



UNIVERSITETET  
I OSLO



# Transaksjonshåndtering

## Del 3

# Samtidighetsfenomener og -anomalier

Dette er uønskede «merkverdigheter» som kan inntreffe i eksekveringsplaner.

Vi har to typer av slike «merkverdigheter»:

- **Samtidighetsfenomener**  
(disse merkes med P for Phenomena).  
Fenomener *kan* gi opphav til feilsitasjoner.
- **Samtidighetsanomalier**  
(disse merkes med A for Anomalies).  
Anomalier vil *alltid* føre til feilsituasjoner.

# Liste over fenomener

- P0 – Skitten skriv  $w_1(x)..w_2(x)..(c_1 \text{ eller } a_1)$
- P1 – Skitten les  $w_1(x)..r_2(x)..(c_1 \text{ eller } a_1)$
- P2 – Ikke-repeterbar les  $r_1(x)..w_2(x)..(c_1 \text{ eller } a_1)$
- P3 – Fantomfenomen  $r_1(Q)..w_2(y \text{ i } Q)..(c_1 \text{ eller } a_1)$
- P4 – Tapt oppdatering  $r_1(x)..w_2(x)..w_1(x)..c_1$

I P3 står  $Q$  for et predikat, dvs. svaret på et where-uttrykk.  $w_2(y \text{ i } Q)$  betyr at skriveoperasjonen  $w_2(y)$  kan øke resultatmengden til  $Q$ .

# Liste over anomalier

- A3A – Fantomleseanomali  
 $r_1(Q)..w_2(y \text{ i } Q)..c_2..r_1(Q)..c_1$
- A3B – Fantomskjevskrivning  
 $r_1(Q)..r_2(Q)..w_1(y \text{ i } Q)..w_2(z \text{ i } Q)..(c_1 \text{ og } c_2)$
- A5A – Skjevlesing  
 $r_1(x)..w_2(x)..w_2(y)..c_2..r_1(y)..c_1$
- A5B – Skjevskrivning  
 $r_1(x)..r_2(y)..w_1(y)..w_2(x)..(c_1 \text{ og } c_2)$
- A6 – Lesetransaksjonsanomali  
 $r_2(x)..r_2(y)..w_1(y)..c_1..r_3(x)..r_3(y)..c_3..w_2(x)..c_2$

# SQL-isolasjonsnivåer

- Isolasjonsnivåer ble innført med SQL-92 standarden.
- Rangert fra det sterkeste til det svakeste er de:
  - Serializable  
Ingen fenomener eller anomalier er tillatt i noen plan (planer skal gi samme resultat som en seriell plan).
  - Repeatable Read  
Bare fantomer er tillatt.
  - Read Committed  
Alle fenomener og anomalier er tillatt med unntak av skitten skriv og skitten les.
  - Read Uncommitted  
Bare skitten skriv er forbudt.

# Monotoni

- La  $S$  være en plan, og la  $T$  være en delmengde av transaksjonene i  $S$ .
- Da definerer vi projeksjonen av  $S$  på  $T$  som den planen vi får hvis vi fra  $S$  fjerner alle operasjoner utført av transaksjoner som ikke ligger i  $T$ .
- En klasse med planer kalles *monoton* hvis alle projeksjoner av planer i klassen selv ligger i klassen.

# Monotoni og planleggere

- La  $E$  være klassen av planer som en gitt planlegger (scheduler)  $\Sigma$  kan lage ( $E$  er klassen av lovlige planer).
- Hvis  $E$  ikke er monoton, kan følgende skje:
  - $\Sigma$  lager en plan  $P$  for en mengde transaksjoner  $T$ .
  - En av transaksjonene i  $T$  aborterer.
  - Projeksjonen av  $P$  på resten av transaksjonene i  $T$  er ikke i  $E$  (det betyr at de danner en ulovlig plan).
- En annen merkverdighet er at en ulovlig plan kan bli lovlig hvis det kommer en ny transaksjon som skal flettes inn i planen.

# Monotoni og planleggere (forts.)

- Eksempelet på forrige lysark viser at det i praksis er (nesten) umulig å lage en fornuftig planlegger for en klasse planer som ikke er monoton.
- Det er derfor viktig å sjekke om en klasse er monoton før man prøver å lage en planlegger for den.
- Planleggere bruker projeksjoner til å håndtere aborter: Når en eller flere transaksjoner i en plan aborterer, erstattes planen av sin projeksjon på de ikke-aborterte transaksjonene i planen.



