UNIVERSITETET I OSLO

Transaksjonshåndtering Del 3

Ragnar Normann

View-serialiserbarhet

- Hittil har vi sett på eksekveringsplaner som har vært konfliktekvivalente med serielle eksekveringsplaner
- View-serialiserbare eksekveringsplaner baserer seg på et svakere ekvivalensbegrep enn konfliktekvivalens, kalt view-ekvivalens
- View-serialiserbarhet tar utgangspunkt i den avhengigheten som oppstår mellom to transaksjoner T og U når U leser en verdi som T har skrevet
- Forskjellen mellom view- og konfliktserialiserbarhet viser seg når T skriver en A som ingen leser (fordi en annen transaksjon også skriver A før noen har lest A)
- En slik $w_T(A)$ kan gi en sykel i presedensgrafen uten å stride mot view-seriabilitet

View-ekvivalens

- La S_1 og S_2 være to eksekveringsplaner for de samme transaksjonene $\{T_1, \ldots, T_m\}$
- Definer to fiktive transaksjoner T_0 og T_f ved at
  - T_0 skriver alle dataelementene i DB før en plan startes
  - T_f leser alle dataelementene etter at en plan er kjørt
- For alle $r_l(A)$ i en plan (inklusiv $r_f(A)$) definerer vi kilden til $r_l(A)$ som den T_k i planen (inklusiv T_0) som skrev den verdien av A som ble lest med $r_l(A)$
- Vi sier at S_1 og S_2 er ***view-ekvivalente*** hvis alle $r_l(A)$ har samme kilde i S_1 og S_2
- En eksekveringsplan er ***view-serialiserbar*** hvis den er view-ekvivalent med en seriell eksekveringsplan

Eksempel: Eksekveringsplan S_V

T_1:			$r_1(A)$		$w_1(B)$			
T_2:	$r_2(B)$	$w_2(A)$				$w_2(B)$		
T_3:				$r_3(A)$			$w_3(B)$	

- S_V er ikke konfliktserialiserbar fordi $r_2(B) <_{Sv} w_1(B) <_{Sv} w_2(B)$ som gir sykelen $T_2 \rightarrow T_1 \rightarrow T_2$ i presedensgrafen
- La oss se på kildene til alle leseoperasjonene i S_V:
  - Kilden til $r_2(B)$ er T_0
  - Kildene til $r_1(A)$, $r_3(A)$ og $r_f(A)$ er alle T_2
  - Kilden til $r_f(B)$ er T_3
- Vi ser at S_V er view-ekvivalent med den serielle planen $T_2T_1T_3$: $r_2(B)\ w_2(A)\ w_2(B)\ r_1(A)\ w_1(B)\ r_3(A)\ w_3(B)$

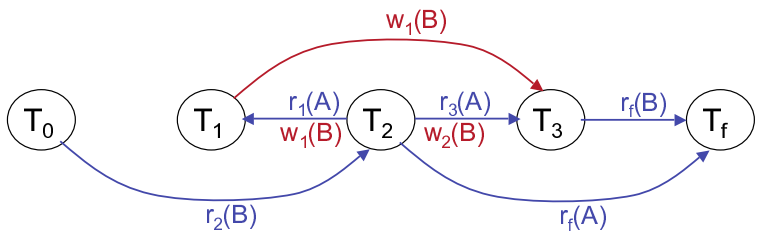
Polygrafer

- Polygrafen for en eksekveringsplan S er en generalisering av presedensgrafen til S:

1. En node for hver transaksjon i S (inklusive T_0 og T_f)
2. For hver $r_l(A)$ med kilde T_k: Legg en kant fra T_k til T_l
3. Hvis T_k er kilde for $r_l(A)$, og en annen T_n også skriver A, må $w_n(A)$ enten komme før $w_k(A)$ eller etter $r_l(A)$
   Dette representeres ved et (stiplet) ***kantpar*** fra T_n til T_k og fra T_l til T_n (bare en av dem er en virkelig kant)
   Det er to unntak:
   a) Hvis $T_k = T_0$, kan ikke T_n komme foran T_k, så kantparet erstattes med en vanlig kant T_l til T_n
   b) Hvis $T_l = T_f$, kan ikke T_n komme etter T_l, så kantparet erstattes med en vanlig kant T_n til T_k

