UNIVERSITETET I OSLO
Institutt for informatikk

Replikeringsgrafer og multiversjons-serialiserbarhet

Jon Grov

Hovedfagsoppgave

29. april 2003

Til Elin

Forord

Denne rapporten ville aldri blitt fullført uten tålmodig og svært kompetent hjelp fra mine veiledere, amanuensis Ragnar Normann ved Institutt for Informatikk og Dag Asheim, Linpro.

Hans Christian Kjølberg fortjener stor takk for å ha lest korrektur.

Tusen takk til familie, venner og alle i Linpro som har vist interesse og engasjement, det har vært til uvurderlig inspirasjon.

Til slutt vil jeg takke Ada, som nesten alltid er blid.

Oslo, 29. april 2003

Jon Grov

Innhold

1 Innledning **7**

1.1 Transaksjoner . . . 7

1.2 Replikering . . . 9

1.3 Bakgrunn og mål for arbeidet . . . 9

1.4 Om rapporten . . . 10

2 Transaksjoner og serialiserbarhet **11**

2.1 ACID . . . 11

2.2 Systemmodell . . . 12

2.2.1 Operasjoner . . . 12

2.2.2 Transaksjoner . . . 13

2.2.3 Lese- og skrivemengder . . . 13

2.3 Samtidighetskontroll . . . 14

2.3.1 Eksekveringsplaner . . . 14

2.3.2 Formålet med samtidighetskontroll . . . 15

2.4 Serialiserbarhet . . . 15

2.4.1 Konfliktekvivalens og konfliktserialiserbarhet . . . 16

2.4.2 Viewekvivalens og viewserialiserbarhet . . . 17

2.5 Algoritmer for samtidighetskontroll . . . 19

2.5.1 Tofase-låsing . . . 20

2.5.2 Vranglåser . . . 22

2.5.3 Samtidighetskontroll med tidsstempelordning . . . 24

2.5.4 Samtidighetskontroll basert på konfliktgraf-testing . . . 25

2.6 Multiversjons-samtidighetskontroll . . . 25

2.6.1 Multiversjonsplaner og multiversjonsserialiserbarhet . . . 26

2.6.2 Konflikter i multiversjonsplaner 27
2.6.3 Multiversjons-viewserialiserbarhet 28
2.6.4 Multiversjons-serialiseringsgrafer 29
2.7 Algoritmer som tilbyr multiversjonsserialiserbarhet 31
2.7.1 Multiversjons-tidsstempelordning 31
2.7.2 Tidsstempelordning med snapshot 32
2.8 Optimistisk samtidighetskontroll 35
2.8.1 Bevisskisse for serialiserbarhet 36

3 Samtidighetskontroll i distribuerte databaser 37
3.1 Distribuerte databasesystemer 37
3.2 Globale eksekveringsplaner 38
3.3 Global serialiserbarhet 39
3.4 Algoritmer som sikrer global serialiserbarhet 39
3.4.1 Global tofase-låsing 41
3.4.2 Global tidsstempelordning 42
3.4.3 Tidsstempelordning med globale snapshot 43
3.4.4 Tofase-commit 44
3.5 Replikerte databaser 45
3.5.1 Replikeringsalgoritmer 45
3.5.2 Skaleringsevne 46
3.5.3 Vranglåser 48
3.5.4 Andre replikeringsstrategier 50

4 Replikeringsgrafer 51
4.1 Definisjon av replikeringsgrafer 51
4.2 Replikeringsgrafer og global serialiserbarhet 52
4.3 Algoritmer basert på en replikeringsgraf 55
4.4 Replikeringsalgoritme i [BK97] 55
4.4.1 Utførelse av transaksjoner 57
4.4.2 Bevisskisse for global serialiserbarhet 58
4.4.3 Vranglåser 59
4.4.4 Algoritmens ytelse 61
4.5 Replikeringsalgoritme i [ABKW98] 62

4.5.1 Bevisskisse for global serialiserbarhet 64
4.5.2 Ytelsesmålinger i [ABKW98] 65
4.5.3 Optimalisering av lesetransaksjoner 66
4.6 Replikeringsgrafer og globale snapshot 67
4.6.1 Globale snapshot 67
4.6.2 Replikeringsalgoritme 68
4.6.3 Snapshotalgoritmens korrekthet 70

5 Avslutning **73**
5.1 Oppsummering 73
5.2 Videre arbeid 74

Kapittel 1

Innledning

Å kunne fordele belastningen over flere fysiske maskiner er en forutsetning for mange databasebaserte tjenester.