# Klassen av konfliktserialiserbare planer er monoton

- Dette er en konsekvens av teoremet som sier at en plan er konfliktserialiserbar hvis og bare hvis presedensgrafen er asyklisk.
- Begrunnelse:
- Anta at  $P$  er en konfliktserialiserbar plan, dvs. at  $P$  har en asyklisk presedensgraf.
- Presedensgrafen til enhver projeksjon av  $P$  vil være en subgraf i  $P$ s presedensgraf, og slike subgrafer er også asykliske.
- Altså er alle projeksjoner av  $P$  konfliktserialiserbare.

# Multiversjonsdatabaser (repetisjon)

- Noen DBMSer kan lagre flere versjoner av hvert dataelement.
- Dette forutsetter at transaksjonene får et tidsstempel (transaksjonsnummer) når de starter.
- Når en transaksjon  $t_k$  (der  $k$  er transaksjonsnummeret) skriver en ny verdi i et element  $x$ , blir det dannet et nytt element  $x_k$  (den gamle verdien av  $x$  blir ikke overskrevet).
- Vi tenker oss at initialtilstanden er skrevet av en fiktiv committet transaksjon  $t_0$ , dvs. at  $x_0$  er initialverdien av  $x$ .
- Siden det kan bli mange versjoner av hvert element, må det finnes en prosess som sletter gamle versjoner som ingen lenger kan få bruk for (søppeltømming).

# Snapshot Isolation

- Snapshot Isolation er en effektiv og populær protokoll som lager multiversjonsplaner.
- Den ble lansert av Borland i InterBase 4 (1995).
- Snapshot Isolation brukes i flere DBMSer, bl.a.
  - Oracle
  - PostgreSQL
  - Microsoft SQL Server
- Vi lar SI betegne klassen av planer som kan genereres av Snapshot Isolation.

# SI-protokollen

- SI-protokollen består i å håndheve følgende to regler:
  1. Når en transaksjon  $T$  leser et element  $x$ , så leser  $T$  den nyeste versjonen av  $x$  som er skrevet av en transaksjon som committet før  $T$  startet.
  2. Skrivemengden til to samtidige transaksjoner må være disjunkte.
- Regel 2 betyr at hvis  $T_1$  og  $T_2$  er to transaksjoner hvor  $T_1$  starter før  $T_2$ , og  $T_1$  gjør commit etter at  $T_2$  er startet, så får ikke  $T_1$  og  $T_2$  skrive samme element.
- Det finnes flere metoder for å håndheve regel 2
  - én er å sammenligne skrivemengdene ved commit.

# Første oppdaterer vinner (FUW)

- Oracle håndhever regel 2 slik at første oppdaterer vinner:
- Anta at to transaksjoner  $T_1$  og  $T_2$  er samtidige, at  $T_1$  skriver  $x$ , og at  $T_2$  også vil skrive  $x$ .
- Da kan ikke  $T_2$  skrive  $x$  før  $T_1$  slipper sin skrivelås på  $x$ .
- Det er da tre muligheter:
  - Hvis  $T_2$  står i kø for å skrive  $x$ , og  $T_1$  gjør commit, blir  $T_2$  øyeblikkelig abortert.
  - Hvis  $T_1$  gjør commit før  $T_2$  prøver å skrive  $x$ , blir  $T_2$  abortert idet den prøver å skrive  $x$ .
  - Hvis  $T_1$  slipper låsen fordi den aborterer, får  $T_2$  skrive  $x$ .

# Administrativ informasjon for FUW

- Når en transaksjon  $T$  starter, noteres starttidspunktet  $TS(T)$
- Når en transaksjon  $T$  committer, noteres commit-tidspunktet  $TC(T)$
- Planleggeren må for hvert element  $A$  vedlikeholde mengden  $Commit(A)$  av transaksjoner som (nylig) har skrevet  $A$

# FUW-protokollen I

1. T ønsker å lese A: Lesing innvilges alltid
  - Les den versjonen  $A_t$  der  $t$  er størst mulig, men mindre enn  $TS(T)$ .
2. T ønsker å skrive A: Be om eksklusiv lås på A
  - Hvis det finnes en  $U$  i  $Commit(A)$  der  $TC(U) > TS(T)$ , må  $T$  ruller tilbake (aborteres) fordi  $T$  og  $U$  er samtidige, har overlappende skrivemengder og  $U$  alt har committet.
  - Ellers: Hvis låsen på  $A$  er ledig, får  $T$  låsen og kan endre  $A$  til ny verdi, men bare i sitt lokale arbeidsområde (andre får ikke tilgang på den nye verdien før  $T$  vet at den kan committes).
  - Ellers: La  $T$  vente i  $A$ -køen ( $T$  venter på å få lås på  $A$  – dvs.  $T$  venter for å se om den som har låsen, committer eller blir rullet tilbake).