Eksempel: Polygraf for S_V

T_1: $r_1(A)$ $w_1(B)$
T_2: $r_2(B)$ $w_2(A)$ $w_2(B)$
T_3: $r_3(A)$ $w_3(B)$

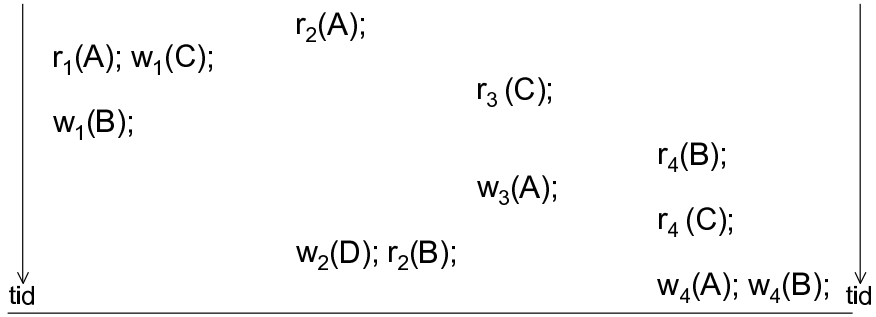


Trinn 1 Trinn 2 Trinn 3

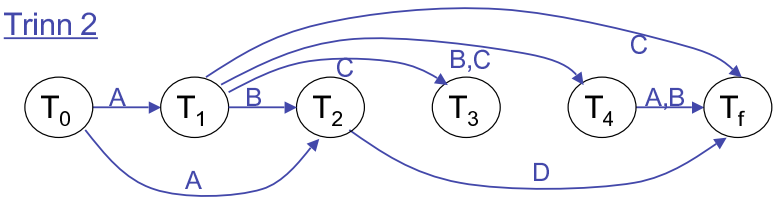
Mer om kantpar i polygrafer

- Hensikten med kantparene er å unngå innblanding i (forstyrrelse av) kantene som kom fra fase 2:
- Kantparet uttrykker at ingen kan skrive et dataitem mellom at kilden skriver det og leseren leser det
- Kandidater til å forstyrre en fase-2-kant $T_k \rightarrow T_l$ er alle transaksjoner som skriver et dataelement som har gitt opphav til kanten $T_k \rightarrow T_l$
- Fire transaksjoner kan aldri forstyrre kanten $T_k \rightarrow T_l$:
  - Endepunktene T_k og T_l
  - T_0 og T_f (som aldri kan komme mellom T_k og T_l)

