UNIVERSITETET
I OSLO

Transaksjonshåndtering
Del 2

En ny type serialiseringsprotokoll

- Hittil har vi bare sett på 2PL-baserte protokoller
- Alle slike protokoller har følgende struktur:
  - først setter vi låser på de dataelementene vi er interessert i
  - så leser og (eventuelt) skriver vi disse dataelementene
  - til slutt frigir vi låsene og slipper andre transaksjoner til
- Slike protokoller egner seg godt når dataene er lagret på tabellform (som arrayer og hashtabeller)
- Den andre hovedmåten å lagre data på er å organisere dem som trær (som oftest B-trær eller B^+-trær)
- For slike data finnes en mer hensiktmessig protokoll, kalt **treprotokollen**

Treprotokollen

1. En transaksjon kan sette sin første lås på en vilkårlig node i treet
2. Senere kan transaksjonen bare sette lås på en node hvis den har låst foreldrenoden
3. En transaksjon kan når som helst slette en lås den har på en node
4. En transaksjon kan ikke sette lås på en node den tidligere har frigitt
   (selv om den fortsatt har lås på foreldrenoden)

Treprotokollen er her formulert ut fra at vi bare har én låstype, men den fungerer like bra med flere.

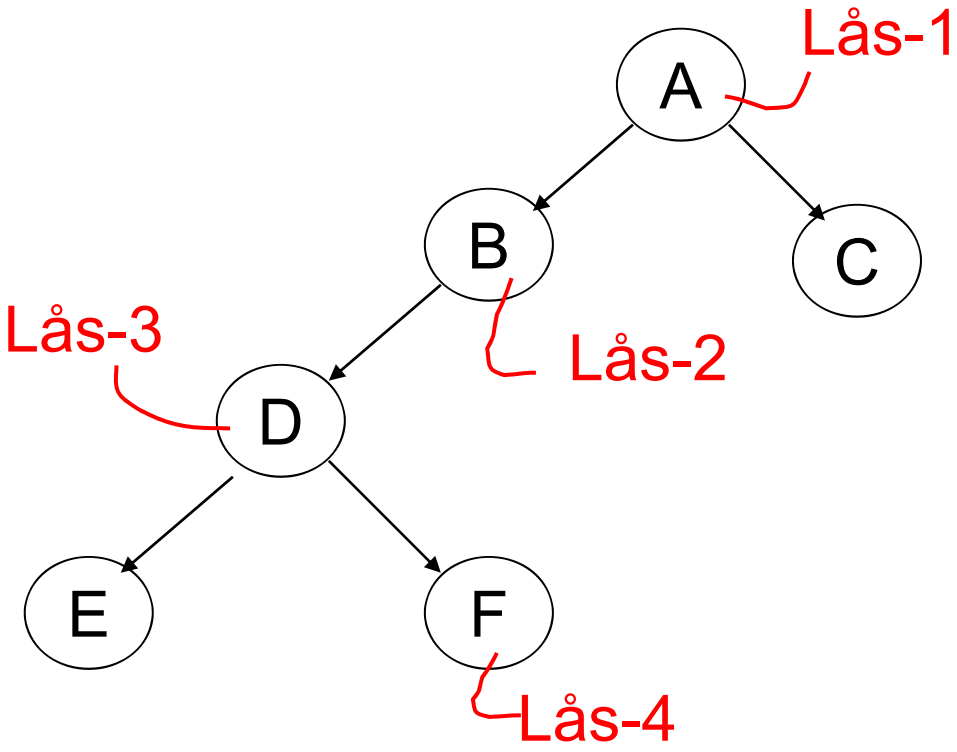
Treprotokollen på B-trær (B^+-trær)

- I et B-tre er det alltid roten som låses først
- I teorien må transaksjonen holde en lås på roten til den er ferdig fordi både insert og delete kan føre til at roten blir endret
- Det ville ha medført at bare én skrivetransaksjon om gangen kunne aksessere B-treet
- Imidlertid vil det normale være at når man aksesserer neste nivå i treet, så kan man straks fastslå at foreldrenoden ikke vil bli berørt, og den kan da øyeblikkelig frigis
- I praksis gir dette høy grad av samtidighet

Eksempel på treprotokollen

1. T låser A (lås-1)
2. T låser B (lås-2)
3. T ser at A ikke blir endret og hever lås-1
4. T låser D (lås-3)
5. T ser at B ikke blir endret og hever lås-2
6. T låser F (lås-4)
7. T ser at D vil bli endret
8. T skriver F og hever lås-4
9. T skriver D og hever lås-3