# FUW-protokollen II

## 3. T ønsker å committe:

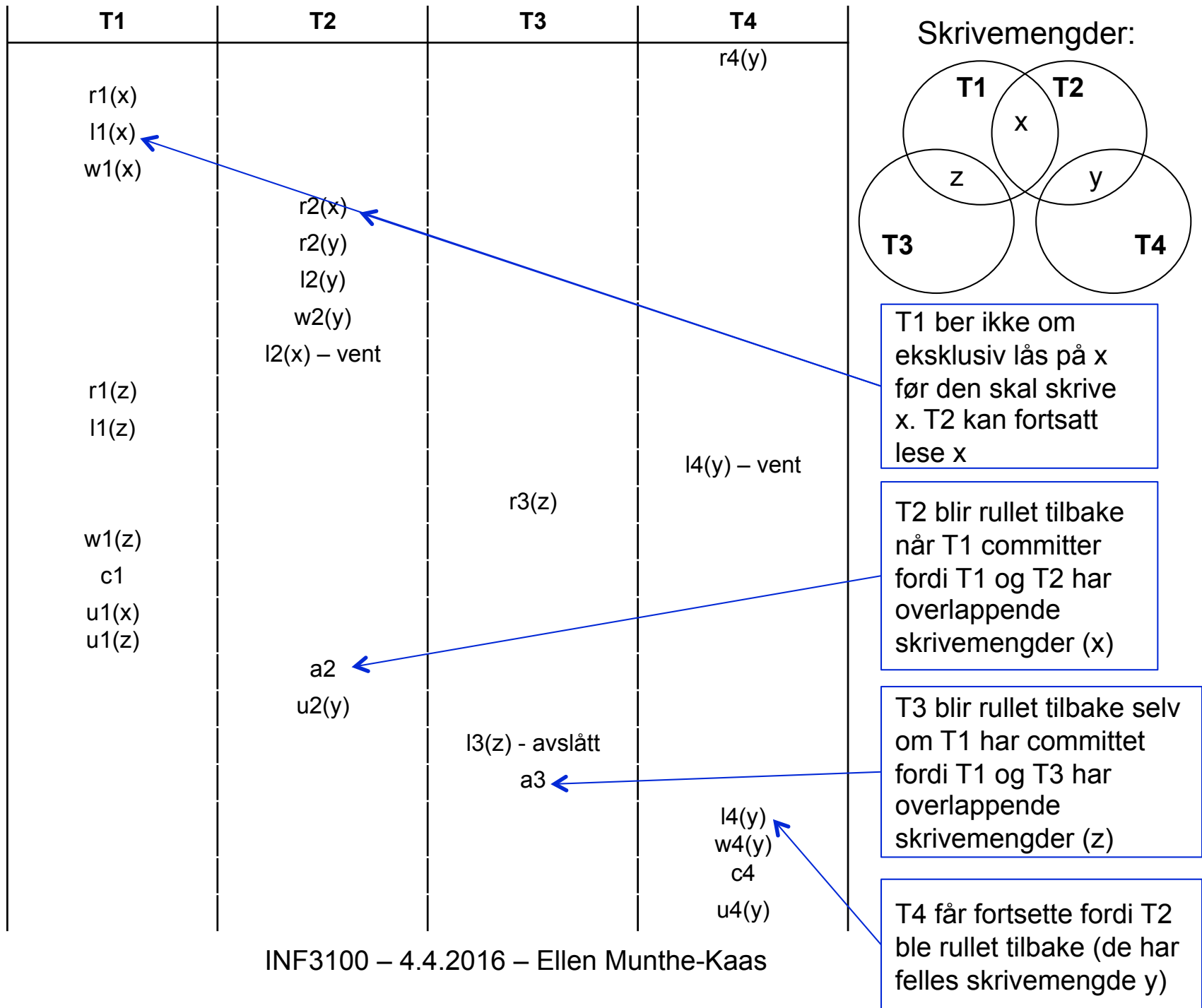
- Utfør  $c_T$  (skriv `commit(T)` i loggen)
- For hvert element  $A$  som  $T$  har lås på, legg  $T$  i `Commit(A)` og skriv  $A$  (dvs. en ny versjon  $A_t$  med  $t = TC(T)$  blir tilgjengelig for andre transaksjoner). Slipp låsen på  $A$ .
- Signaliser til alle som venter på å få lås på en slik  $A$ , at de må rulles tilbake.

