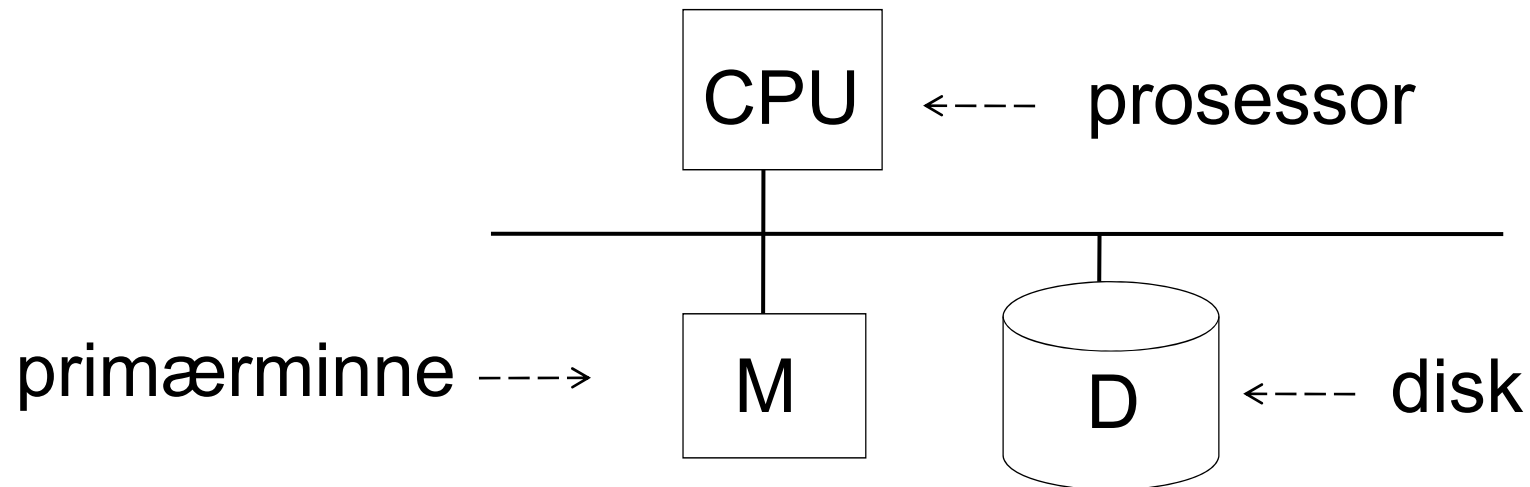
# Viktige ting INF3100 *ikke* omfatter

- Hvordan å skrive/programmere feilfrie transaksjoner
- Hvordan å skrive/programmere et feilfritt DBMS
- Hvordan å kontrollere integritetsreglene i databaseskjemaet og rette opp brudd på dem

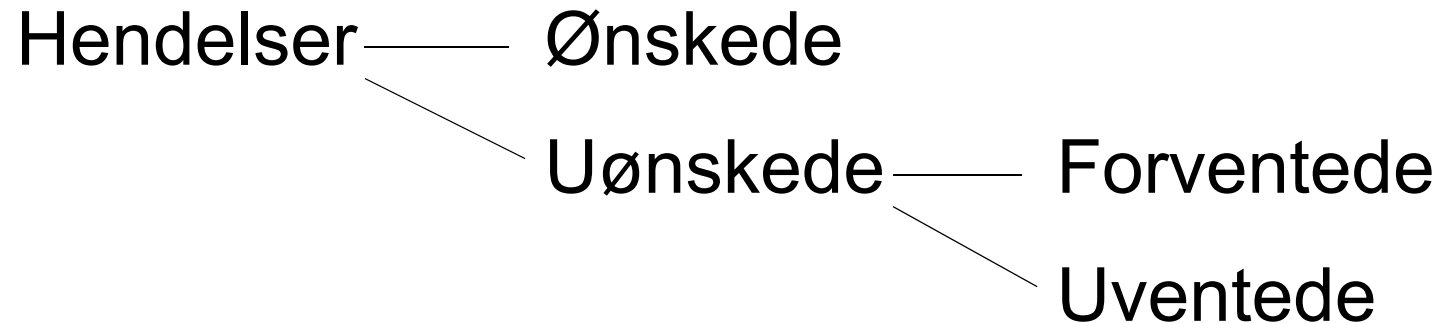
Det siste betyr at de løsningene som vi gir på håndtering av feilsituasjoner, ikke trenger å ta hensyn til hvilke integritetsregler som er definert i databaseskjemaet

# En modell for beskrivelse av feil I

## Konfigurasjonsmodellen:



# Klassifikasjon av hendelser



- Ønskede hendelser: Se produkthåndbøkene ...
- Forventede uønskede hendelser:
  - Systemkræsj: – tap av primærminnet
  - CPUen stopper eller må resettes

- 
- Uventede uønskede hendelser: Alt annet!

# Uventede uønskede hendelser: Alt annet!

## Eksempler:

- Diskdata går tapt
- Primærminnet går tapt uten at CPUen stopper
- CPUen tar kontroll over alle datamaskiner hos Hafslund, setter opp spenningen og brenner av alle strømledninger i Oslo og Akershus
- Kapittel 1 i «The Hitch Hiker's Guide to the Galaxy» viser seg å ikke være Science Fiction likevel ....