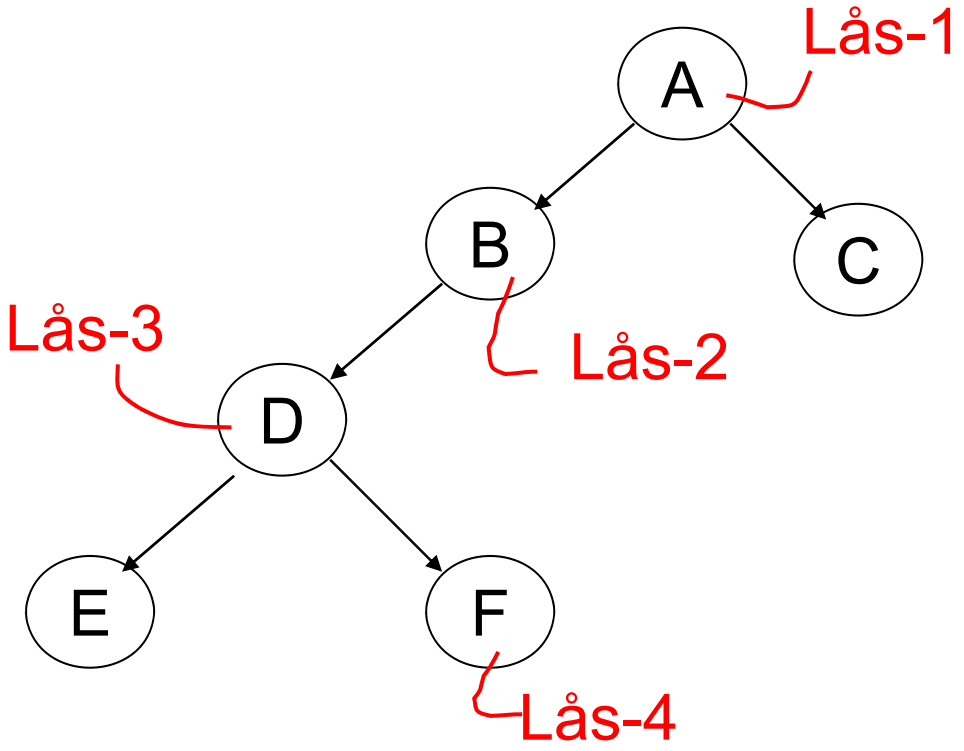
Utsnitt av et B^+-tre med rot A



Treprotokollen sikrer serialiserbarhet

- Anta at T_1 er som før, at T_2 også vil skrive, og at T_1 låser A først
- T_2 prøver å låse A og blir lagt i A-køen til T_1 slipper lås-1
- T_2 får låse A og kan gå videre til C, men ikke til B før T_1 har sluppet lås-2
- T_2 kan aldri «passere» T_1, så rekkefølgen ved roten bestemmer en serialiseringsrekkefølge

Utsnitt av et B^+-tre med rot A



Tidsstempling

- Tidsstempling er grunnlag for en familie av serialiserbarhetsprotokoller som ikke bruker låser
- Tidsstempling gir optimistisk samtidighetskontroll hvor transaksjonene får gå uhindret til man eventuelt oppdager at noe gikk galt slik at transaksjonen må aborteres (2PL er pessimistisk og prøver å hindre feil på forhånd)
- Når en transaksjon T starter, får den et **tidsstempel**, TS(T)
- To transaksjoner kan ikke ha samme tidsstempel, og hvis T_1 starter før T_2, så skal vi ha $TS(T_1) < TS(T_2)$
- Serialiseringsrekkefølgen er bestemt av tidsstemplene
- De to vanligste formene for tidsstempler er:
  - tidspunktet da T startet (verdien av systemklokken)
  - et løpenummer (transaksjonsnr i systemets levetid)

Tidsstempler på data

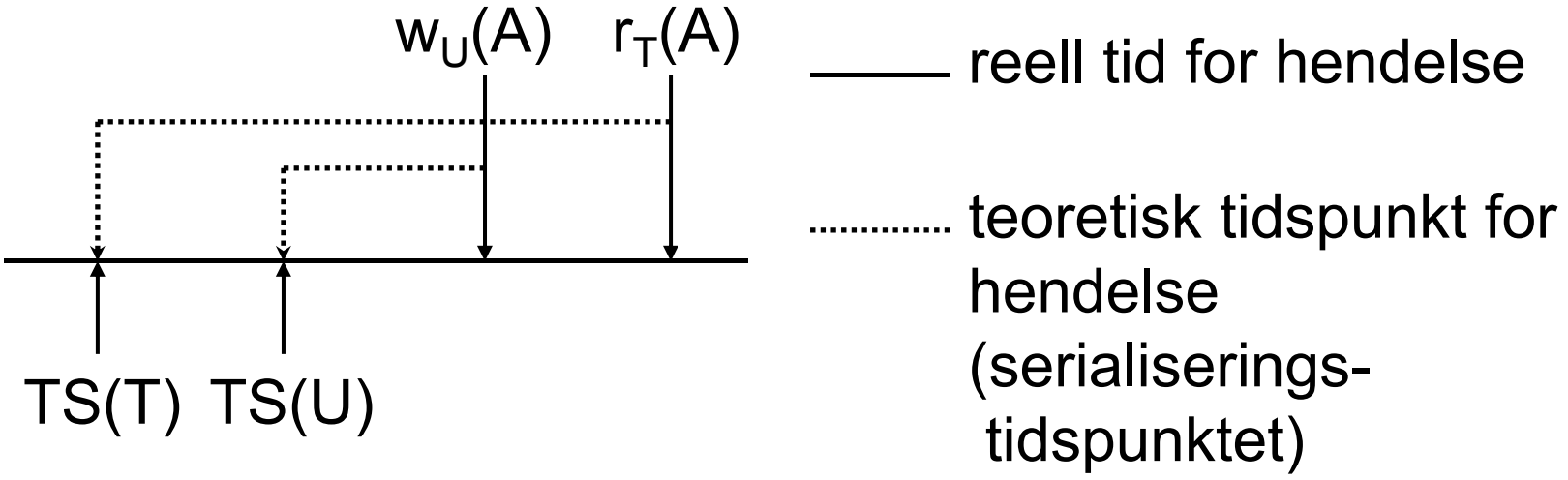
- Planleggeren (The Scheduler) må vedlikeholde en tabell over aktive transaksjoner og deres tidsstempler
- For å bruke tidsstempling til samtidighetskontroll, knyttes det to tidsstempler og en boolsk variabel til hvert eneste dataelement A i databasen:
  - RT(A) – **lesetiden til A**: Det høyeste tidsstempelet til noen transaksjon som har lest A
  - WT(A) – **skrivetiden til A**: Det høyeste tidsstempelet til noen transaksjon som har skrevet A
  - C(A) – **commit-flagget til A**: Sant hvis og bare hvis den siste transaksjonen som skrev A, er committet