## 4. T blir abortert (eller ønsker selv å abortere):

- Skriv `abort(T)` i loggen
- For hvert element  $A$  som  $T$  har lås på, slipp låsen. (En av de transaksjonene som eventuelt venter på låsen, får den og kan fortsette.)



# Eksempel



# Søppeltømming ved bruk av SI

- Regelen for når søppeltømmeren kan fjerne «gamle» versjoner av dataelementer, er slik:
  - En versjon  $A_t$  av et dataelement  $A$  kan bare fjernes hvis det finnes en nyere versjon  $A_u$  som er slik at alle aktive transaksjoner startet etter at  $A_u$  ble skrevet.
    - Hvis  $U$  skrev  $A_u$ , dvs.  $u = TC(U)$ , er det altså slik at for alle aktive transaksjoner  $V$  er  $TS(V) > TC(U)$ .
  - Samtidig som en versjon  $A_t$  blir fjernet, kan den transaksjonen  $T$  som skrev  $A$ , fjernes fra  $Commit(A)$ .
- En konsekvens av denne regelen er at den sist skrevne versjonen av et dataelement aldri kan bli slettet av søppeltømmeren.

# SI vs serialiserbarhet

- Betrakt planen  $P = r_1(x)r_1(y)r_2(x)r_2(y)w_1(y)w_2(x)c_1c_2$
- $P$  er et eksempel på anomalien A5B, skjevskrivning –  $T_1$  skriver  $y$  som  $T_2$  allerede har lest;  $T_2$  skriver  $x$  som  $T_1$  allerede har lest.
- $P$  er opplagt ikke konfliktserialiserbar ( $T_1 \rightarrow T_2 \rightarrow T_1$ ).
- Derimot er  $P$  i SI – både  $T_1$  og  $T_2$  leser bare initielle data (dvs. data som var committet før  $T_1$  og  $T_2$  startet), og skrivemengdene deres er disjunkte.