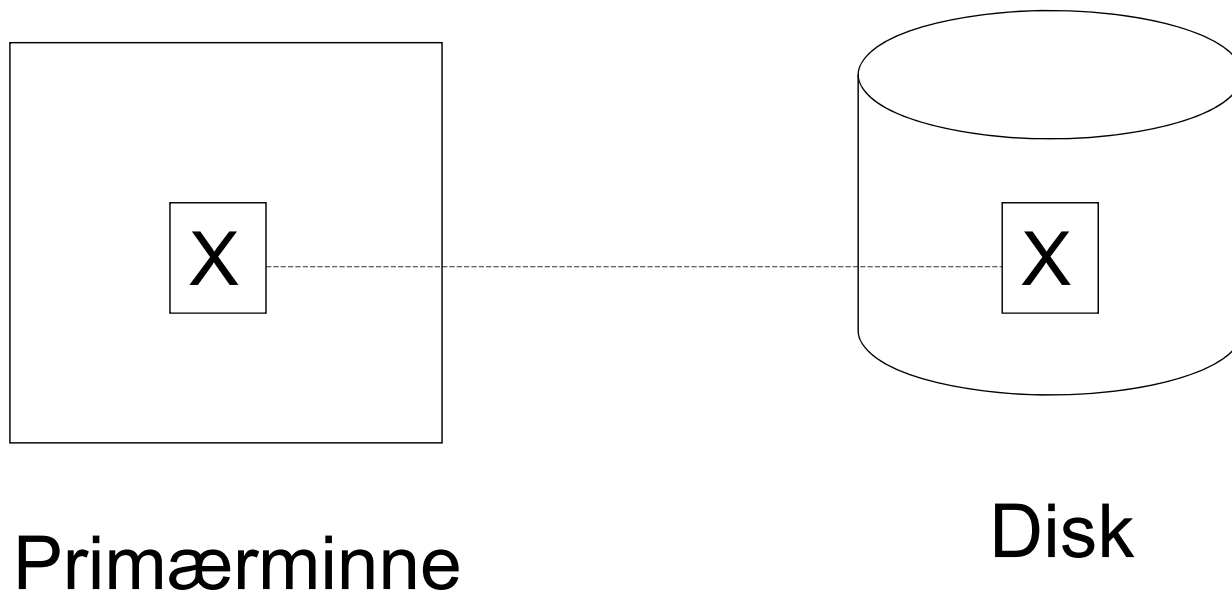
# Et forslag til løsning

Strategi: Legg til lavnivåkontroller og redundans for å øke sannsynligheten for at modellen holder

Eksempler: Replikert disk (stable store)  
Paritetssjekk av minnet  
CPU-sjekker

# En modell for beskrivelse av feil II

## Lagringshierarkiet:





# Primitive operasjoner i transaksjoner

- Input(x): diskblokk med  $x \rightarrow$  primærminnet
- Read(x,v): utfør, om nødvendig,  $\text{input}(x)$   
 $v \leftarrow$  verdien av  $x$  i blokken
- Write(x,v): utfør, om nødvendig,  $\text{input}(x)$   
verdien av  $x$  i blokken  $\leftarrow v$
- Output(x): minneblokk med  $x \rightarrow$  disk

# En viktig antakelse

- I beskrivelsen av de primitive operasjonene har vi implisitt antatt at hvert dataelement ligger inne i én diskblokk/minneblokk
- Dette er opplagt riktig hvis dataelementet er en blokk
- Hvis et dataelement er fordelt på flere fysiske blokker, skal vi håndtere hver av disse blokkene som egne dataelementer
- Dermed antar vi at hvert dataelement alltid ligger inne i en enkelt blokk

# Ufullførte transaksjoner – I

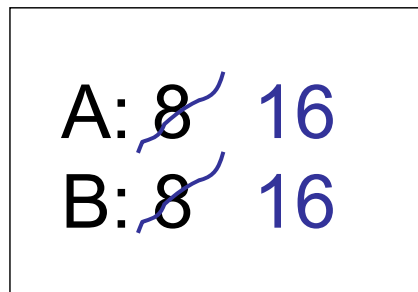
- Dette er hovedproblemet i transaksjonshåndtering
- Eksempel: Integritetsregel:  $A = B$

$$\begin{array}{l} \text{T1: } A \leftarrow A \times 2 \\ \quad B \leftarrow B \times 2 \end{array}$$

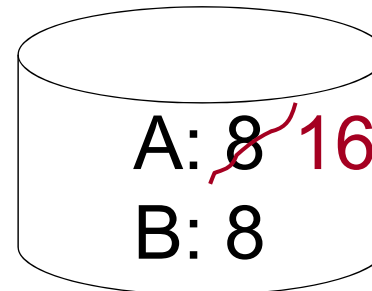
# Ufullførte transaksjoner – II

T1: Read(A,t);  $t \leftarrow t \times 2$ ;  
Write(A,t);  
Read(B,t);  $t \leftarrow t \times 2$ ;  
Write(B,t);  
Output(A);  
Output(B);

*feil!*



primærminne



disk

# Atomisitet

- Transaksjoner må være ***atomære***
- Dette gir oss bare to muligheter:
  - 1) Å utføre alle operasjonene i transaksjonen
  - eller
  - 2) Å ikke utføre noen av operasjonene i transaksjonen

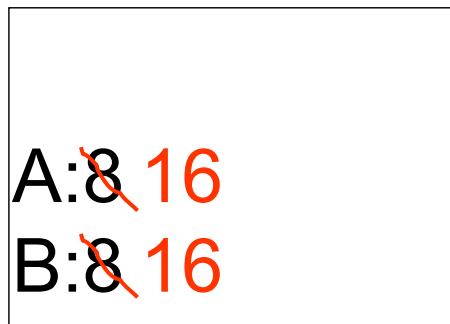
# Logging

- Logging er den viktigste teknikken et DBMS har for å rette opp forventede uønskede hendelser
- Logging består i å skrive alle endringer gjort i databasen til en egen loggfil, en fil som aldri skal ligge på samme fysiske disk som databasen
- Det er to hovedtyper logging, undo og redo
- Upresist kan disse beskrives slik:
  - Undo-logging lar oss rette opp alt som har gått galt
  - Redo-logging lar oss gjenopprette alt som har gått riktig