Tidsstempelserialisering

- Protokollen for serialisering ved hjelp av tidsstempler har som utgangspunkt at serialiseringsrekkefølgen er bestemt av tidsstemplene
- Protokollen sikrer at vi får en eksekveringsplan som er konfliktekvivalent med den serielle planen vi hadde fått om alle transaksjonene hadde gjort alle sine lese- og skriveoperasjoner ved tidsstempelet sitt
  (det vil si dersom transaksjonene var momentane og ikke brukte noe tid fra de startet til de var ferdige)
- Det er to problemer som kan oppstå:
  - Transaksjonen leser for sent
  - Transaksjonen skriver for sent

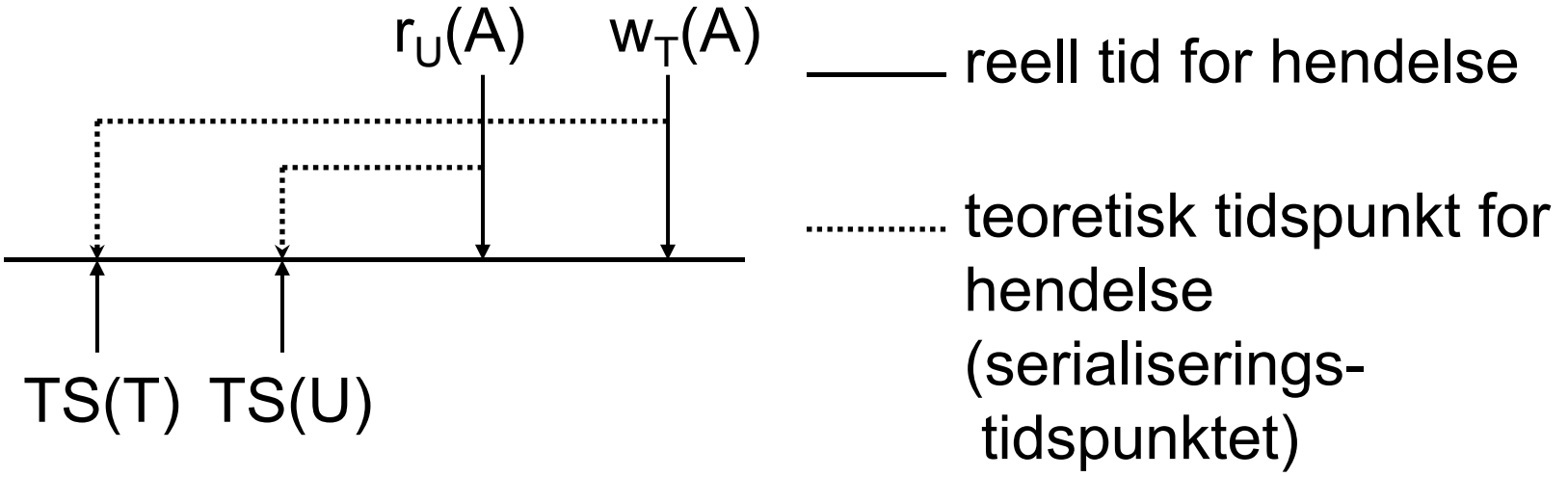
Transaksjonen leser for sent

- Transaksjon T vil lese dataelement A
- Verdien av A er skrevet av en transaksjon som startet etter T, dvs at WT(A) > TS(T)
- T ville komme til å lese «gal» verdi av A
- Konsekvens: T må abortere