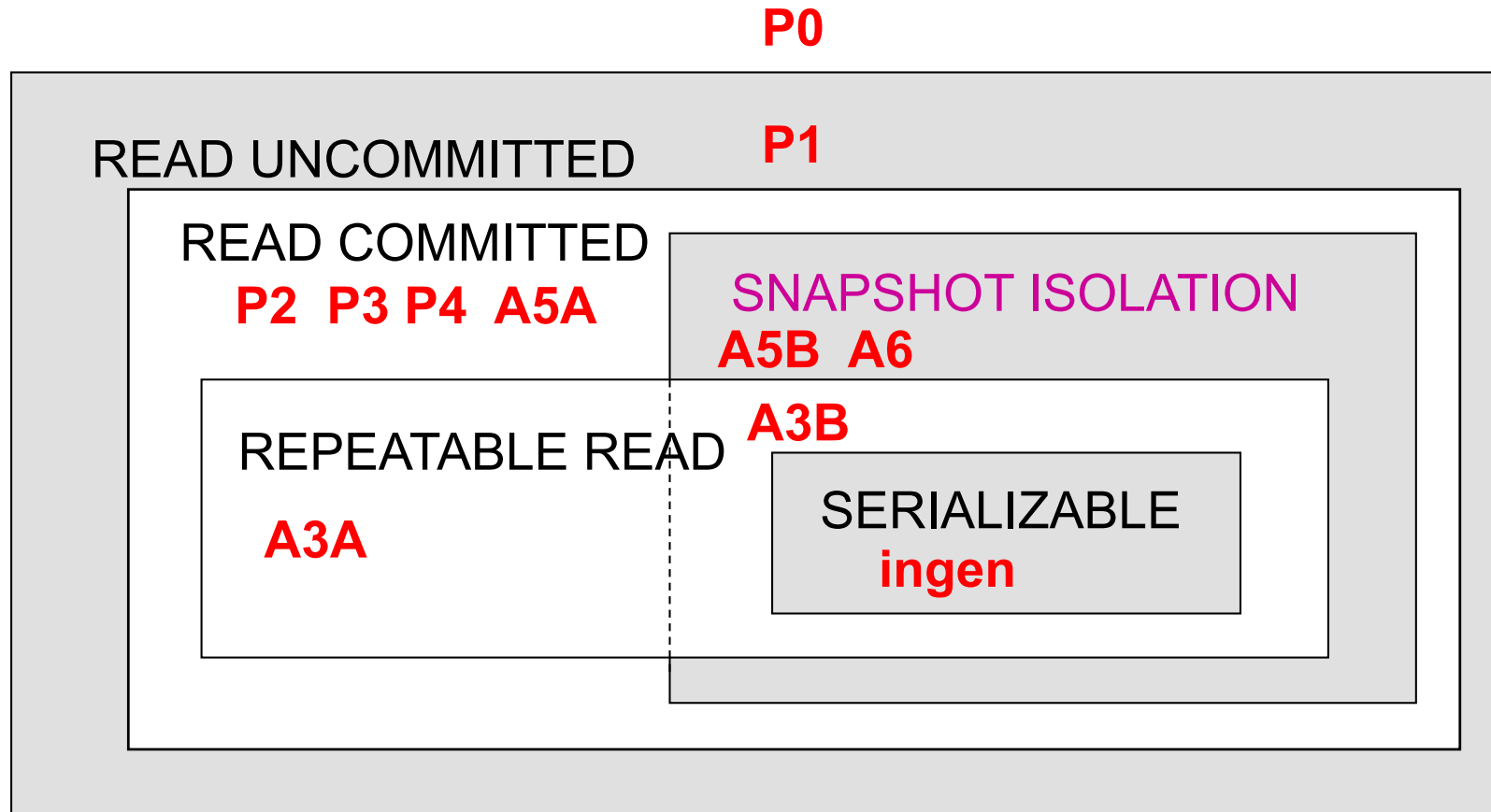
# SI vs fenomener og anomalier

- Disse tre, og ingen andre av de nevnte samtidighets-anomaliene på lysark 4, kan forekomme i planer laget av Snapshot Isolation:
- A3B – fantomskjevskrivning
- A5B – skjevskrivning
- A6 – lesetransaksjonsanomali

# SI vs SQL-isolasjonsnivåer

- På bakgrunn av forrige lysark kan vi fastslå:
- SI er strengere enn Read Committed, men ikke så streng som Serializable.
- SI er hverken strengere eller svakere enn Repeatable Read.

# Isolasjonsnivåer



Isolasjonsnivåer og hvilke fenomener og anomalier som kan forekomme i hvert av dem

# Monotoni i multiversjonsdatabaser

Dette og de to neste lysarkene er hentet fra  
Lene Østbys masteroppgave (2008)

- Det er ikke opplagt hvordan monotoni skal defineres i multiversjonsdatabaser (det egentlige problemet er hvordan projeksjoner skal defineres).
- Betrakt følgende multiversjonsplan for  $T_1$ ,  $T_2$  og  $T_3$ :  
$$S = r_1(x_0)r_1(y_0)r_2(y_0)w_2(y_2)c_2r_3(x_0)r_3(y_2)c_3w_1(x_1)c_1$$
- En rett frem projeksjon av  $S$  på  $T = \{T_1, T_3\}$  blir slik:  
$$\Pi_T(S) = r_1(x_0)r_1(y_0)r_3(x_0)r_3(y_2)c_3w_1(x_1)c_1$$
- Men da lar  $\Pi_T(S)$   $T_3$  lese en versjon av  $y$  skrevet av  $T_2$  som ikke finnes i planen (så  $y_2$  eksisterer kanskje ikke).
- Vi lar derfor  $T_3$  lese siste committede verdi av  $y$ , som gir:  
$$\Pi_T(S) = r_1(x_0)r_1(y_0)r_3(x_0)r_3(y_0)c_3w_1(x_1)c_1$$

# Klassen av SI-planer er monoton

- Bevis (Lene Østby 2008):

La  $S$  være en plan generert i henhold til SI-protokollen.

La  $P$  være projeksjonen av  $S$  på en delmengde av transaksjonene i  $S$  (de ikke-aborterte transaksjonene i  $S$ ).

La  $T$  være en transaksjon i  $P$  som leser et dataelement  $x$ .

Da planleggeren konstruerte  $S$ , planla den at  $T$  skulle lese den siste versjonen  $x_k$  av  $x$  skrevet av en transaksjon  $T_k$  som gjorde commit før  $T$  startet.

