UNIVERSITETET
I OSLO

ifi

Systemfeil og logging

Integritetsregler

- Vi ønsker at data alltid skal være **korrekte**: Integritetsregler er predikater som data må tilfredsstille

- Eksempler:
  - X er kandidatnøkkel i relasjonen R
  - X $\rightarrow$ Y holder i R
  - Domain(farge) = {Rød,Grønn,Blå}
  - Ingen ansatt får tjene mer enn det dobbelte av gjennomsnittslønnen

  } statiske regler

  - Hver gang lønnen oppdateres, skal *ny lønn* > *gammel lønn*

  } dynamisk regel

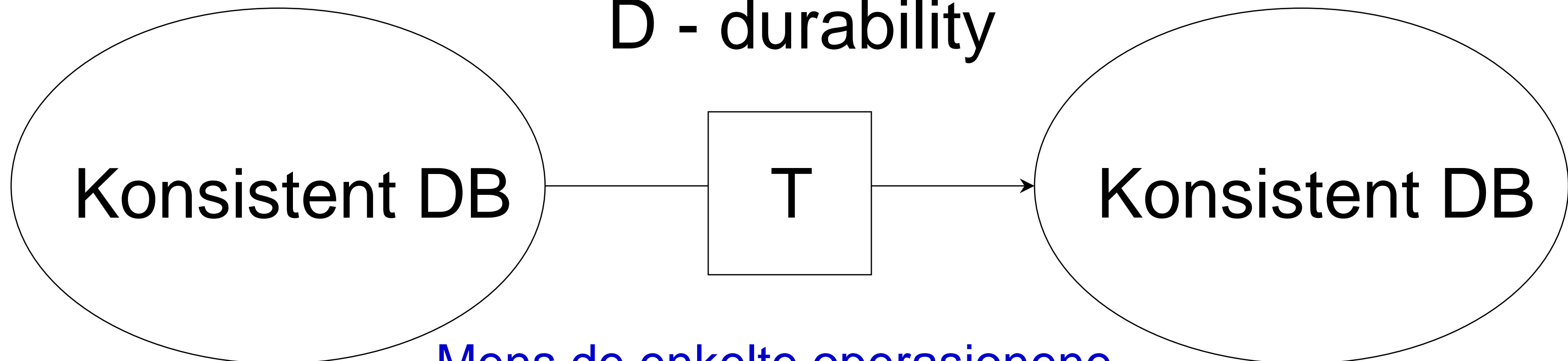
Integritetsregler og konsistens

- Definisjon av konsistens:
  - **Konsistent tilstand**:
    Tilfredsstiller alle (statiske) integritetsregler
  - **Konsistent database**:
    Databasen er i en konsistent tilstand

Transaksjoner – I

En **transaksjon** er en samling aksjoner ***som bevarer konsistens***

A – atomicity
C – **consistency**
I – integrity
D - durability



Mens de enkelte operasjonene utføres, kan noen av integritets-reglene måtte brytes ***midlertidig***

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 databasen i en konsistent tilstand
- Alle transaksjoner opplever det som om databasen er konsistent

Årsaker til inkonsistens og brudd på integritetsreglene

- Feil i programmeringen av en transaksjon
- Feil i DBMS (programvarefeil)
- Hardwarefeil
  - F.eks.: Mediafeil gjør at saldo på en konto endres
- Deling av data
  - F.eks.: Integritetsregel: $x \geq y$

    T1: if x>y then y:=y+1

    T2: if x>y then x:=x-1

    T1 og T2 utføres samtidig fra en tilstand der x=10 og y=9. Etterpå er x=9 og y=10.