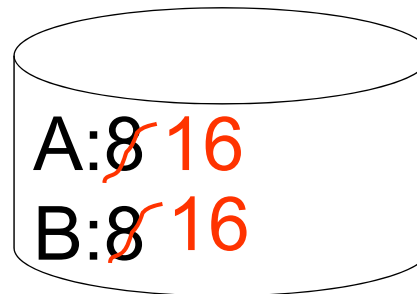
# Undo-logging – I

T1: Read(A,t);  $t \leftarrow t \times 2$ ;  
Write(A,t);  
Read(B,t);  $t \leftarrow t \times 2$ ;  
Write(B,t);  
Output(A);  
Output(B);

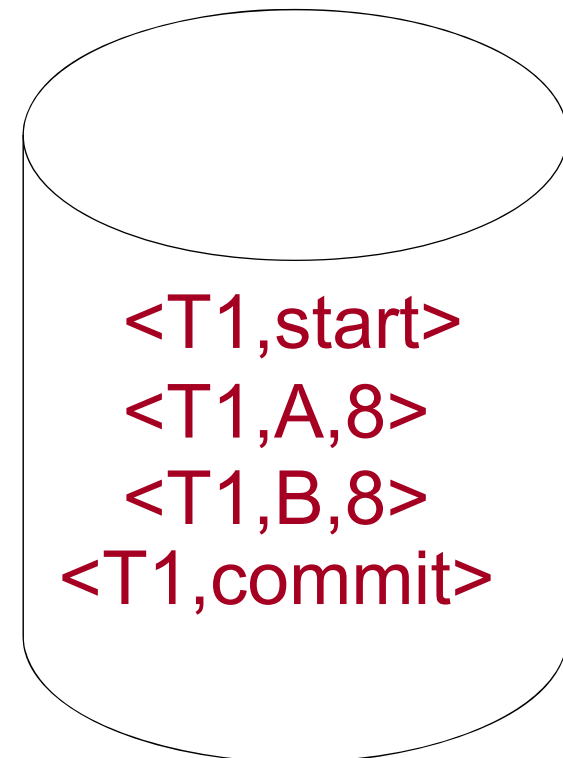
Invariant:  $A = B$



primærminne



disk

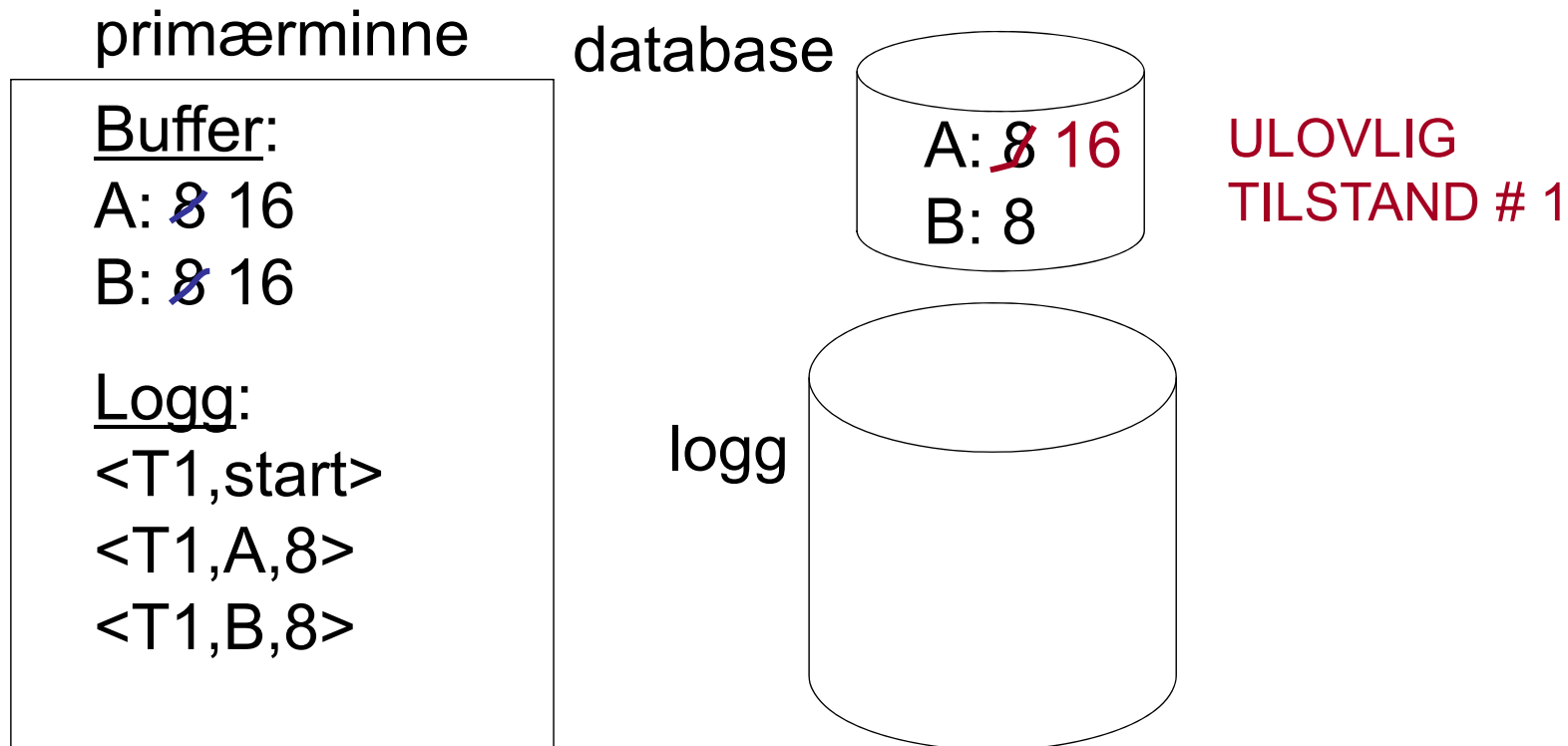


logg

# Undo-logging – II

## En kompliserende faktor:

- Loggen skrives i primærminnet **før** den skrives til disk
- Loggen skrives ikke til disk for hver enkeltoperasjon