Selv om noen transaksjoner i  $S$  er abortert, skal  $T$  fortsatt lese den samme  $x_k$ .

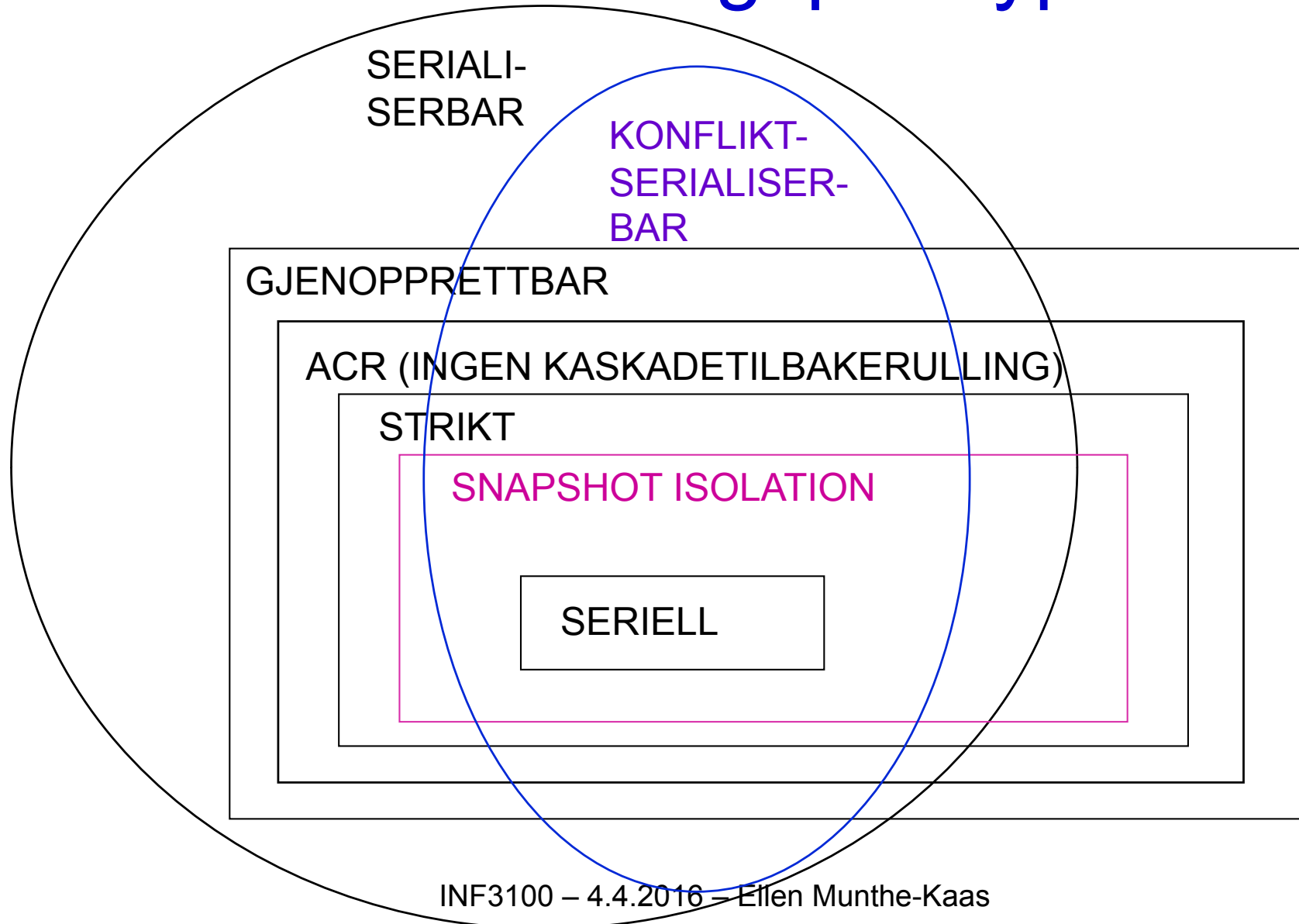
Da gjenstår det bare å observere at en projeksjon ikke kan generere nye skrive-skrive konflikter.