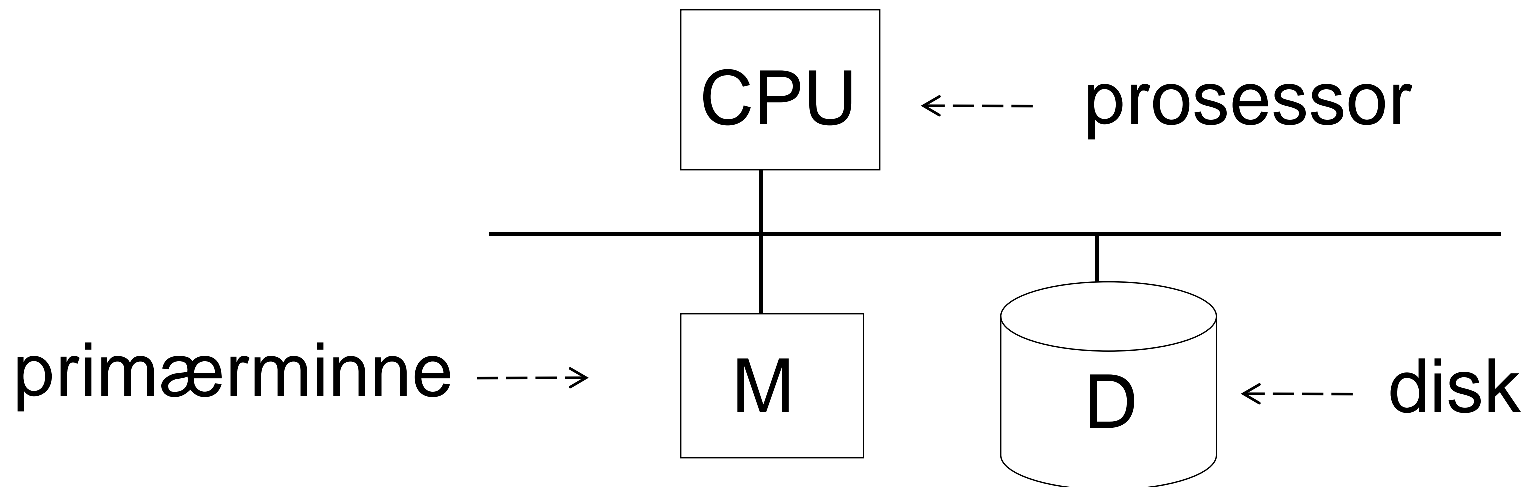
Viktige ting INF3100 *ikke* omfatter

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

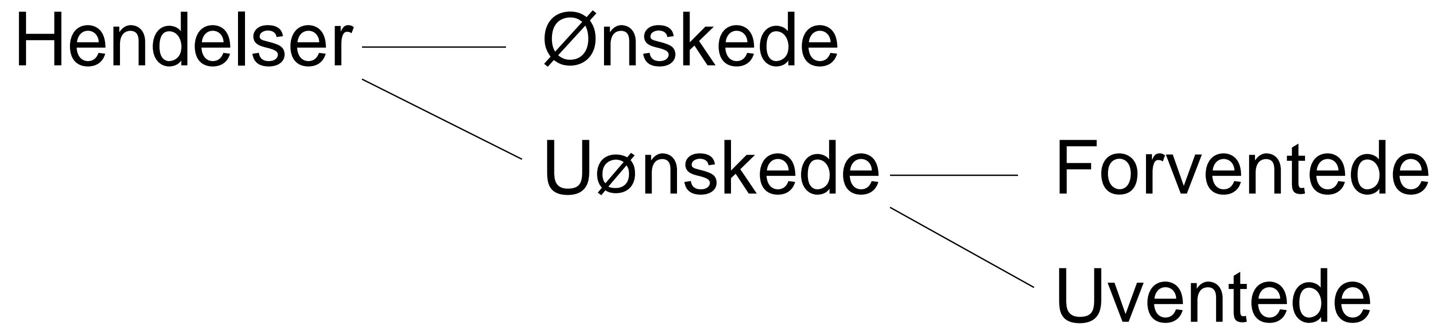
De løsningene som vi nå skal gi på håndtering av feilsituasjoner, er **uavhengige** av hvilke integritetsregler som er definert i database-skjemaet

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 ….