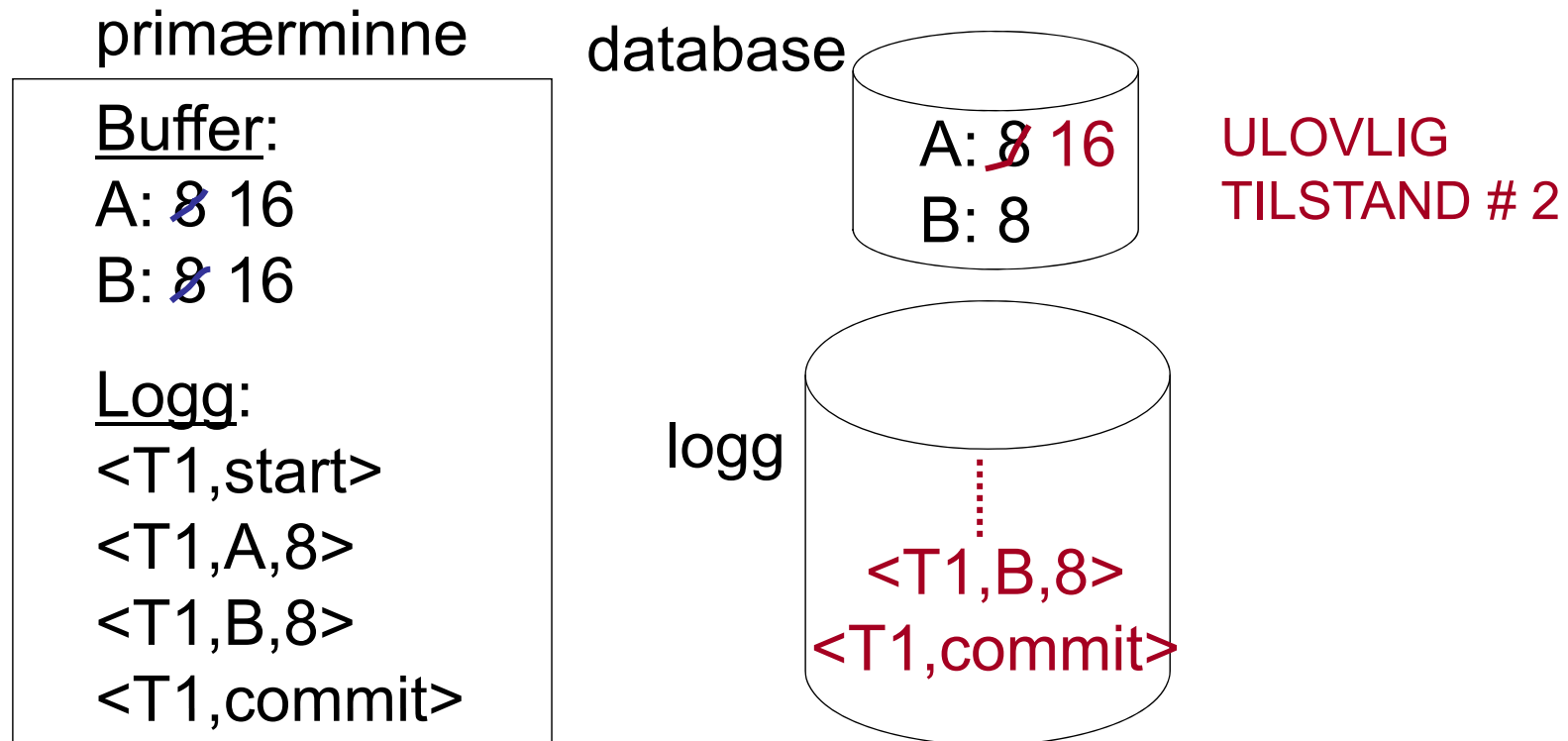




# Undo-logging – III

## En kompliserende faktor:

- Loggen skrives i primærminnet **før** den skrives til disk
- Loggen skrives ikke til disk for hver enkeltoperasjon



# Undo-logging – IV

- Regler for undo-logging:
  - 1) For hver skriveoperasjon  $\text{write}(X,v)$ :  
Skriv en linje i loggen som inneholder den gamle verdien av  $X$
  - 2) Før  $X$  endres på disken, må alle logglinjer som gjelder  $X$ , være skrevet til disk  
Strategi: *Skriv først til logg, så til disk*
  - 3) Før commit kan skrives i loggen, må alle skriveoperasjonene i transaksjonen være overført til disken
- Strategi: *Alt må oppdateres før commit*

# Undo-logging – V

- I praksis må  $X$  alltid være en hel blokk i en Undo-logg

## Begrunnelse:

Anta at  $A$  og  $B$  ligger i samme blokk og blir oppdatert av hver sin (samtidige) transaksjon,  $T_A$  og  $T_B$ , og at  $T_A$  blir ferdig først

Da må blokken skrives til disk før  $T_A$  kan committe

Men blokken kan ikke skrives før vi vet at  $T_B$  har loggført alle sine endringer i blokken