Replikerte databaser, dvs. databaser hvor identiske data lagres ved ulike maskiner, gir i teorien mer effektiv lastbalansering og høyere toleranse for feil på enkeltkomponenter. Vi skal se på transaksjonsutførelse i slike databaser.

1.1 Transaksjoner

En transaksjon representerer en samling operasjoner som databasesystemet gir brukeren garantier for utførelsen av. Følgende eksempel illustrerer behovet for transaksjonsstøtte:

Eksempel 1.1 [1]

Anta at vi for en handelsløsning har en database som lagrer ordrer og varebeholdning. En ordre skal bare aksepteres dersom beholdningen av den aktuelle varen er større enn 0, og følgende pseudokodesekvens registrerer en ny ordre:

```
b := les(beholdning)   # Sett b lik verdien av 'beholdning'
if (b == 0) {
  error("Tomt - lang leveringstid!")
} else {
  b := b - 1
```

[1] Start på eksempler markeres alltid med overskrift, og en vannrett linje viser eksempelets avslutning (se neste side).

```
    o := NyOrdre()       # Oppretter en ny ordre
    skriv(beholdning, b) # Skriv ny beholdning til databasen
    skriv(ordre, o)      # Legg inn den nye ordren i ordredatabasen
}
```

Hvis vi lar a og b representere to kunder som samtidig bestiller en gitt vare, kan innleggingen av disse ordrene forløpe slik:

```
# Anta beholdning == 1 før utførelsen starter
b_a := les(beholdning)
b_b := les(beholdning)

# Både b_a og b_b er lik 1, og utførelsen fortsetter:
b_a := b_a - 1
b_b := b_b - 1

skriv(beholdning, b_a)
skriv(beholdning, b_b)

# beholdning == 0
o_a := NyOrdre()
o_b := NyOrdre()

skriv(ordre, o_a)
skriv(ordre, o_b)
```

Vi ser at databaseintegriteten er ødelagt: En ordre skal ikke kunne registreres dersom beholdningen er tom, og beholdningsinformasjonen blir feilaktig satt lik 0.

Et databasesystem som håndterer transaksjoner er ansvarlig for *samtidighetskontroll*, dvs. at tilgangen til dataobjektene kontrolleres slik at samtidig utførelse av flere transaksjoner aldri gir inkonsistens. Uformelt kaller vi utførelsen av transaksjoner *korrekt* dersom vi er sikre på at de aldri etterlater inkonsistente data. I eksempelet over kan vi garantere korrekt eksekvering dersom vi lar operasjonene for innlegging av en ordre utgjøre en transaksjon.

Dersom ett eller flere av objektene som aksesseres i en transaksjon lagres ved forskjellige maskiner, har vi behov for *distribuerte transaksjoner*. I eksempel 1.1 er databasen distribuert dersom ordre- og beholdningsinformasjon lagres ved separate maskiner.

En samling databasesystemer som tilbyr distribuerte transaksjoner kalles et distribuert databasesystem, og databaser som lagres i et slikt system kalles distribuerte databaser. Vi kaller en komponent i et distribuert databasesystem en *node*.

1.2 Replikering

Dersom man i en distribuert database lagrer ett eller flere av objektene ved mer enn én node, er databasen replikert. Ofte tilbyr databasesystemet automatisk oppdatering av kopier, dvs. at dersom en transaksjon oppdaterer et objekt, sikrer databasesystemet at alle kopiene også oppdateres. Databasen i eksempel 1.1 vil være replikert dersom beholdningen eller ordrene, eller begge deler, lagres ved begge nodene. Dette gir større pålitelighet, siden databasen blir mindre sårbar for feil på enkeltnoder. Det kan også gi bedre ytelse: Siden en leseoperasjon bare trenger å lese objektet ved én node, kan lesebelastningen for samme objekt fordeles over flere noder.

Ytelsen blir imidlertid bare bedre dersom antall leseoperasjoner er betydelig større enn antall skriveoperasjoner, og den største utfordringen med replikerte databaser er skaleringsevnen: Oppdatering av et replikert objekt krever én oppdateringsoperasjon ved hver node som lagrer objektet. Dersom antall noder er lavt, eller oppdateringer skjer relativt sjelden, er dette problemet overkommelig. Vi skal imidlertid se at store replikerte databaser stiller betydelige krav til effektiv samtidighetskontroll.

1.3 Bakgrunn og mål for arbeidet

Formålet med dette arbeidet har vært å forstå, og om mulig forbedre, kjente strategier for samtidighetskontroll i replikerte databaser.