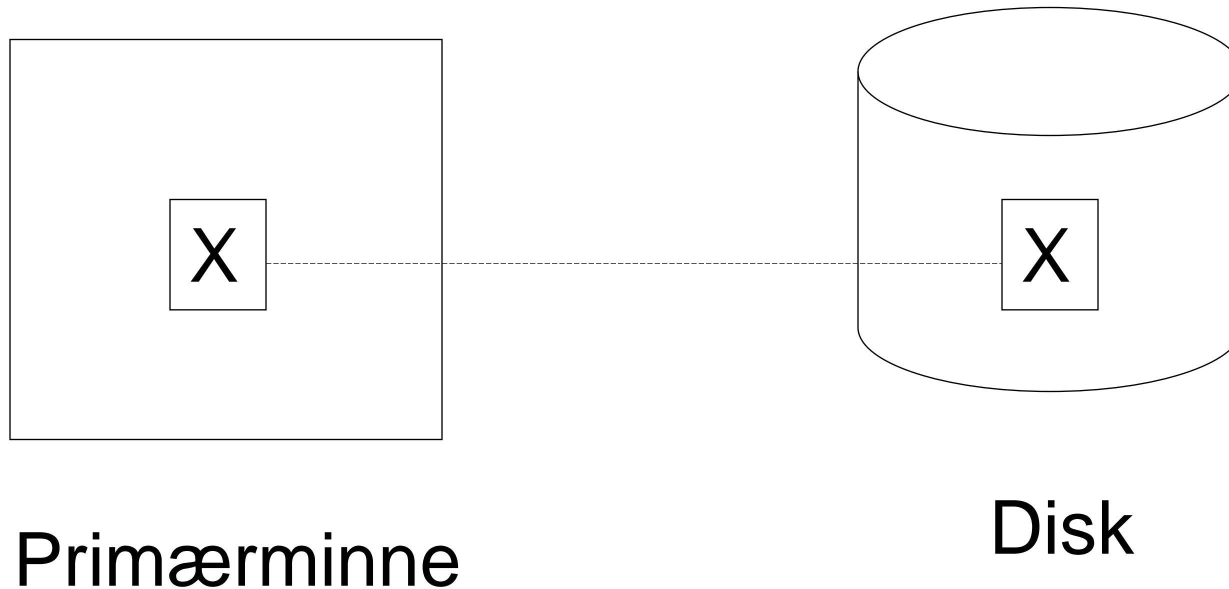
Et forslag til løsning

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

Eksempler: Replikert disk
Paritetssjekker
CPU-sjekker

En modell for beskrivelse av feil II

Lagringshierarkiet:



Primitive operasjoner i transaksjoner

- Input(x): diskblokk med x → primærminnet
- Read(x,v): utfør om nødvendig Input(x)
  verdien av x i blokken → v
- Write(x,v): utfør om nødvendig Input(x)
  v → verdien av x i blokken
- Output(x): minneblokk med x → 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

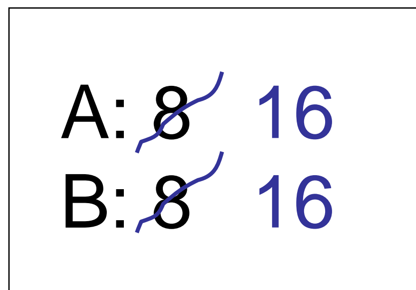
- Eksempel: Integritetsregel: $A = B$

$$
\begin{aligned}
T1: \quad & A \leftarrow A \times 2 \\
& B \leftarrow B \times 2
\end{aligned}
$$

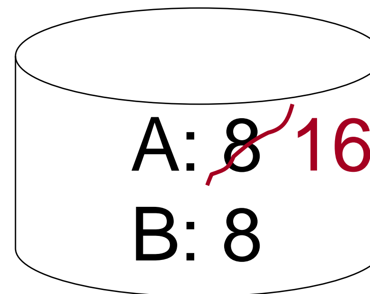
Ufullførte transaksjoner – II

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);

systemkræsj!



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

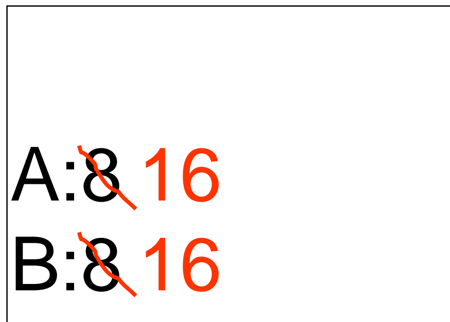
Logging

- 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

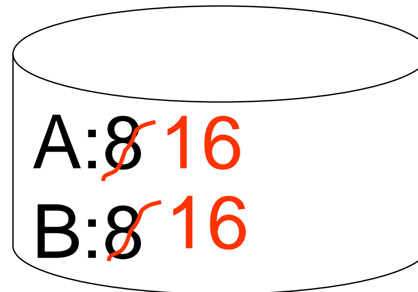
Undo-logging – I

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);