Følgelig må  $T_A$  vente på  $T_B$

# Undo-logging – VI

- Algoritme for gjenoppretting :
  - 1) Sett  $S$  = mengden av transaksjoner  $T_i$  hvor  $\langle T_i, \text{start} \rangle$ , men hverken  $\langle T_i, \text{commit} \rangle$  eller  $\langle T_i, \text{abort} \rangle$  finnes i loggen
  - 2) Les loggen i bakvendt orden (fra sist til først).  
For hver  $\langle T_i, X, v \rangle$  i loggen:  
    Hvis  $T_i \in S$  så  
        write( $X, v$ )  
        output( $X$ )
  - 3) For hver  $T_i \in S$ :  
    Skriv  $\langle T_i, \text{abort} \rangle$  i loggen

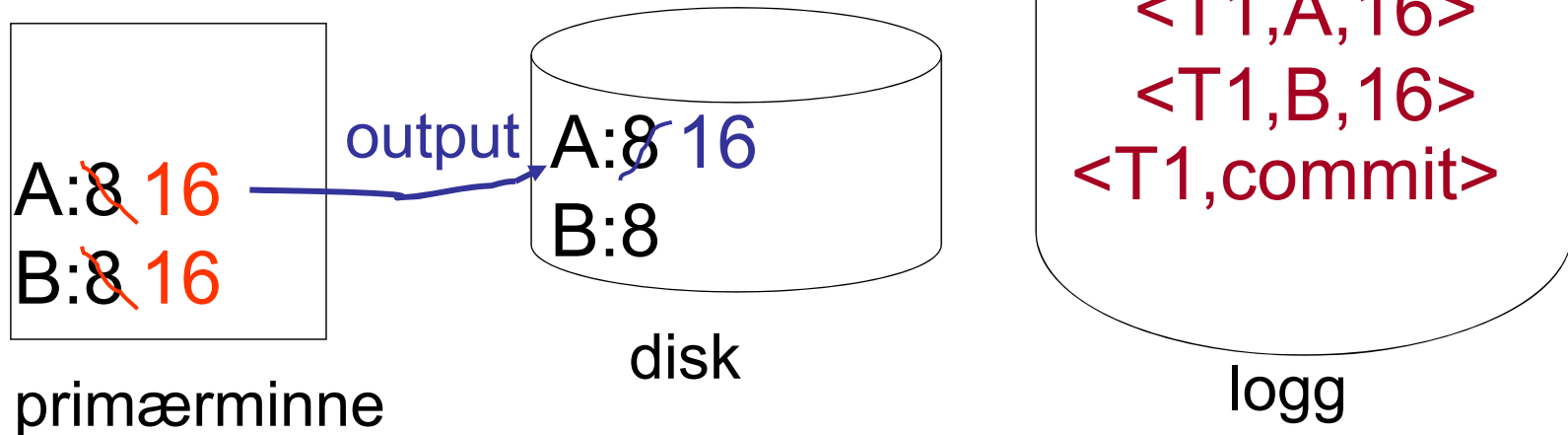
# Undo-logging – VII

- Hva hvis det skjer en feil under gjenoppretting?
- Ikke noe problem!  
Undo er **idempotent**  
(Det betyr at å gjøre undo mange ganger gir samme resultat som å gjøre undo én gang)

# Redo-logging – I

T1: Read(A,t);  $t \leftarrow t \times 2$ ;  
Write(A,t);  
Read(B,t);  $t \leftarrow t \times 2$ ;  
Write(B,t);  
Output(A);  
Output(B);

Invariant:  $A = B$



# Redo-logging – II

- Regler for redo-logging:
  - 1) For hver skriveoperasjon  $\text{write}(X,v)$ :  
Skriv en linje i loggen som inneholder den nye verdien av  $X$
  - 2) Flush loggen (dvs. skriv loggen til disk) ved commit
  - 3) Før  $X$  endres på disken, må alle logglinjer som gjelder  $X$  (inkludert commit), være skrevet til disk
- Strategi:  
*Skriv ikke til disk før loggen er ferdigskrevet*