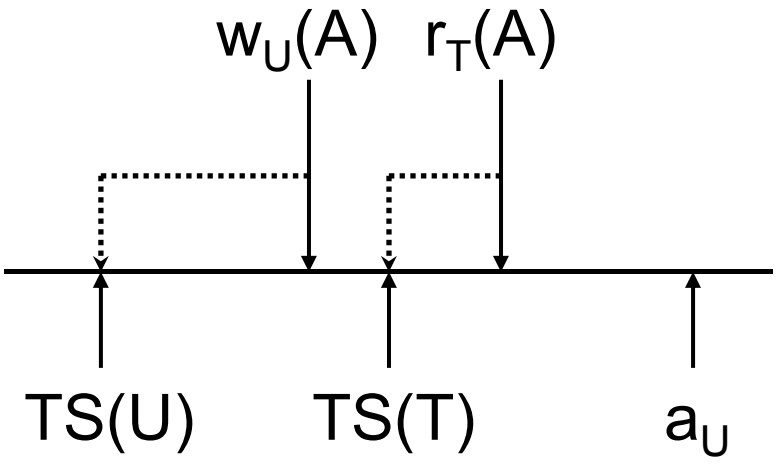
Transaksjonen skriver for sent

- T vil skrive dataelement A, men A er allerede lest av U som startet etter T, dvs at RT(A) > TS(T)
- Hvis WT(A) > TS(T), skal ikke T skrive A (alt er OK)
- Hvis ikke, skulle U ha lest den verdien av A som T nå skal skrive, og T må abortere



Skitne data

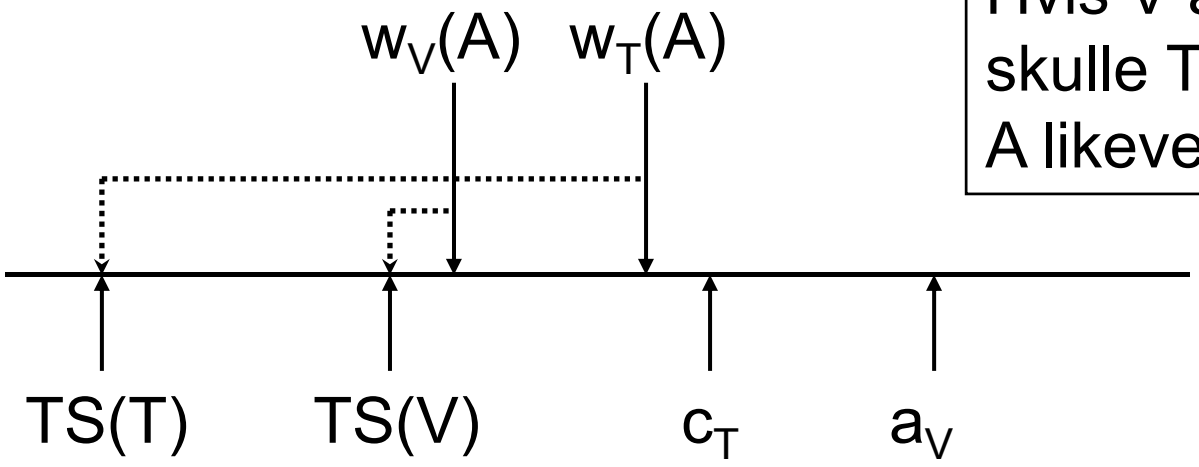
- Transaksjon T leser dataelement A som er skrevet av en annen transaksjon U
- Hvis U aborterer etter at T har lest A, har T lest en verdi av A som aldri skulle ha vært i DB
- Vi sier at T har lest en skitten verdi av A
- Dette er et brudd på isolasjonskravet, så T bør vente med å lese A til commit-flagget til A er sant



Merk!
Vi skriver
a_T for at T aborterer og
c_T for at T gjør commit

Thomas' skriveregel

- Transaksjon T ønsker å skrive dataelement A som allerede er skrevet av en annen transaksjon V med TS(V) > TS(T)
- Siden senere transaksjoner skal lese A-verdien skrevet av V, skal ikke T skrive A



Problem:

Hvis V aborterer, skulle T ha skrevet A likevel!

Tidsstempelserialiseringsprotokollen–1

1. T ønsker å lese A
   a) Hvis TS(T) ≥ WT(A), kan T få lov til å lese A:
      I. Hvis C(A), så utfør $r_T(A)$
         og sett RT(A) = max(RT(A),TS(T))
      II. Hvis ikke C(A), la T vente i en kø på A
         (Når T vekkes opp, må den teste på nytt)
   b) Hvis TS(T) < WT(A), er det fysisk umulig for T å lese (riktig verdi av) A
      T må rulles tilbake (vi må utføre rollback(T)):
      I. Utfør a_T
      II. Start på nytt med et høyere tidsstempel

Tidsstempelserialiseringsprotokollen–2

2 T ønsker å skrive A

a) Hvis TS(T) ≥ RT(A) og TS(T) ≥ WT(A), får T skrive:

I. Utfør $w_T(A)$, sett WT(A) = TS(T) og C(A) = falsk

II. Signaliser til de som venter

b) Hvis TS(T) ≥ RT(A) og TS(T) < WT(A),
er A skrevet av en nyere transaksjon enn T
Hvis C(A), ignorer ønsket (Thomas' skriveregel)
Hvis ikke C(A), la T vente i A-køen
(Test på nytt når T vekkes)

c) Hvis TS(T) < RT(A), er A lest av en nyere transaksjon enn T, og T må rulles tilbake

Tidsstempelserialiseringsprotokollen–3

3 T ønsker å committe

a) Utfør c_T (dvs. skriv commit(T) i loggen)

b) For hver A hvor T var siste transaksjon som skrev A, sett C(A) = sann

c) La transaksjoner som venter på å lese eller skrive en slik A, få fortsette

4 T blir abortert (eller ønsker selv å abortere)

a) Bruk loggen til å gjøre Undo på alle skrivinger foretatt av T og skriv abort(T) i loggen

b) La alle transaksjoner som venter på å lese eller skrive et dataelement skrevet av T, forsøke på nytt

Versjonering

- Metode for å unngå abort pga at T vil lese en A skrevet av en nyere transaksjon
- Når en skriveoperasjon $w_T(A)$ utføres, blir det laget en ny versjon A_t av A, der t = TS(T)
- Når en leseoperasjon $r_T(A)$ utføres, leses den versjonen A_t som har den største $t \leq TS(T)$
- Foreldede versjoner av A kan, og bør, slettes
- For at A_t skal kunne slettes, må det finnes en versjon A_u der $t < u$ og der $u \leq TS(T)$ for alle aktive transaksjoner T
- En effektiv implementasjon av versjonering krever i praksis at dataelementene er blokker

Snapshot Isolation (SI)

- I kommersielle DBMSer brukes en form for versjonering som kalles «Snapshot Isolation», forkortet SI
- I SI vil en transaksjon T i hele sin levetid lese de committede verdiene databasen hadde ved TS(T)
- T blir altså ikke påvirket av skriveoperasjoner som andre transaksjoner gjør etter TS(T)
- SI sikrer ikke serialiserbarhet
- Men SI gir høy grad av fletting og er standard-strategien for samtidighetskontroll i de fleste av dagens kommersielle DBMSer

Tidsstempling vs låsing

- Tidsstempling er best hvis de fleste transaksjonene bare leser, eller hvis konflikter er sjeldne
- Hvis konflikter er vanlige, vil tidsstempling føre til mange tilbakerullinger, og låsing er bedre
- Noen kommersielle systemer tilbyr et kompromiss:
  - Lese/skrive-transaksjoner bruker 2PL
  - Rene lesetransaksjoner bruker multiversjon tidsstempling

  Dette gjør at lesetransaksjoner aldri må abortere og sjelden må vente

Validering

- Validering er en optimistisk serialiseringsstrategi som baserer seg på tidsstempling
- Den skiller seg fra vanlig tidsstempling ved at man ikke lagrer lese- og skrivetidsstempel for alle dataelementene i databasen
- For hver aktiv transaksjon T lagres to mengder
  - **lesemengden** til T, RS(T)
  - **skrivemengden** til T, WS(T)

  som inneholder alle dataelementer som T henholdsvis leser og skriver

Validering (forts.)

Utførelsen av en transaksjon T deles inn i tre faser:

1. ***Lesefasen***
   All lesing og beregning gjøres her
   RS(T) og WS(T) bygges opp i Ts adresserom
   Starttidspunktet for lesefasen kalles Start(T)
2. ***Valideringsfasen***
   T valideres ved å sammenligne RS(T) og WS(T) med lese- og skrivemengdene til andre transaksjoner (detaljene kommer snart)
   Hvis valideringen feiler, rulles T tilbake
   Sluttidspunktet for valideringsfasen kalles Val(T)
3. ***Skrivefasen***
   Her skriver T verdiene i WS(T) til databasen
   Sluttidspunktet for Skrivefasen kalles Fin(T)