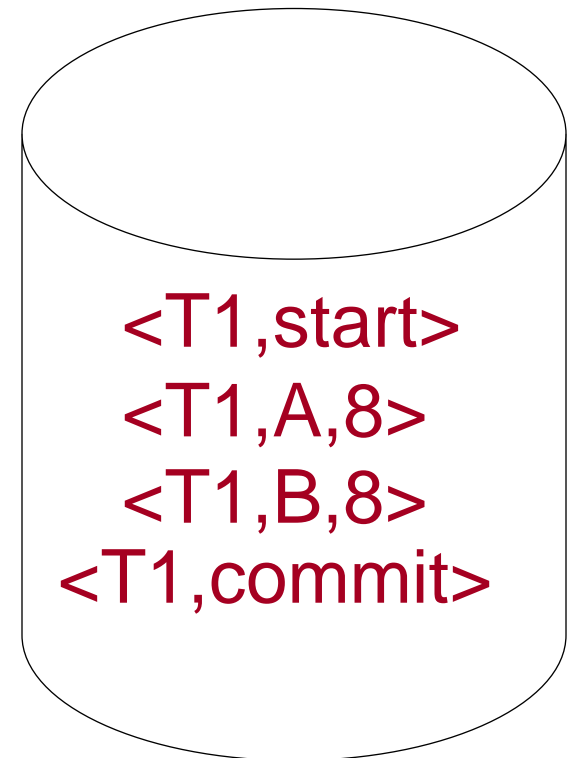
Invariant: A = B



primærminne



disk

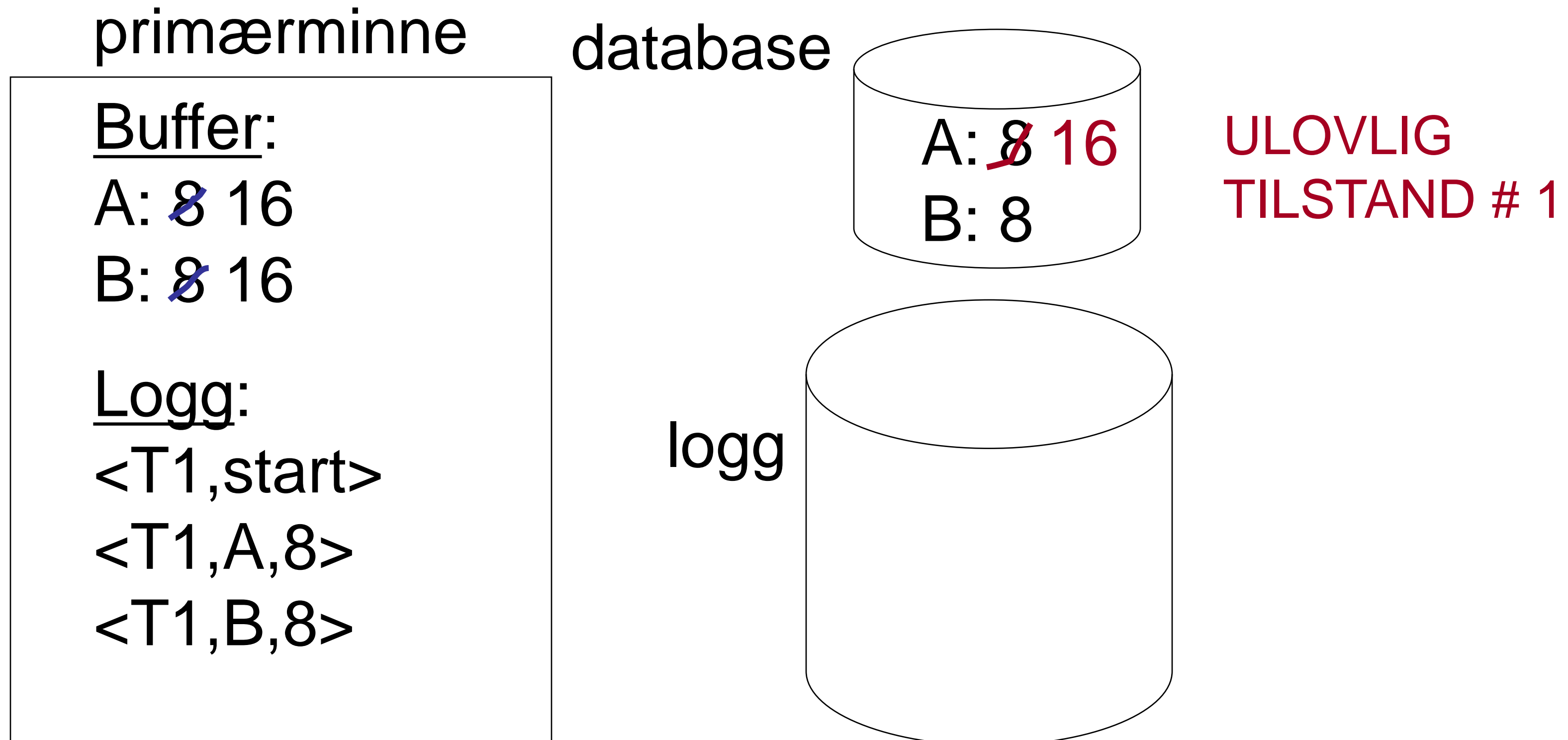


logg

Undo-logging – II

En kompliserende faktor:

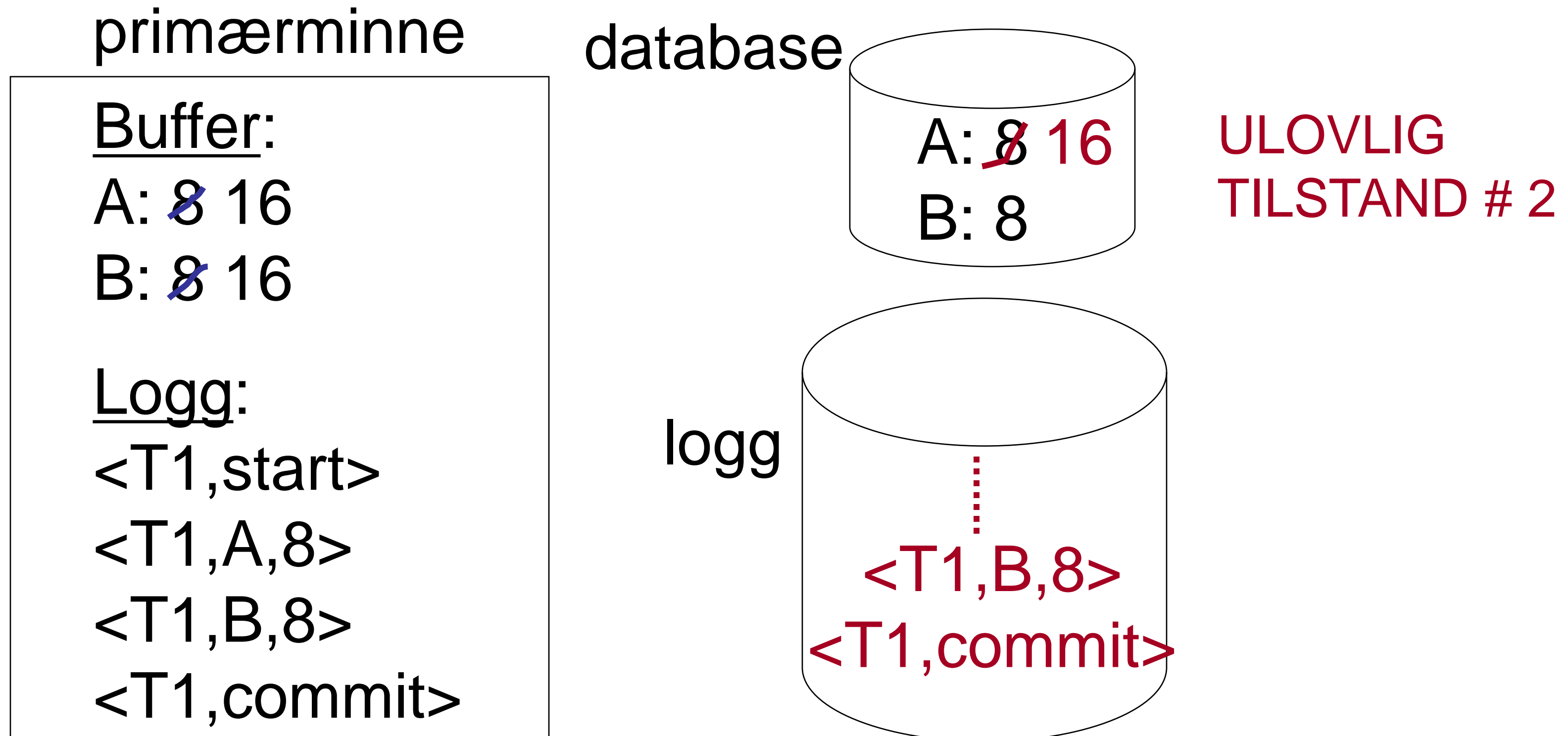
- Loggen skrives i primærminnet ***før*** den skrives til disk
- Loggen skrives ikke til disk for hver enkeltoperasjon



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 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

- <u>Algoritme for gjenoppretting</u> :
  1) Sett S = mengden av transaksjoner T_i hvor <T_i, start>, men hverken <T_i, commit> eller <T_i, abort> finnes i loggen
  2) Les loggen i bakvendt orden (fra sist til først). For hver <T_i, X, v> i loggen:

     Hvis $T_i \in S$ så

     write(X,v)

     output(X)
  3) For hver $T_i \in S$:

     Skriv <T_i, abort> i loggen

Undo-logging – VII

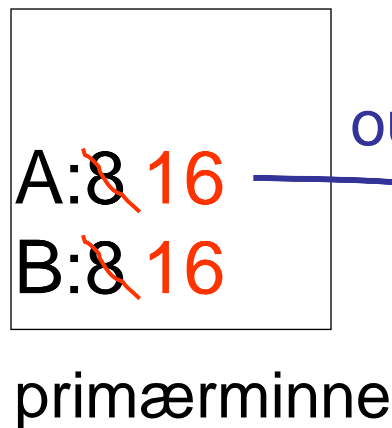
- Hva hvis det skjer en feil under gjenoppretting?

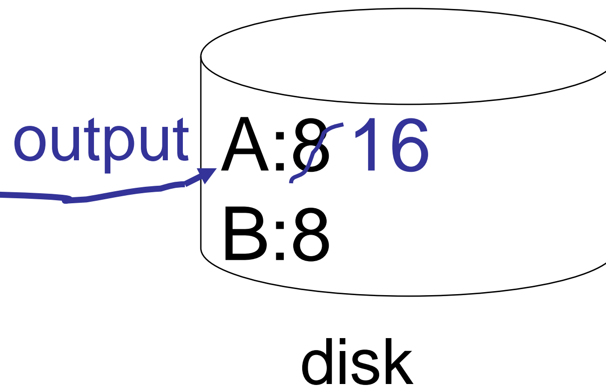
- Ikke noe problem!
  Undo er **idempotent**
  (Det betyr at å gjøre undo mange ganger gir samme resultat som å gjøre undo én gang)