Opprinnelig var målet å implementere støtte for replikering i databasesystemet PostgreSQL[2]. Denne implementasjonen skulle være generelt anvendbar, dvs. at den skulle oppfylle databasesystemets krav til korrekt eksekvering og akseptabel ytelse, og den skulle tilby transparent replikering, dvs. at brukeren ikke trenger å kjenne til at databasen er replikert.

Etter å ha lest noen artikler om emnet, særlig [GHOS96], ble det klart at slik replikering er relativt lite utbredt, og grunnen er enkel: det finnes ingen velutprøvde algoritmer for samtidighetskontroll som kombinerer korrekt eksekvering og god skaleringsevne med replikerte data.

[BK97] foreslår en algoritme basert på *replikeringsgrafer*, og simuleringsresultatene i [ABKW98] indikerer at denne gir langt bedre ytelse enn noen av algoritmene [GHOS96] omtaler. Implementasjonen ble skrinlagt, og hovedtemaet i denne rapporten er, sammen med en presentasjon av denne algoritmen, et forslag til endringer som trolig kan gjøre den enda raskere og mer anvendelig.

[2] http://www.postgresql.org/

1.4 Om rapporten

I kapittel 2 presenteres transaksjoner og samtidighetskontroll i ikke-distribuerte databaser, og i kapittel 3 defineres dette også for distribuerte og replikerte databaser. Kapittel 4 omhandler utelukkende replikeringsgrafer, mens kapittel 5 gir en kort oppsummering og et forslag til tema for videre undersøkelser.

Alle fagtermer er fornorsket dersom jeg har funnet en passende oversettelse. For enkelte begreper, de mest sentrale er *commit* og *snapshot*, har jeg ikke greid å finne noen norsk formulering som bevarer originalens eleganse og anvendelighet, og jeg har derfor valgt å bruke dem uoversatt.

Kapittel 2

Transaksjoner og serialiserbarhet

Dette kapittelet gir en grundigere presentasjon av transaksjoner og samtidighetskontroll.

De viktigste kildene er [WV01] og [BHG87], som begge er sentrale verk for emnet transaksjonshåndtering. Det meste av innholdet i [BHG87] er også omtalt i [WV01], men [BHG87] er av og til brukt for å få en annen vinkling på vanskelig stoff.

Der ikke annet er angitt, er presentasjonen basert på [WV01]. Hvis andre kilder er brukt, er disse eksplisitt referert.

2.1 ACID

Transaksjoner gir, som nevnt i innledningen, garantier for korrekt utførelse av en mengde operasjoner. Blant disse garantiene inngår sikring mot at samtidig utførende transaksjoner gir inkonsistens, og et krav om «alt eller ingenting», dvs. at dersom ikke alle operasjonene i en transaksjon kan fullføres, skal hele transaksjonen kanselleres.

Disse kravene oppsummeres gjerne i akronymet ACID, *Atomicity* (atomisitet), *Concistency* (konsistens), *Isolation* (isolasjon) og *Durability* (varighet):

- *Atomisitet*

  Dersom én eller flere av operasjonene i en transaksjon ikke kan fullføres, skal hele transaksjonen kanselleres. I praksis krever dette støtte for *rollback*, dvs. at tidligere utførte operasjoner må kunne kanselleres så lenge transaksjonen ikke er committet.

- *Konsistens*

  I en konsistent database oppfyller alle dataene databasens integritetsregler. En transaksjon som skriver objekter representerer en tilstandsendring i databasen, og det er et krav at utførelsen av en transaksjon ikke skal gjøre en konsistent database inkonsistent.

- *Isolasjon*

  Samtidig utførelse av flere transaksjoner vil kunne gi inkonsistente data. For å hindre dette, må transaksjonene isoleres fra hverandre, det vil si at en transaksjon ikke skal kunne lese data som skrives av andre, samtidig kjørende transaksjoner.

- *Varighet*

  Når en transaksjon er gjennomført, skal resultatet forbli i databasen. Dette medfører at databasen må være i stand til å gjenopprette seg selv etter et systemkræsj.

2.2 Systemmodell

Vi begrenser oss foreløpig til databaser som ikke er distribuert. En transaksjon består av en begrenset mengde lese- og skriveoperasjoner, og hver transaksjon tilordnes en unik id. Vi lar t_i betegne en transaksjon i. En transaksjon er til enhver tid i én og bare én av følgende tilstander: *AKTIV, AVBRUTT* eller *COMMITTET*, og den blir aktiv idet den registreres av databasesystemet. Transaksjoner kan avbrytes eksplisitt av brukeren, eller ved at databasesystemet ikke ser seg i stand til å fullføre i henhold til ACID-kravene (dette inkluderer systemkræsj). En transaksjon committes etter at den har fullført alle operasjonene og alle oppdateringer er lagret, slik at varighetskravet oppfylles.