Verdien av Val(T) bestemmer serialiseringsrekkefølgen

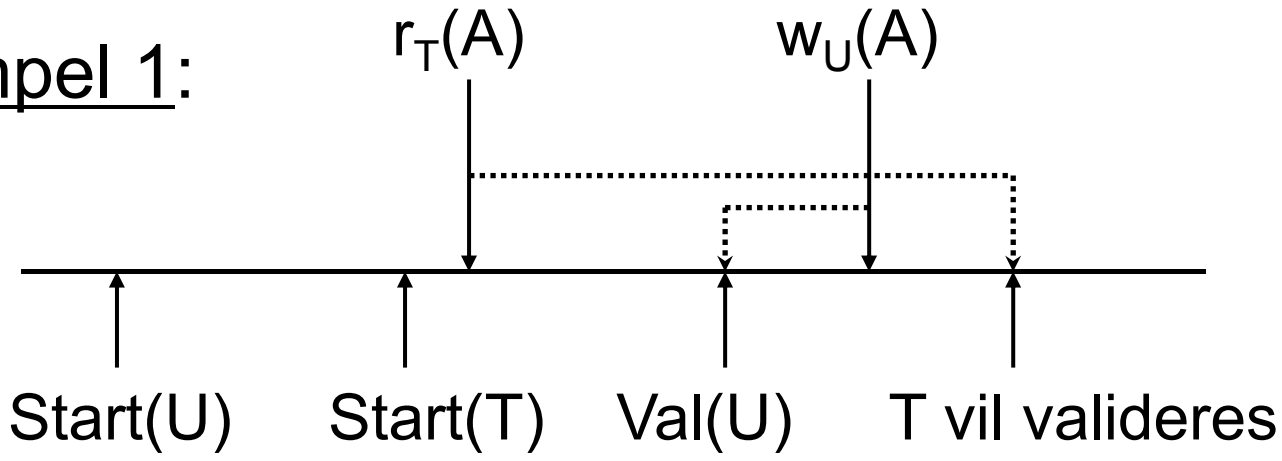
Validering (forts.)

- Planleggeren vedlikeholder tre mengder med transaksjoner:

1. **START**
   De som har startet, men ennå ikke har avsluttet valideringsfasen
   For hver $T \in$ START lagres Start(T)
2. **VAL**
   De som er validert, men ikke har avsluttet skrivefasen
   For hver $T \in$ VAL lagres Start(T) og Val(T)
3. **FIN**
   De som (nylig) har avsluttet skrivefasen
   For hver $T \in$ FIN lagres Start(T), Val(T) og Fin(T)
   T kan fjernes fra FIN når vi for alle $U \in$ START $\cup$ VAL har at Start(U) > Fin(T)

Valideringsfasen

Eksempel 1:



Anta at idet T vil valideres, så finnes en U slik at

a) $U \in VAL \cup FIN$
(dvs at U er validert)

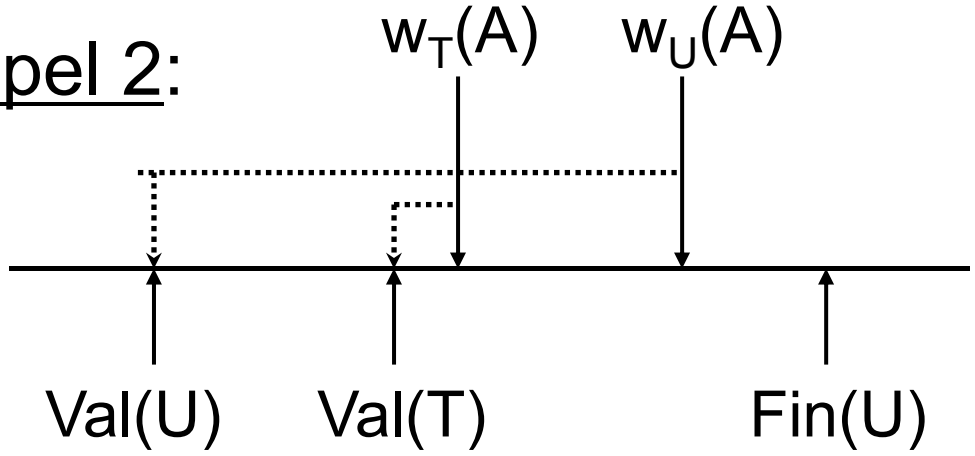
b) Hvis $U \in FIN$: Fin(U) > Start(T)
(dvs at U ikke var avsluttet da T startet)

c) $RS(T) \cap WS(U) \neq \emptyset$ (på figuren er $A \in RS(T) \cap WS(U)$)

T må rulles tilbake fordi T kan ha lest en «gal» verdi av A

Valideringsfasen (forts.)

Eksempel 2:



Anta at idet T vil valideres, så finnes en U slik at

a) $U \in VAL$ (dvs at U er validert, men ikke avsluttet)

b) $WS(T) \cap WS(U) \neq \emptyset$ (på figuren er $A \in WS(T) \cap WS(U)$)

T må rulles tilbake da T kan komme til å skrive A før U gjør det

Valideringstesten

- De to foregående eksemplene dekker alle mulige feilsituasjoner
- Dermed består valideringsfasen av følgende to tester:
  - Sjekk at $RS(T) \cap WS(U) = \emptyset$ for alle $U \in VAL$ og alle $U \in FIN$ med $Fin(U) > Start(T)$
  - Sjekk at $WS(T) \cap WS(U) = \emptyset$ for alle $U \in VAL$
- Hvis T består begge testene, er T validert og går inn i skrivefasen
- Hvis ikke, må T rulles tilbake

Kaskadetilbakerulling

T_1	T_2	Integritetsregel: A=B	A	B
			25	25
$l_1(A)$; $r_1(A)$; A←A+100;				
$w_1(A)$; $l_1(B)$; $u_1(A)$;			125	
	$l_2(A)$; $r_2(A)$;			
	A←Ax2; $w_2(A)$;		250	
	$l_2(B)$; ***Avslått***			
$r_1(B)$;				
a_1; $u_1(B)$;				
	$l_2(B)$; $u_2(A)$; $r_2(B)$;			
	B←Bx2; $w_2(B)$; $u_2(B)$;			50
			250	50

Når T_1 aborterer, sletter planleggeren alle låsene T_1 har
Hvis T_2 får fortsette, vil T_2 lage en inkonsistent tilstand
Altså må T_2 rulles tilbake (fordi T_2 har lest en skitten A)

Kaskadetilbakerulling (forts.)