# Alle SI-planer er strikte

- Definisjon: En plan er **strikt** hvis det er slik at hver gang en transaksjon  $T$  skriver et dataelement  $x$ , så må  $T$  gjøre abort eller commit før andre transaksjoner kan lese eller skrive  $x$ .
- Alle SI-planer er strikte. Bevis (Lene Østby 2008):  
La  $S$  være en plan generert i henhold til SI-protokollen.  
La  $T_1$  og  $T_2$  være to transaksjoner i  $S$  hvor  $T_1$  skriver et dataelement  $x$  før  $T_2$  leser det.  
Siden  $T_2$  bare leser verdier som er committet før  $T_2$  startet, må  $T_1$  enten ha abortert eller gjort commit før  $T_2$  leser  $x$ .  
Siden  $S$  ikke inneholder noen skrive-skrive konflikter, følger det at  $S$  er strikt.

# Eksekveringsplantyper



# Snapshot Isolation i industrien

- Merk at det som Oracle og Postgres kaller Serializable, i virkeligheten er Snapshot Isolation.
- Som vi har sett, er ikke det i overensstemmelse med SQL-1992-standardens definisjon av Serializable.
- Microsoft SQL Server overholder standarden.
- Der tilbys både Serializable og Snapshot Isolation som isolasjonsnivåer.
- Av effektivitetsgrunner anbefaler de sterkt å bruke Snapshot Isolation hvis ikke applikasjonen virkelig har behov for nivå Serializable.

# Vranglåser og «timeout»

- I et låsbasert system sier vi at vi har en vranglås når to eller flere transaksjoner venter på hverandre.
- Når en vranglås er oppstått, er det generelt umulig å unngå å rulle tilbake (minst) en transaksjon.
- En «timeout» er en øvre grense for hvor lenge en transaksjon får lov til å være i systemet.
- En transaksjon som overskrider grensen, må frigi alle sine låser og bli rullet tilbake.
- Lengden av «timeout» og velegnethet av denne metoden er avhengig av hva slags transaksjoner vi har.

# Vent-på-grafer

- For å unngå (eventuelt oppdage) vranglåser, kan planleggeren vedlikeholde en Vent-på-graf:
  - Noder: Transaksjoner som har eller venter på en lås
  - Kanter  $T \rightarrow U$ : Det finnes et dataelement A slik at
    - U har låst A.
    - T venter på å få låse A.
    - T får ikke sin ønskede lås på A før U frigir sin.
- Vi har vranglås hvis og bare hvis det er en sykel i Vent-på-grafen.
- En enkel strategi for å unngå vranglås er å rulle tilbake alle transaksjoner som kommer med et låseønske som vil generere en sykel i Vent-på-grafen.

# Vranglåshåndtering ved ordning

- Dersom alle låsbare dataelementer er ordnet, har vi en enkel strategi for å unngå vranglås:
  - la alle transaksjoner sette sine låser i ordningsrekkefølge.
- Bevis for at vi unngår vranglås med denne strategien:
  - Anta at vi har en sykel  $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \dots T_n \rightarrow T_1$  i Vent-på-grafen, at hver  $T_k$  har låst  $A_k$  og at hver  $T_k$  venter på å låse  $A_{k+1}$ , unntatt  $T_n$  som venter på å låse  $A_1$ .
  - Da er  $A_1 < A_2 < \dots < A_n < A_1$ , noe som er umulig.
- Da vi sjelden har en naturlig ordning av dataelementene, er nytteverdien av denne strategien begrenset.

# Vranglåstidsstempler

- Vranglåstidsstempler er et alternativ til å vedlikeholde en Vent-på-graf.
- Alle transaksjoner tildeles et entydig vranglåstidsstempel idet de starter, og dette tidsstempelet har følgende egenskaper:
  - ved tildelingen er det det største som er tildelt til nå
  - det er **ikke** det samme tidsstempelet som (eventuelt) blir brukt til samtidighetskontroll
  - det forandres aldri; transaksjonen beholder sitt vranglåstidsstempel selv om den rulles tilbake
- En transaksjon T sies å være eldre enn en transaksjon U hvis T har et mindre vranglåstidsstempel enn U.

# Vent-dø-strategien

- La T og U være transaksjoner og anta at T må vente på en lås holdt av U.
- **Vent-dø** (**Wait-Die**) er følgende strategi:
  - Hvis T er eldre enn U, får T vente til U har gitt slipp på låsen(e) sin(e).
  - Hvis U er eldre enn T, så dør T, dvs at T rulles tilbake.
- Siden T får beholde sitt vranglåstidsstempel selv om den rulles tilbake, vil den før eller siden bli eldst og dermed være sikret mot flere tilbakerullinger.  
Vi sier at Vent-dø-strategien sikrer mot **utsulting** (**starvation**).