# Redo-logging – III

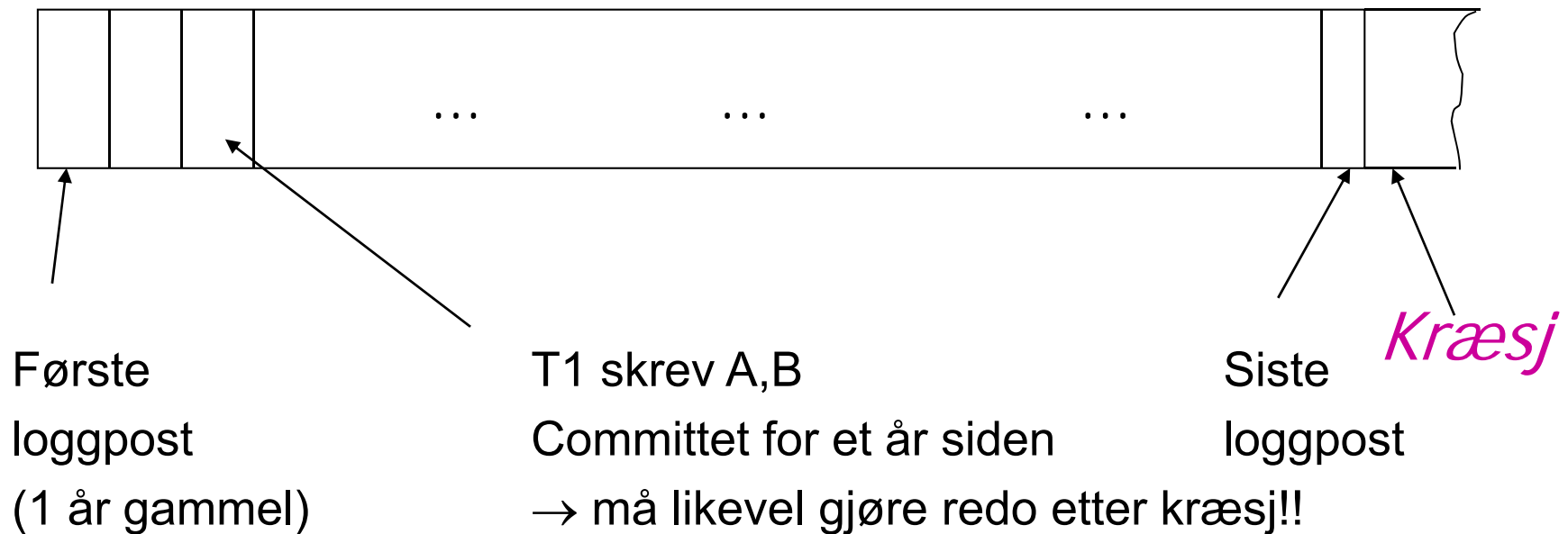
- Algoritme for gjenoppretting :
  - 1) Sett  $S$  = mengden av transaksjoner  $T_i$  hvor  $\langle T_i, \text{commit} \rangle$  finnes i loggen
  - 2) Les loggen forlengs (fra først til sist).  
For hver  $\langle T_i, X, v \rangle$  i loggen:
    - Hvis  $T_i \in S$  så
      - write( $X, v$ )
      - output( $X$ ) ← ikke nødvendig
- Her trenger ikke  $X$  være en hel blokk



# Redo-logging – IV

- Med tiden blir gjenoppretting en **TREG** prosess

Redo-logg:



# Redo-logging – V

Løsning: Sjekkpunkt (enkel versjon)

Utfør regelmessig:

- 1) Stans start av nye transaksjoner
- 2) Vent til alle transaksjoner har avsluttet
- 3) Skriv alle loggposter til disk (flush loggen)
- 4) Skriv alle buffere til disk (DB)  
(behold dem fortsatt i primærminnet)
- 5) Skriv en sjekkpunkt-post til disk (i loggen)
- 6) Gjenoppta transaksjonsbehandlingen

# Redo-logging – VI

Eksempel: Hva må gjøres ved gjenoppretting?

Redo-logg (disk):

⋮	<T1,A,16>	⋮	<T1,commit>	⋮	Checkpoint	⋮	<T2,B,17>	⋮	<T2,commit>	⋮	<T3,C,21>	<i>Kræsje</i>
---	-----------	---	-------------	---	------------	---	-----------	---	-------------	---	-----------	---------------

# Undo- vs redo-logging

- Hovedproblemene med de to strategiene er:
- ***Undo-logging:*** Vi kan ikke holde en backup-DB oppdatert til enhver tid  
Dataelementene må være en hel blokk
- ***Redo-logging:*** Vi må holde alle oppdaterte blokker i minnet helt til commit (og litt til)

# Undo/redo-logging – I

- Løsningen på problemet er undo/redo-logging:

For alle skriveoperasjoner skriver vi i loggen:

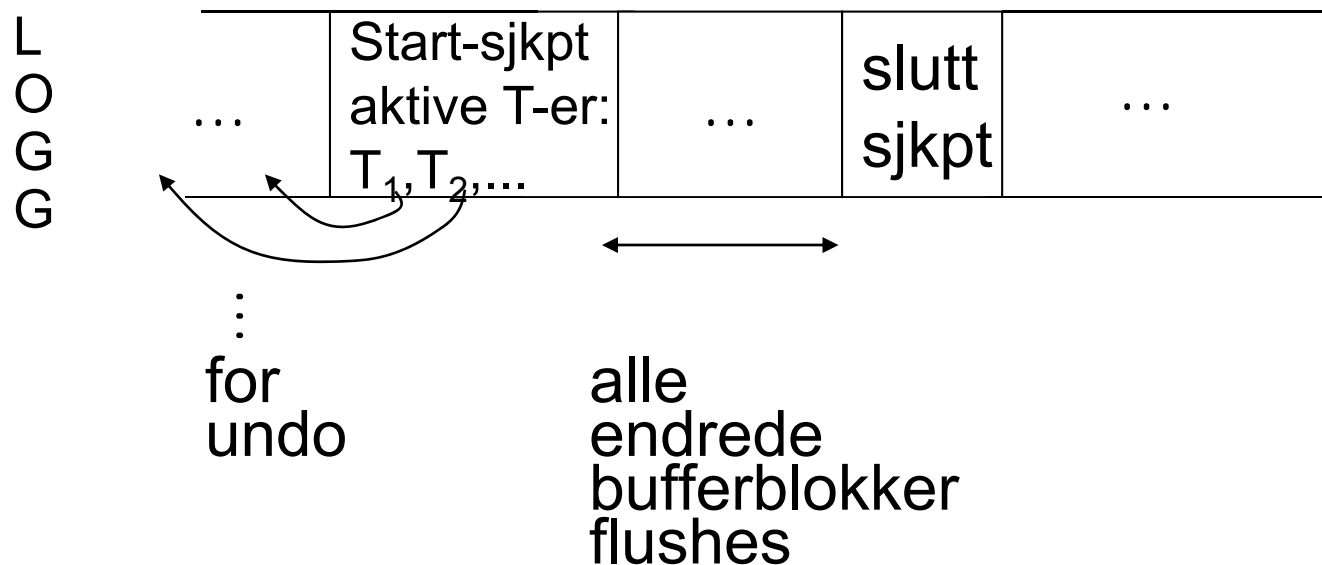
$\langle T_i, X, \text{gammel } X\text{-verdi, ny } X\text{-verdi} \rangle$