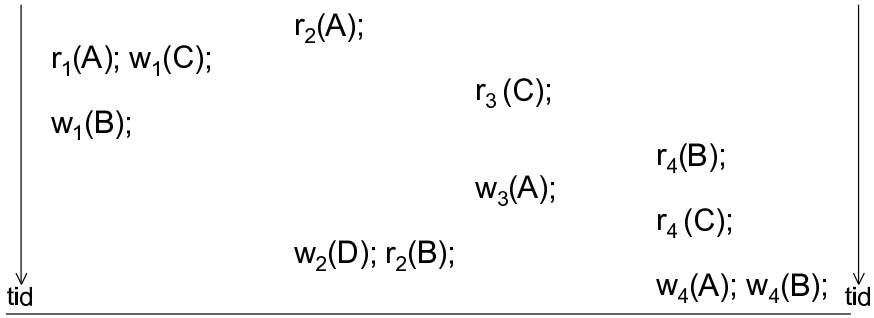
Eksempel: En større polygraf – 1,2



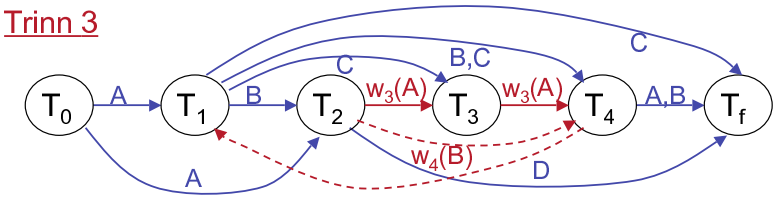
Trinn 2



Eksempel: En større polygraf – 3



Trinn 3



En feil i læreboken

- Det siste eksemplet er eksempel 19.11 i læreboken
- Figur 19.8 på side 1008 i læreboken har en liten feil
- «Vår» polygraf er den riktige
- Det er bare figuren som er gal
- Teksten i eksempel 19.11 er riktig

Test for view-serialiserbarhet

TEOREM

En eksekveringsplan S er view-serialiserbar hvis, og bare hvis, den har en polygraf som kan gjøres til en asyklisk graf G ved å velge eksakt en av kantene i hvert kantpar

Bevis (⇐):

Anta at vi har en slik asyklisk graf G

Enhver topologisk sortering av G gir en sortert liste S_S av transaksjonene i S med følgende egenskaper:

- ingen skriver kommer mellom en leser og dens kilde
- alle skrivere kommer foran sine lesere

Dermed er den serielle planen S_S view-ekvivalent med S, og S er view-serialiserbar

Test for view-serialiserbarhet (forts.)

Bevis (⇒):

Anta at S er view-serialiserbar, dvs at S har en view-ekvivalent seriell plan S_S

For ethvert kantpar ($T_n \rightarrow T_k$, $T_l \rightarrow T_n$) i polygrafen til S må T_n enten komme før T_k eller etter T_l i S_S
(ellers ville skrivingen til T_n bryte forbindelsen fra T_k til T_l i S_S og umuliggjøre at S og S_S er view-ekvivalente)

Videre må hver (ekte) kant i polygrafen til S overholdes av transaksjonsrekkefølgen i S_S

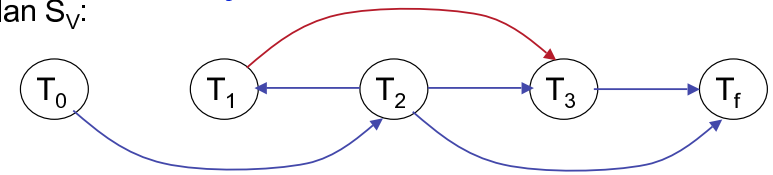
Altså kan vi velge en kant fra hvert kantpar i polygrafen til S slik at alle kantene i den resulterende grafen G er i overensstemmelse med den serielle rekkefølgen i S_S

Da er G sykelfri

QED

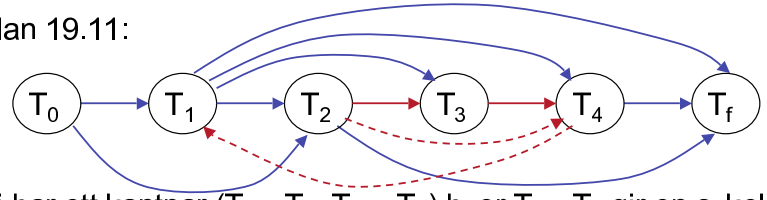
Gjensyn med to eksempler

Plan S_V:



Polygrafen er en graf med serialiseringsordning $T_2 \rightarrow T_1 \rightarrow T_3$

Plan 19.11:



Vi har ett kantpar ($T_4 \rightarrow T_1$,$T_2 \rightarrow T_4$) hvor $T_4 \rightarrow T_1$ gir en sykel

Velger $T_2 \rightarrow T_4$ og får serialiseringsordning $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$

View- vs konfliktserialiserbarhet

- Alle konfliktserialiserbare eksekveringsplaner er view-serialiserbare
- Ved å legge på et krav om ***begrenset skriving***:

  En transaksjon får ikke lov til å skrive et dataelement uten først å ha lest det

  blir alle view-serialiserbare eksekveringsplaner konflikt-serialiserbare
- Det er mye dyrere å håndheve view-serialiserbarhet enn konfliktserialiserbarhet
- Å håndheve view-serialiserbarhet er en NP-komplett oppgave og derfor «umulig» for store transaksjonsmengder (dette er ikke pensum, men informasjon til de interesserte)

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 (evt oppdage) vranglåser, kan planleggeren vedlikeholde en Vent-på-graf:
  - Noder: Transaksjoner som har eller venter på en lås
  - Kanter T→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 $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \ldots T_n \rightarrow T_1$ 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 $A_1 < A_2 < \ldots < A_n < A_1$, 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 vranglåstids-stempel 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***) strategien er som følger:
  - 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 ***utsultning*** (***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***) strategien er som følger:
  - 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 sykler i Vent-på-grafen

Så, ad absurdum, 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

QED

Distribuerte databaser

- En database kalles ***distribuert*** hvis den er spredt over flere datamaskiner, kalt ***noder*** (sites), som er bundet sammen i et nettverk
- Hver node har sitt eget operativsystem og sitt eget DBMS
- Tre viktige formål med distribuerte databaser er:
  - større lagringskapasitet og raskere svartider
  - økt sikkerhet mot tap av data
  - økt tilgjengelighet av data for flere brukere
- Databasen kalles ***distribusjonstransparent*** hvis brukerne (applikasjonene) ikke merker noe til at databasen er distribuert (bortsett fra variasjon i svartidene)
- I dag er det en selvfølge at en distribuert database er distribusjonstransparent

Distribusjon av data

- Et eksempel på distribuerte data kan vi finne i en butikk-kjede hvor alle salg registreres av kassaapparatene og lagres lokalt på en datamaskin (node) i hver enkelt butikk
- Logisk sett har butikkjedens relasjonsdatabase én relasjon som inneholder alle salgsdata fra alle butikkene
- Vi sier at salgsdataene er ***horisontalt fragmentert*** med ett fragment på hver node
- Hvis fragmentene er disjunkte, er fragmenteringen ***total***
- En relasjon er ***vertikalt fragmentert*** hvis ulike attributter er lagret på ulike noder
- Vertikale fragmenter må inneholde primærnøkkelen
- En vertikal fragmentering er ***total*** hvis ingen andre attributter enn primærnøkkelen ligger på flere noder

Replikerte data

- Data som er lagret på flere noder, kalles ***replikerte***
- I en konsistent tilstand er replikerte data like (de er kopier av hverandre)
- Hvis alle data er replikert til alle noder, har vi en ***fullreplikert*** database (kalles også en ***speildatabase***)
- Internt bruker DBMS et replikeringsskjema som forteller hvilke data som ligger på hvilke noder
- Distribusjonstransparens medfører at applikasjonene ikke skal ha kjennskap til replikeringsskjemaet, og at de ikke har ansvar for å oppdatere replikatene
- Replikering er dyrt, men det gir økt hastighet og økt tilgjengelighet til data

Distribuerte transaksjoner og queries

- Når optimalisereren og planleggeren skal lage fysiske eksekveringsplaner, må de ta hensyn til hvilke noder de ulike dataene ligger på (denne informasjonen finnes i replikeringsskjemaet som er kopiert til alle noder)
- Det er to hovedstrategier å velge mellom:
  - kopier (på billigste måte) de data som trengs, til samme node og utfør eksekveringen der
  - splitt eksekveringen opp i subtransaksjoner på de aktuelle nodene og gjør mest mulig eksekvering der dataene er (viktig for projeksjon og spesielt seleksjon)
- En god optimaliserer kombinerer de to strategiene for å minimalisere datatransmisjonen mellom nodene