# Skad-vent-strategien

- La T og U være transaksjoner og anta at T må vente på en lås holdt av U.
- **Skad-vent** (*Wound-Wait*) er følgende strategi:
  - Hvis T er eldre enn U, blir U skadet av T.  
Som oftest blir U rullet tilbake og må overgi sin(e) lås(er) til T.  
Unntaket er hvis U allerede er i krympefasen.  
Da overlever U og får fullføre.
  - Hvis U er eldre enn T,  
så venter T til U har gitt slipp på låsen(e) sin(e).
- Om U rulles tilbake, vil den før eller siden bli eldst og dermed være sikret mot flere tilbakerullinger, så også Skad-vent-strategien sikrer mot utsulting.

# Vranglåstidsstempler gjør jobben sin

## TEOREM

Både Vent-dø og Skad-vent forhindrer vranglås.

### Bevis:

Det er nok å vise at begge strategiene sikrer at det ikke kan bli evigvarende sykler i Vent-på-grafen.

Så anta at Vent-på-grafen har en sykel, og la  $T$  være den eldste transaksjonen som inngår i sykelen.

Hvis vi bruker Vent-dø-strategien, kan transaksjoner bare vente på yngre transaksjoner, så ingen i sykelen kan vente på  $T$  som dermed ikke kan være med i sykelen.

Hvis vi bruker Skad-vent, kan transaksjoner bare vente på eldre transaksjoner, så  $T$  kan ikke vente på noen andre i sykelen og kan følgelig ikke selv være med i den.

# Lange transaksjoner I

- En transaksjon kalles *lang* hvis den varer så lenge at den ikke kan få lov til å holde låser i hele sin levetid, f.eks. fordi den involverer en man-in-the-loop
- Vanlig samtidighetskontroll kan ikke brukes for en lang transaksjon

# Lange transaksjoner II

- I utgangspunktet prøver man å skyve så mye som mulig av samtidighetskontrollen ned til DBMSet i form av ordinære database-transaksjoner
  - Hver databasetransaksjon utgjør bare en del av den lange transaksjonen
- I tillegg må konsistens for den lange transaksjonen som helhet håndteres

# Sagaer

- En **saga** representerer alle mulige forløp av en lang transaksjon og består av
  - en mengde (korte) transaksjoner kalt aksjoner
  - en graf hvor nodene er aksjonene samt to terminalnoder **abort** og **ferdig**.
    - En kant  $A_i \rightarrow A_k$  betyr at  $A_k$  bare kan utføres dersom  $A_i$  er utført.
    - Alle noder unntatt **abort** og **ferdig** har utgående kanter.
  - en markert **startnode** (første aksjon som utføres).
- Merk at en saga kan inneholde sykler.

# Samtidighetskontroll for sagaer

- En lang transaksjon L er en sti gjennom sagaen fra start-noden  $A_0$  til en av terminalnodene (fortrinnsvis **ferdig**).
- Aksjonene er, og behandles som, vanlige database-transaksjoner.
- L aborterer ikke selv om en aksjon blir rullet tilbake.
- I en saga har hver aksjon A en kompenserende aksjon  $A^{-1}$  som opphever virkningen av A.

Presist: Hvis D er en lovlig databasetilstand og S er en eksekveringsplan, skal det å utføre S og  $ASA^{-1}$  på D gi samme resultattilstand.

- Hvis L ender i **abort**, fjernes virkningen av L ved å kjøre de kompenserende aksjonene i omvendt rekkefølge:  
 $A_0A_1\dots A_n$  **abort** kompenseres med  $A_n^{-1}\dots A_1^{-1}A_0^{-1}$  **ferdig**.

# Optimistic offline locks

- Brukes for å håndheve samtidighetskontroll ved lange transaksjoner
- Velegnet når det typisk forekommer få konflikter
- En vanlig implementasjon er å benytte et versjonsnummer som lagres sammen med dataelementet
  - Når dataelementet leses, leses også versjonsnummeret
  - Når dataelementet skal skrives, må det leste versjonsnummeret oppgis. Hvis det er identisk med versjonsnummeret i databasen, skrives ny verdi, og versjonsnummeret inkrementeres. Hvis de er forskjellige, er det en konflikt. Hvordan konflikten skal håndteres, kommer an på virksomhetsområdet.