- Kaskadetilbakerulling kan, som navnet antyder, være rekursiv:
  Abort av T_1 kan føre til abort av T_2 som kan føre til abort av T_3 osv
- Kaskadetilbakerulling kan omfatte committede transaksjoner, noe som er i strid med D i ACID
- Kaskadetilbakerulling bør derfor unngås
- Tidsstempelprotokoller med commit-flagg sikrer mot kaskadetilbakerulling
- Validering sikrer også mot kaskadetilbakerulling (ingen skriving gjøres før man vet at det ikke blir noen abort)

Gjenopprettbare eksekveringsplaner

- En eksekveringsplan er **gjenopprettbar** hvis ingen transaksjon T gjør commit før alle transaksjoner som har skrevet data som T har lest, har gjort commit
- Eksempel 1: En gjenopprettbar serialiserbar (seriell) plan:

  S_1: $w_1(A);w_1(B);w_2(A);r_2(B);c_1;c_2$
- Eksempel 2: En gjenopprettbar ikke-konfliktserialiserbar plan:

  S_2: $w_2(A);w_1(B);w_1(A);r_2(B);c_1;c_2$
- Eksempel 3: En serialiserbar, ikke gjenopprettbar, plan:

  S_3: $w_1(A);w_1(B);w_2(A);r_2(B);c_2;c_1$

Gjenopprettbare eksekveringsplaner (forts.)

- For at en eksekveringsplan skal være gjenopprettbar både for undo-, redo- og undo/redo-logging, må loggens commit-poster skrives til disk i samme rekkefølge som de skrives i loggen (flere loggposter kan ligge i samme blokk og bli skrevet samtidig)

ACR-planer

- En eksekveringsplan unngår kaskadetilbakerullinger hvis transaksjonene bare kan lese data skrevet av committede transaksjoner
- Slike planer kalles **ACR-planer**
  (ACR = Avoid Cascade Rollback)
- Alle ACR-planer er gjenopprettbare

  Bevis: Anta at vi har en ACR-plan.

  Anta at T_2 leser en verdi skrevet av T_1 etter at T_1 har gjort commit

  Siden T_2 hverken har gjort commit eller abort før den leser, må T_2 gjøre sin commit eller abort etter at T_1 gjorde sin commit