Distribuert commit

- En transaksjon i en distribuert database kan oppdatere data på flere noder (spesielt må replikerte dataelementer som er endret, oppdateres på alle noder med replikater)
- Den noden som mottar en transaksjon T, kalles *startnoden* (eller *utgangsnoden*) til T
- Planleggeren finner ut hvilke noder T trenger å aksessere og starter en subtransaksjon på hver av disse (inklusiv startnoden) for å gjøre Ts jobb der
- For å oppnå global atomisitet må enten alle subtransaksjonene gjøre commit, eller alle må abortere
- Følgelig kan ikke T gjøre commit før alle subtransaksjonene har gjort det

Tofase commit (2PC)

- 2PC er en protokoll for å sikre atomisitet av distribuerte transaksjoner
- 2PC bygger på følgende forutsetninger
  - Det finnes ingen global logg
  - Hver node logger sine operasjoner (inklusive meldinger den har sendt til andre noder (disse brukes til gjenoppretting etter nettverksfeil))
  - Hver node sikrer atomisitet for sine lokale transaksjoner
- 2PC forutsetter at en av nodene utpekes til koordinator Vanligvis, men ikke alltid, er det startnoden som velges

2PC-protokollen – Fase I

- Koordinatoren for en distribuert transaksjon T bestemmer seg for å gjøre commit
- Koordinatoren skriver <Prepare T> i loggen på sin node
- Koordinatoren sender meldingen ***prepare T*** til alle noder som har subtransaksjoner av T
- Hver mottager fortsetter eksekveringen til den vet om dens subtransaksjon T_n kan gjøre commit
- Hvis ja,
  – skriv nok i loggen til at det kan gjøres redo på T_n
  – skriv <Ready T> i loggen og skriv loggen til disk
  – send meldingen ***ready T*** til koordinatoren
- Hvis nei,
  – skriv <Don't commit T> i loggen
  – send meldingen ***don't commit T*** til koordinatoren

2PC-protokollen – Fase II

- Hvis koordinatoren har mottatt ***ready T*** fra alle nodene (subtransaksjonene)
  - skriver koordinatoren <Commit T> i sin logg og
  - sender ***commit T*** til alle andre involverte noder
- Hvis koordinatoren har mottatt ***don't commit T*** fra minst en node, eller ikke alle har svart ved «timeout»
  - skriver koordinatoren <Abort T> i sin logg og
  - sender ***abort T*** til alle andre involverte noder
- En node som mottar ***commit T***, gjør commit på sin subtransaksjon og skriver <Commit T> i loggen sin
- En node som mottar ***abort T***, aborterer sin subtransaksjon og skriver < Abort T> i loggen sin

Feilhåndtering ved 2PC

- Hvis en ordinær node går ned, er det opplagt hva den skal gjøre med mindre dens siste loggpost er <Ready T>
  I så fall må den spørre en annen node om den skal gjøre commit T eller abort T
- Hvis koordinatoren går ned, velges en ny koordinator
- Med ett unntak kan den nye koordinatoren fullføre 2PC
- Unntaket er hvis alle nodene har <Ready T> som siste loggpost
  Da kan man ikke avgjøre om den opprinnelige koordinatoren har gjort commit eller abort
  Det er to mulige fortsettelser
  - Vente til koordinatoren kommer opp igjen
  - DBA griper inn og fatter en manuell avgjørelse

Distribuert låsing

- Låsing av et replikert dataelement krever varsomhet:
  Anta T har leselås på en kopi A_1 av et dataelement A
  Anta U har skrivelås på en annen kopi A_2 av A
  Da kan U oppdatere A_2, men ikke A_1
  Resultatet blir en inkonsistent database
- Med replikerte data må vi skille mellom to typer låsing:
  - låsing av et logisk dataelement A (global lås)
  - fysisk låsing av en av kopiene av A (lokal lås)
- Reglene for logiske lese- og skrivelåser er de samme som de som gjelder for vanlige låser i en ikke-distribuert database
- Logiske låser er fiktive – de må avledes av de fysiske