# Undo/redo-logging – II

- Regler for undo/redo-logging:
  - 1) Ingen skriveoperasjon  $\text{write}(X,v)$  får utføres før transaksjonen vet at den ikke må abortere
  - 2) For hver  $\text{write}(X,v)$ :
    - Skriv en linje (post) i loggen som inneholder både den gamle og den nye verdien av  $X$
  - 3) Skriv loggposten til disk før  $X$  oppdateres i DB
  - 4) Flush loggen ved commit
- De eneste kravene som må oppfylles før  $X$  skrives til disk, er punkt 1 og 3 ovenfor
- $X$  kan skrives til disk både før og etter at transaksjonen committer
- Dette gjør at  $X$  ikke trenger å være en hel blokk

# Ikke-passiviserende sjekkpunkt – I

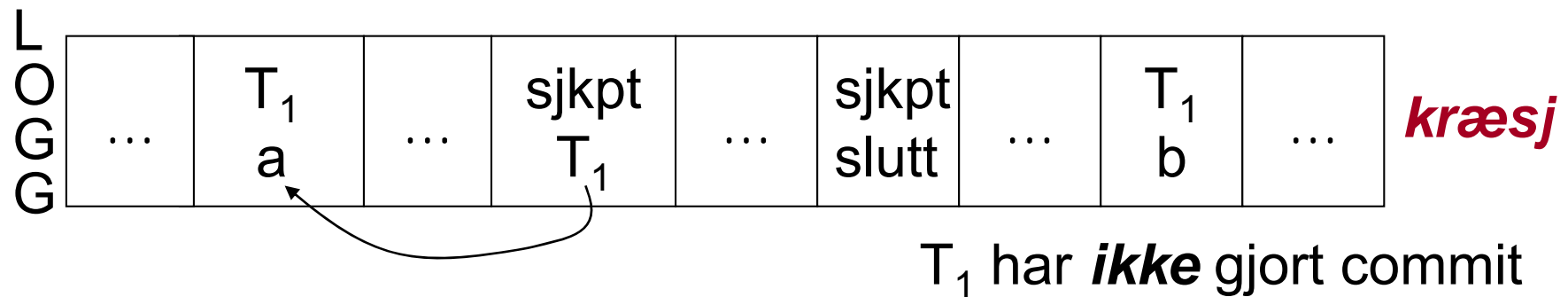
Sjekkpunkt som ikke hindrer start av nye transaksjoner:



Sjekkpunktet er slutt når alle endrede bufferblokker er skrevet til disk. Loggen flushes både ved start og stopp

# Ikke-passiviserende sjekkpunkt – II

Eksempel: Hva gjøres ved gjenoppretting?



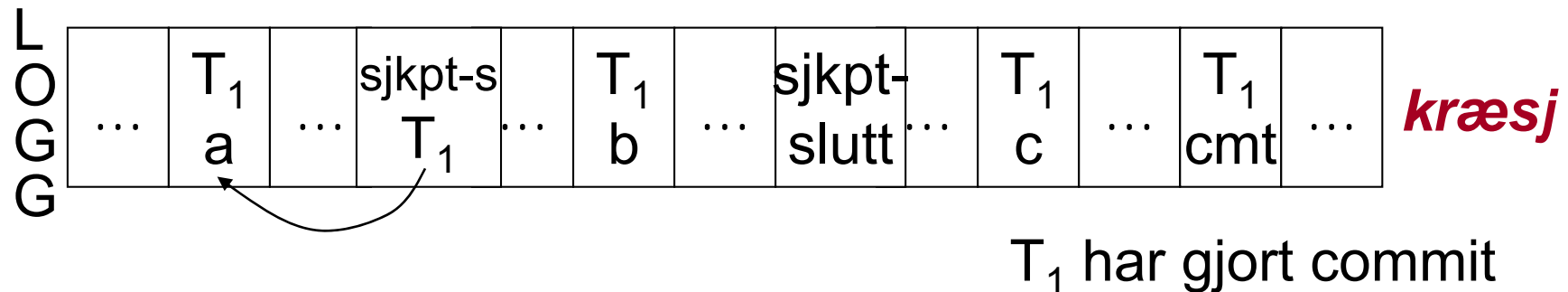
→ Undo T<sub>1</sub> (undo a,b)

Husk: Undo tas bakfra frem til starten av eldste ikke-committede transaksjon



# Ikke-passiviserende sjekkpunkt – III

Eksempel: Hva gjøres ved gjenoppretting? (forts.)

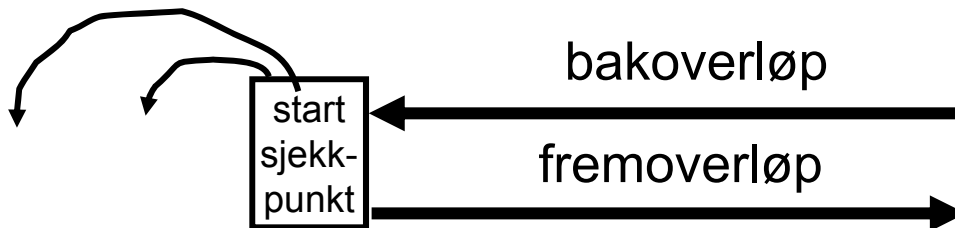


→ Redo T<sub>1</sub> (redo b,c)