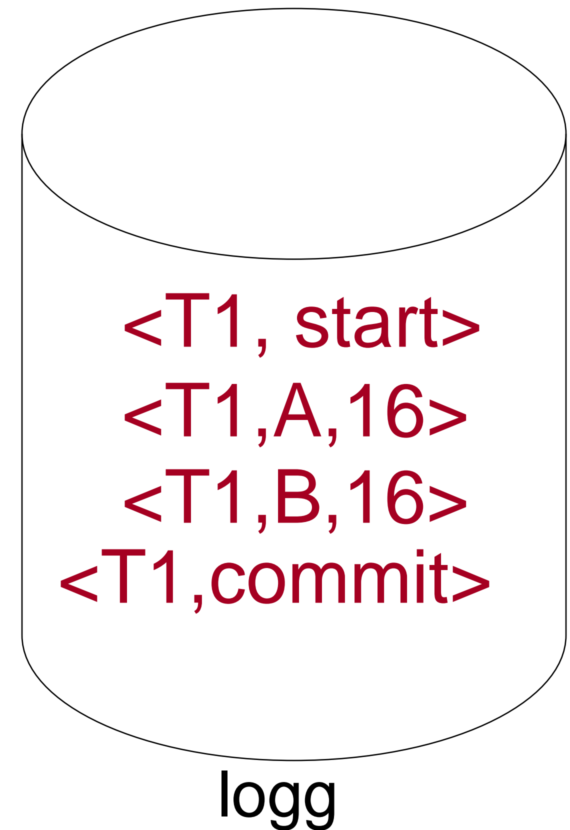
Redo-logging – I

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 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 (inklusive commit), være skrevet til disk

- Strategi:
  Skriv ikke til disk før loggen er ferdigskrevet

Redo-logging – III

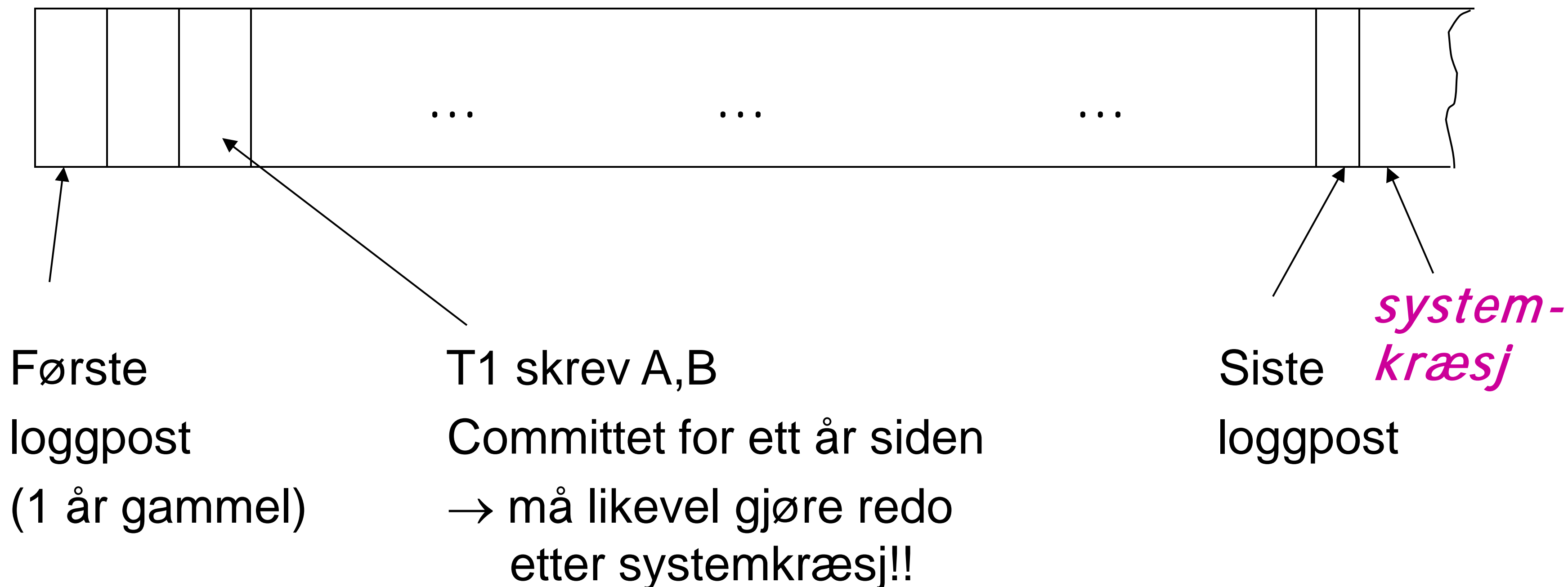
- Algoritme for gjenoppretting :
  1) Sett S = mengden av transaksjoner T_i hvor <T_i, commit> finnes i loggen
  2) Les loggen forlengs (fra først til sist).
     For hver <T_i, X, v> i loggen:
     Hvis $T_i \in$ S så
     write(X,v)
     output(X) ← ikke nødvendig