2.2.1 Operasjoner

Vi lar i denne sammenhengen transaksjonene bestå av to typer operasjoner:

- lesing av objekter
- skriving av objekter

Det er for oss uvesentlig *hva* transaksjonene bruker dataene til, men vi må anta at de går gjennom en eller annen form for prosessering underveis, og alle verdier en transaksjon skriver kan derfor avhenge av objekter transaksjonen tidligere har lest.

Vi antar at en transaksjon aldri leser eller skriver et objekt mer enn én gang (men det er ingenting i veien for at en transaksjon både leser og skriver samme objekt), og alle lese- og skriveoperasjoner er derfor unikt identifiserbare ved trippelet *(operasjonstype, transaksjon, objekt)*. En transaksjon kan bare utføre lese- eller skriveoperasjoner dersom den er AKTIV.

Vi antar at alle databaseobjektene er av samme type, men hva slags objekt det faktisk er snakk om (tuppel, relasjon etc.) er i denne sammenhengen irrelevant, og vi identifiserer enkeltobjekter med bokstavene x, y og z.

Vi lar *read* og *write* betegne henholdsvis lese- og skriveoperasjoner. Det er gode grunner til å bruke en relativt kompakt notasjon for operasjonene, og en sekvens av operasjoner *(read, i, x), (write, j, y)* presenteres vanligvis slik: $r_i(x)\ w_j(y)$.

Vi har også ofte bruk for å definere transaksjonens avslutningstilstand. Vi innfører derfor to avslutningsoperasjoner, *commit* og *abort*, og c_i og a_i betegner henholdsvis *commit* og *abort* for en transaksjon t_i.

2.2.2 Transaksjoner

Formelt definerer vi en transaksjon slik:

En transaksjon t_i består av en mengde operasjoner O_i og en *partiell* ordning $<_i$ over O_i. Det er et krav at for alle par av operasjoner q, r i O_i som aksesserer samme objekt, og hvor minst en av dem er en skriveoperasjon, inngår enten (q, r) eller (r, q) i $<_i$.

At vi bare krever en partiell ordning av operasjonene, skyldes at det ikke alltid er nødvendig å kjenne rekkefølgen operasjonene utføres i, og det vil, for eksempel i en distribuert database, være ønskelig å parallelleksekvere. Vi forutsetter imidlertid at en transaksjon aldri leser objekter den allerede har skrevet, og vi må derfor, for en transaksjon som både leser og skriver et objekt x, alltid sikre at leseoperasjonen utføres før skriveoperasjonen.

Operasjonene i en transaksjon listes gjerne sekvensielt, men vi er sjelden interessert i om dette faktisk representerer en total ordning.

Eksempel 2.1

La x, y være to databaseobjekter. En transaksjon t_i, som leser og skriver begge objektene, presenteres slik: $t_i = r_i(x)\ r_i(y)\ w_i(x)\ w_i(y)$.

2.2.3 Lese- og skrivemengder

For en transaksjon t_i er *lesemengden* for t_i definert som mengden av alle objekter t_i leser. Tilsvarende inneholder *skrivemengden* for t_i alle objekter som skrives av t_i.

Vi vil ofte ha bruk for å skille mellom *skrivetransaksjoner*, som betegner transaksjoner som skriver ett eller flere objekter, og *lesetransaksjoner*, hvor skrivemengden er tom.

2.3 Samtidighetskontroll

To av de viktigste komponentene i et system for transaksjonshåndtering er mekanismer for *samtidighetskontroll* og *gjenopprettelse*. Bruk av felles ressurser må kontrolleres slik at utførelsen av flere samtidige transaksjoner ikke gir inkonsistens, og en konsistent database må til enhver tid kunne gjenopprettes, uansett når og hvordan utførelsen av en eller flere transaksjoner avbrytes. Hvis vi tillater oss en svært upresis klassifisering, kan vi si at samtidighetskontroll handler om å innfri isolasjonskravet, mens behovet for gjenopprettelse skyldes atomisitets- og varighetskravene.

Denne rapporten omhandler samtidighetskontroll i replikerte databaser, og problemer knyttet til gjenopprettelse blir i liten grad omtalt. Dette kan forhåpentligvis forsvares med at gjenopprettelse kan behandles uavhengig av samtidighetskontroll. Et viktig unntak er *dirty read*, dvs. at en transaksjon t_i leser et objekt x, hvor x er skrevet av en transaksjon t_j som ennå ikke er committet. Hvis vi committer t_i, og t_j senere må avbrytes, blir databasen inkonsistent. En løsning på dette skisseres kort i begynnelsen av seksjon 2.5.