Husk: Redo tas forfra fra siste sjekkpunktstart

# Undo/redo-gjenopprettingsprosessen

- Bakoverløp (fra loggslutt til siste start sjekkpunkt):
  - bygg opp en mengde S av committede transaksjoner
  - gjør Undo på alle operasjoner gjort av transaksjoner som ikke er i S
- Oppryddingsfase (foran siste start sjekkpunkt):
  - gjør Undo på resten av operasjonene gjort av de transaksjonene i sjekkpunktets aktiv-liste som ikke er i S
- Fremoverløp (fra siste start sjekkpunkt til loggslutt):
  - gjør Redo på operasjonene gjort av transaksjonene i S



# Fysiske hendelser

Eksempel: Uttak av penger i en bankautomat

$$T_i = a_1 a_2 \dots a_k \dots a_n$$

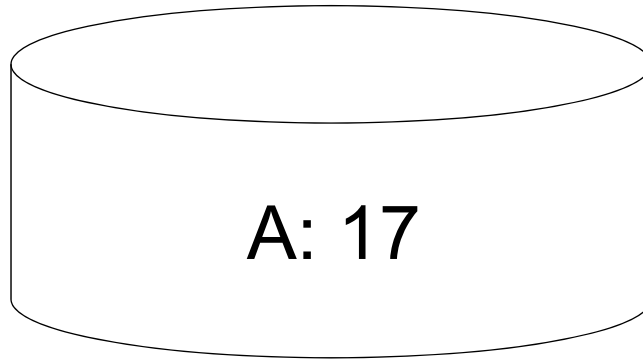
↓  
kontantuttak

Løsning:

- 1) Utfør fysiske operasjoner etter commit
- 2) Prøv å gjøre fysiske operasjoner idempotente

# Håndtering av fysiske feil – I

Eksempel: Diskkræsje eller lignende ødeleggelse av et permanent datalagringsmedium

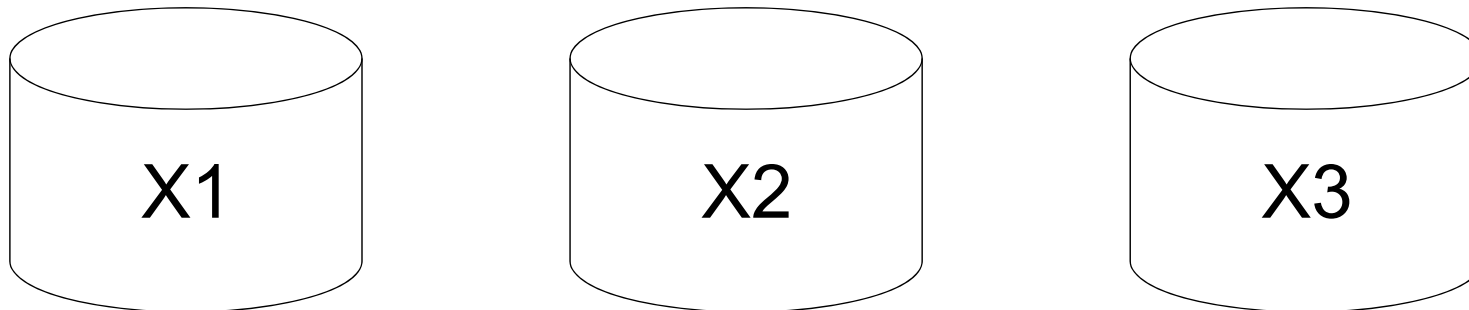


Løsning: Lag kopier av data!

# Håndtering av fysiske feil – II

## Forslag 1: Fysisk trippel-lagring av alle data

- Ha tre kopier på hver sin disk
- $\text{Output}(X) \Rightarrow$  tre fysiske  $\text{output}(X)$
- $\text{Input}(X) \Rightarrow$  tre fysiske  $\text{input}(X)$  + votering



# Håndtering av fysiske feil – III

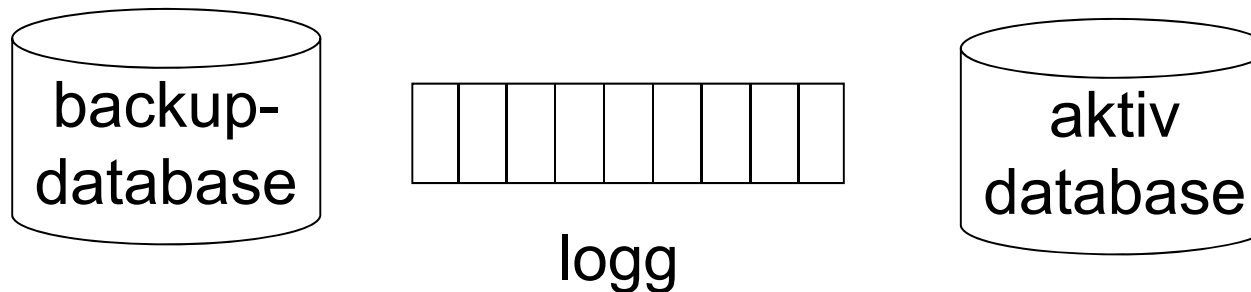
## Forslag 2: Redundant skrivning, enkel lesning

- Ha N kopier på hver sin disk
- $\text{Output}(X) \Rightarrow N$  fysiske  $\text{output}(X)$
- $\text{Input}(X) \Rightarrow$  en fysisk  $\text{input}(X)$ 
  - hvis OK, ferdig
  - ellers, prøv en annen

*Strategien forutsetter at korrupte data kan oppdages*

# Håndtering av fysiske feil – IV

## Forslag 3: Dump av DB + logg



- Hvis den aktive databasen går tapt, så
  - kopier backupdatabasen til en ny aktiv database
  - bruk redo-dataene i loggen til å gjøre databasen up-to-date

# Sletting av loggfilen