- 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>	

system-kræsj

Undo- vs redo-logging

Hovedproblemene med de to strategiene er:

Undo-logging:

- Vi må skrive data til disk umiddelbart etter at en transaksjon er ferdig
  → mer I/O enn ved redo-logging

Redo-logging:

- Vi må holde alle oppdaterte blokker i minnet helt til commit (og litt til)
  → større behov for bufferplass enn ved undo-logging

Begge:

- Dataelementene må være en hel blokk (ellers kan reglene gi konflikter, dvs. risikerer at reglene sier at en blokk både må skrives til disk umiddelbart og får ikke skrives til disk ennå)
  → økt behov for bufferplass

Undo/redo-logging – I

- Løsningen på problemet er undo/redo-logging:

  For alle skriveoperasjoner skriver vi i loggen:

  <T_i, X, gammel X-verdi, ny X-verdi>

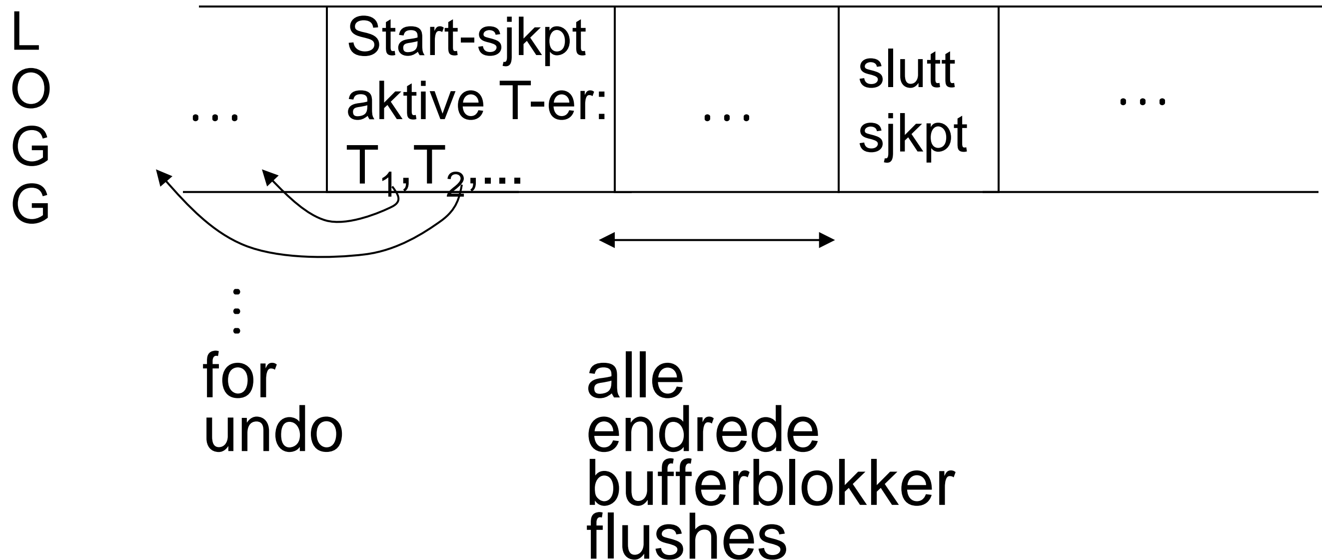
Undo/redo-logging – II

- Regler for undo/redo-logging:
  1) Ingen skriveoperasjon write(X,v) får utføres før transaksjonen vet at den ikke må abortere
  2) For hver 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

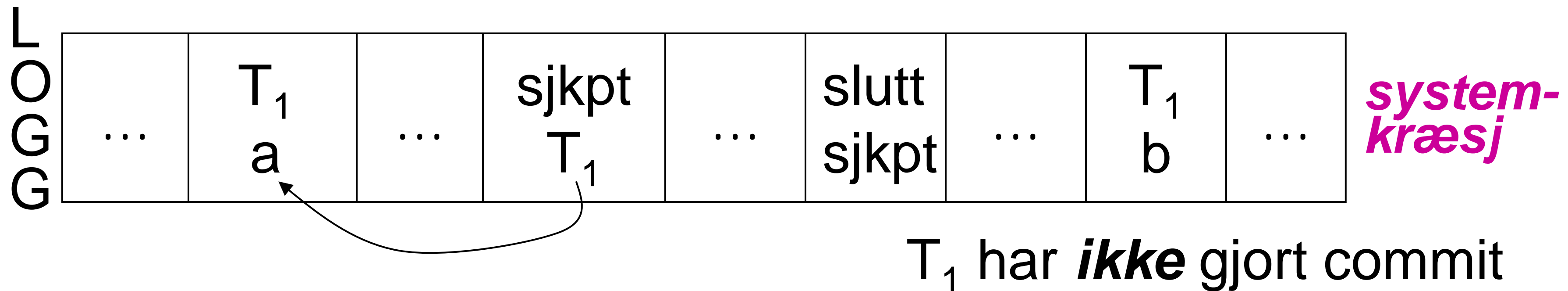
Sjekkpunkt som ikke hindrer start av nye transaksjoner:



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?