  qed

Strikt låsing

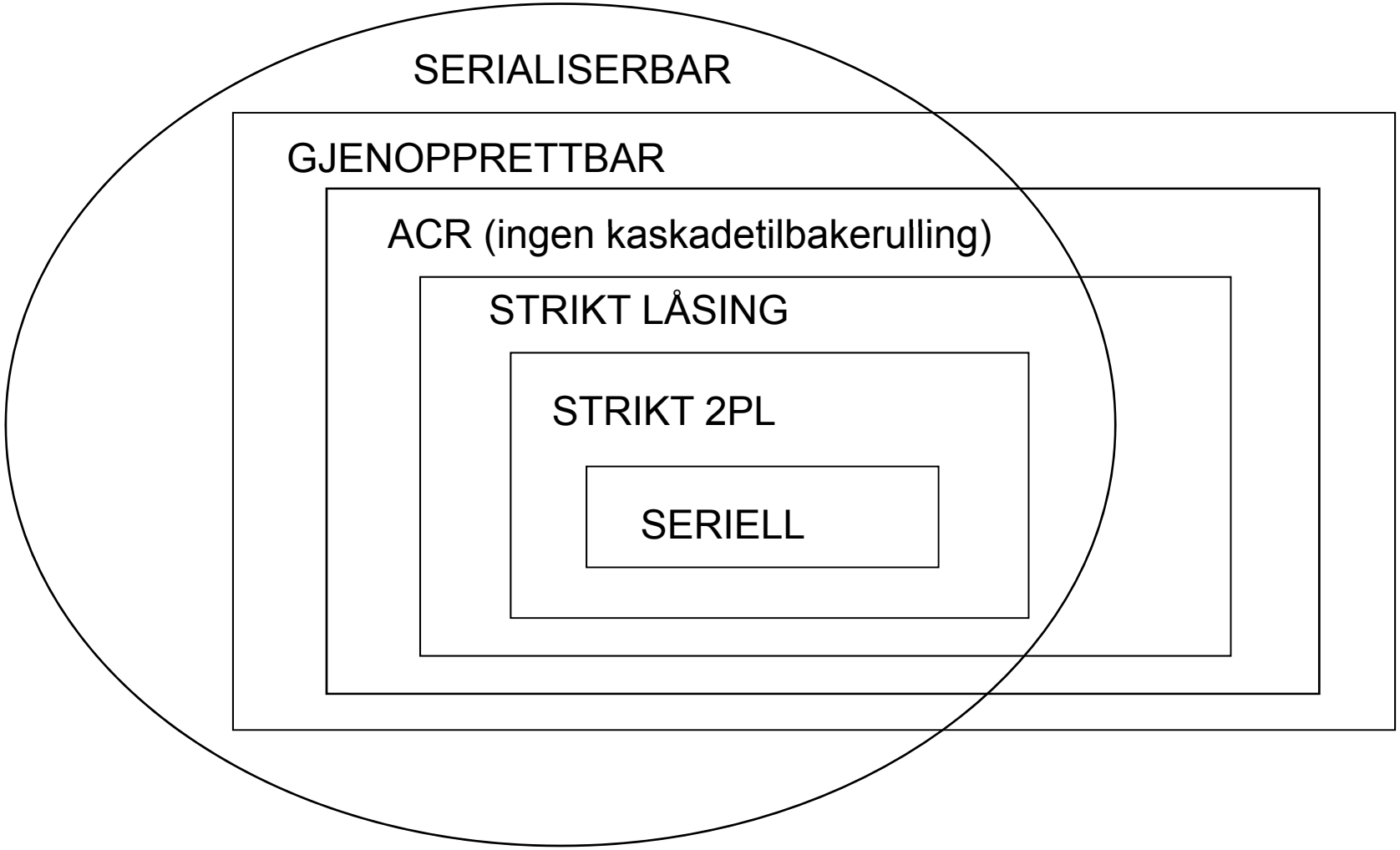
- En eksekveringsplan bruker **strikt låsing** hvis den er basert på låser og overholder følgende regel:
  Strikt låseregel: En transaksjon kan ikke frigi noen skrivelås før den har gjort commit eller abort og commit- eller abort-loggposten er skrevet til disk
- En eksekveringsplan som bruker tofaselåsing og først slipper skrivelåsene etter commit/abort, kalles **strikt 2PL**. Leselåsene kan slippes når som helst i krympefasen.

Strikt låsing (forts.)

- En eksekveringsplan med strikt låsing er en ACR-plan

  Bevis: Fordi skrivelåsene ikke blir frigitt før etter at transaksjonene har gjort commit, kan ingen lese data skrevet av en transaksjon som ikke har gjort commit
- Vi skal se under gjennomgangen av snapshot isolation at strikt låsing ikke er tilstrekkelig til å sikre serialiserbarhet

Eksekveringsplantyper



Tilbakerulling ved bruk av låser

- Hvis dataelementene er blokker, er alt enkelt:

  All skriving gjøres i bufferet;
  intet skrives til disk før commit

  Ved abort frigis blokken, som blir ubrukt bufferareale

  Samme teknikk virker ved versjonering;
  blokken med «abortert versjon» frigis
- Hvis det er flere dataelementer i hver blokk, er det tre måter å restaurere data på etter en abort
  1. Originalen kan leses fra databasen på disk
  2. Med en undo- eller undo/redo-logg kan originalen hentes fra loggen
  3. Hver aktiv transaksjon kan ha sin egen logg over sine endringer i hukommelsen

Group commit

- Ved group commit kan låser slippes tidligere enn ved strikt låseregel
- **Group commit**:
  - En transaksjon kan ikke frigi noen skrivelås før den har gjort commit eller abort og commit- eller abort-loggposten er skrevet til primærminnet
  - Loggblokker må flushes til loggdisken i den rekkefølgen de blir fylt opp i primærminnet

Group commit (forts.)

- Group commit gir gjenopprettbare planer
  Bevis: Anta at T_1 skriver X og committer, og at T_2 leser X. T_2 kan bare lese skitne data hvis systemet går ned før T_1s commitrecord er på disk. Men i såfall er ikke T_2s commitrecord på disk heller, så både T_1 og T_2 aborteres av Recoverymanageren. Dermed hindres lesing av skitne data
- Group commit gir serialiserbare planer
  Bevis: Planene er konfliktekvivalente med de serielle planene vi får ved å la hver transaksjon bli utført ved committidspunktet

Logisk logging

- Dette er en loggtype som benytter seg av transaksjons-logikken til å foreta tilbakerullinger
- Typiske logiske loggposter består av fire felt
  - L – et loggpostløpenummer
  - T – transaksjonens id
  - A – aksjon (operasjon) utført (f.eks. insert tuppel t)
  - B – blokken hvor A ble utført
- For hver aksjon finnes en kompenserende aksjon som opphever virkningen (f.eks. delete for insert) og som kan konstrueres utifra A
- Hvis T aborterer, blir alle Ts aksjoner kompensert, og kompensasjonene blir loggført
- Hver blokk har loggpostnummeret på siste aksjon som berørte den