Sentralisert låsing

- Den enkleste måten å implementere logiske låser på er å utnevne en av nodene til ***låsesjef***
- Låsesjefen håndterer alle ønsker om logiske låser og bruker sin egen låstabell som logisk låstabell
- Det er to viktige svakheter ved et slikt sentralisert låsesystem:
  - låsesjefen blir fort en flaskehals ved stor trafikk
  - systemet er svært sårbart; hvis låsesjefen går ned, får ingen satt eller hevet noen lås
- Kostnaden er minst tre meldinger for hver lås som settes:
  - en melding til låsesjefen for å be om en lås
  - en svarmelding som innvilger låsen
  - en melding til låsesjefen for å frigi låsen

Primærkopilåsing

- Primærkopilåsing er en annen type sentralisert låssystem
- I stedet for en felles låsesjef velger vi for hvert logisk dataelement ut en av kopiene som ***primærkopi***
- Den fysiske låsen på primærkopien brukes som logisk lås på et dataelement
- Metoden reduserer faren for flaskehalser ved låsing
- Ved å velge kopier som ofte blir brukt, til primærkopier, reduseres antall meldinger ved håndtering av låser

Distribuert vranglås

- Faren for vranglås i et distribuert låsesystem er stor
- Det finnes mange varianter av Vent-på-grafer som kan forhindre distribuert vranglås
- Erfaring sier at det enkleste og beste i de fleste tilfeller er å bruke «timeout»:
  Transaksjoner som bruker for lang tid, rulles tilbake

Avledede logiske låser

- Metoden går ut på at en transaksjon får en logisk lås ved å låse et tilstrekkelig antall av replikatene
- Mer presist:
  Anta at databasen har n kopier av et dataelement A
  Velg to tall s og x slik at 2x > n og s+x > n
  - en transaksjon får logisk leselås på A ved å ta leselås på minst s kopier av A
  - en transaksjon får logisk skrivelås på A ved å ta skrivelås på minst x kopier av A
- At 2x > n medfører at to transaksjoner ikke begge kan ha logisk skrivelås på A
- At s+x > n betyr at to transaksjoner ikke samtidig kan ha henholdsvis logisk leselås og logisk skrivelås på A

Leselås-En – Skrivelås-Alle

- Dette oppnår vi ved å velge s = 1 og x = n
- Logisk skrivelås krever minst 3(n-1) meldinger og blir svært dyr
- Logisk leselås krever høyst 3 meldinger,
  og hvis det finnes en kopi på transaksjonens startnode, krever den ingen
- Metoden egner seg der skrivetransaksjoner er sjeldne
- Eksempel:
  Elektronisk bibliotek der nodene har kopi av ofte leste dokumenter

Majoritetslåser

- Dette oppnår vi ved å velge s = x = ⌈(n+1)/2⌉
- Logisk skrivelås kan ikke bli billigere enn dette
- Men det at logisk leselås også krever omtrent 3n/2 meldinger, virker svært dyrt
- I systemer med effektiv kringkasting av meldinger blir kostnaden lavere, men virker fortsatt høy
- Fordelen er at metoden er robust mot nettverksfeil
- Eksempel:
  Dersom en nettverksfeil deler databasen i to, kan den delen som inneholder flertallet av nodene fortsette som om intet var hendt
  I minoritetsdelen kan ingen få så mye som en leselås

Lange transaksjoner og sagaer

- En transaksjon kalles ***lang*** hvis den varer så lenge at den ikke kan få lov til å holde låser i hele sin levetid
- Vanlig samtidighetskontroll kan ikke brukes for lange transaksjoner – de håndteres med ***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 og to terminalnoder ***abort*** og ***ferdig***, og hvor en kant $A_i \rightarrow A_k$ betyr at A_k bare kan utføres dersom A_i er utført, og hvor 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 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: $A_0A_1\ldots A_n$***abort*** kompenseres med $A_n^{-1}\ldots A_1^{-1}A_0^{-1}$***ferdig***

INF3100 – Databasesystemer

Slutt på pensum

- Husk spørsmål- og svar-time 30. mai kl. 10.15
- Spørsmål og repriseønsker sendes til

ellenmk@ifi.uio.no og/eller ragnarn@ifi.uio.no