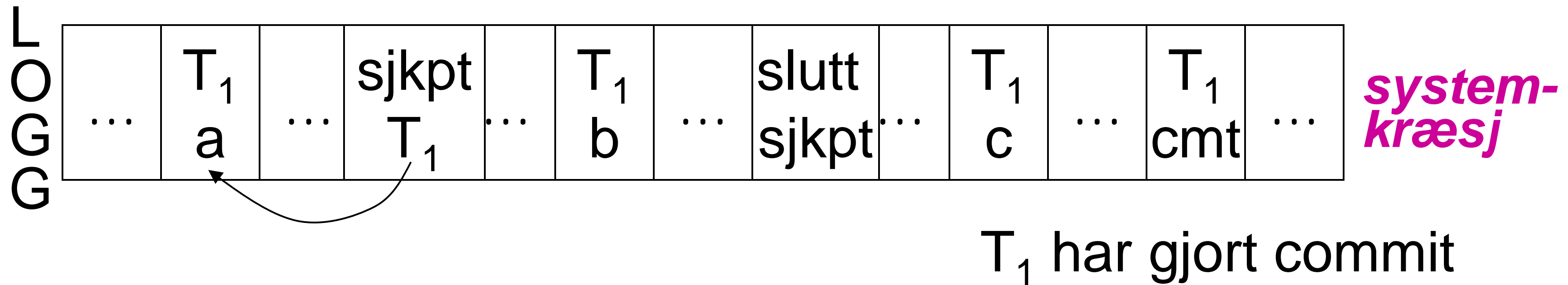


→ Undo T_1 (undo a,b)

Husk: Undo utføres fra nyeste loggpost til starten av eldste ikke-committede transaksjon

Ikke-passiviserende sjekkpunkt – III

Eksempel: Hva gjøres ved gjenoppretting? (forts.)

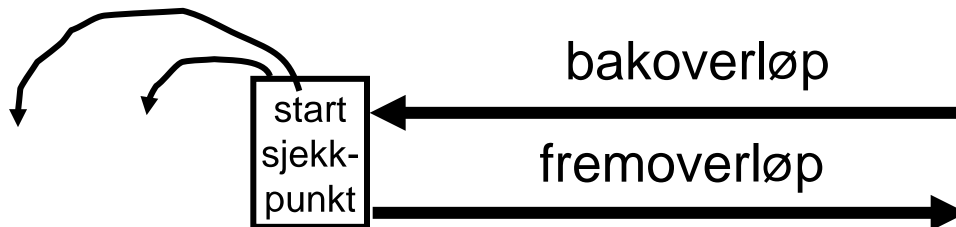


→ Redo T_1 (redo b,c)

Husk: Redo utføres fra siste sjekkpunktstart til slutten av loggen

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 aktivliste 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

$T_i = a_1 \; a_2 \; \ldots\ldots \; a_k \; \ldots\ldots \; a_n$

↓ (a_k)

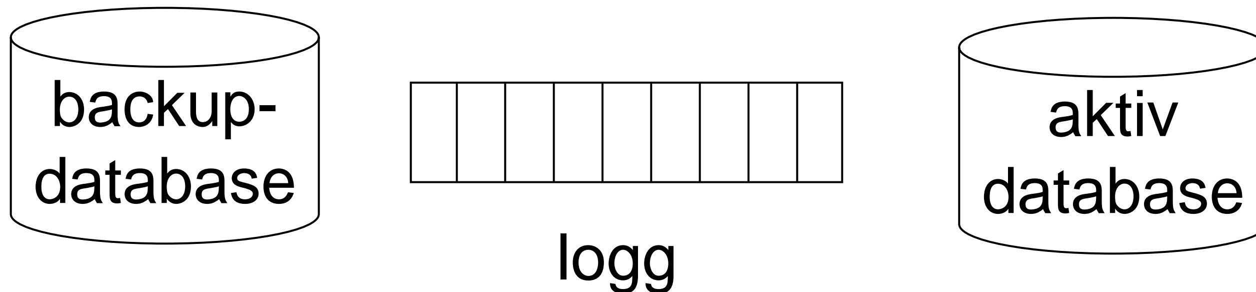
kontantuttak

Løsning:

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

Håndtering av fysiske feil

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