2.3.1 Eksekveringsplaner

En *eksekveringsplan*, vanligvis forkortet til *plan*, beskriver utførelsen av én eller flere transaksjoner. Eksekveringsplaner brukes for å kunne presentere problemer knyttet til samtidig utførelse av transaksjoner. En plan beskriver en sekvens av operasjoner, og rekkefølgen av operasjoner tilhørende en bestemt transaksjon må være identisk med rekkefølgen operasjonene har i transaksjonen. Formelt kan vi definere en eksekveringsplan slik:

La O_T være unionen av operasjonene i en mengde transaksjoner $T = \{t_1, t_2, ..., t_n\}$, og la A_T være unionen av avslutningsoperasjonene for transaksjonene i T. For hver transaksjon t_i i T inngår enten c_i eller a_i i A_T, men ikke begge.

En plan p for T vil da bestå av en mengde OA_T, hvor $OA_T \subseteq (O_T \cup A_T)$, samt en partiell ordning $<_p$ over OA_T. På samme måte som for transaksjoner er det et krav at for alle par av operasjoner o_k, o_l i O_T som aksesserer samme objekt, og hvor minst en av dem er en skriveoperasjon, inngår enten (o_k, o_l) eller (o_l, o_k) i $<_p$.[1] Ofte bruker vi notasjonen $o_k <_p o_l$ for å vise at o_k utføres før o_l i p.

[1] Indeksen på en operasjon angir hvilken transaksjon den tilhører, slik at en operasjon o_k forutsettes å inngå i en transaksjon t_k.

I tillegg har vi at for alle par av operasjoner q, r, i en transaksjon t_i i T, må (q, r) inngå i $<_p$ dersom (q, r) inngår i $<_i$. Med andre ord: Hvis en operasjon q skal utføres før en operasjon r i transaksjon t_i, må q også utføres før r i p. En transaksjon kan ikke utføre andre operasjoner etter avslutningsoperasjonen, dvs. at dersom c_i eller a_i inngår i en plan p, må vi for alle operasjoner o_i ha at $o_i <_p \{c_i, a_i\}$.[2]

En plan p er komplett dersom den, for enhver transaksjon t_i som utfører operasjoner i p, oppfyller følgende:

- c_i eller a_i inngår i p.
- Det finnes ingen operasjoner i t_i som ikke utføres i p.

Et synonym for en komplett plan er *historie*. Vi bruker ofte den generelle termen plan, og angir bare om den er komplett hvis det er av betydning.

2.3.2 Formålet med samtidighetskontroll

I kapittel 1 viste vi et eksempel på inkonsistent transaksjonsutførelse. Behovet for å kontrollere samtidig utførende transaksjoner kan også illustreres med følgende (kanoniske) eksempel:

Eksempel 2.2

La x representere saldoen for en bankkonto, og la t_i og t_j være to transaksjoner som skriver x. Transaksjon t_i legger til 300, mens t_j trekker fra 300. Før t_i og t_j begynner er x lik 300, og et rimelig konsistenskrav er da at x fremdeles er 300 etter at t_i og t_j er ferdig utført. En mulig plan for utførelsen av t_i og t_j er imidlertid: $p = r_i(x)\ r_j(x)\ w_i(x)\ c_i\ w_j(x)\ c_j$. Etter at p er ferdig, vil x være lik 0.

Vi kaller den delen av et databasesystem som utfører samtidighetskontroll en *planlegger* (engelsk: *scheduler*). Planleggerens oppgave er å sikre korrekt eksekvering, og vi har derfor bruk for klare kriterier for hvorvidt en eksekveringsplan er korrekt.

2.4 Serialiserbarhet

En plan er *seriell* dersom vi for alle par av transaksjoner t_i, t_j i p enten har at samtlige operasjoner i t_i utføres før t_j utfører noen operasjoner, eller motsatt, at alle operasjonene i t_j utføres før noen av operasjonene i t_i.

[2] Konstruksjonen $o_i <_p \{c_i, a_i\}$ skal tolkes slik: «i p ordnes o_i før t_i's avslutningsoperasjon, som enten er c_i eller a_i»

Hvis vi forutsetter at utførelsen av én enkelt transaksjon alltid er korrekt, vil utførelsen av enhver seriell plan også være korrekt. Det er lett å verifisere at dersom transaksjonene i eksempel 2.2 ble utført serielt, ville isolasjonskravet vært oppfylt.

Siden man sjelden ønsker å utføre alle transaksjoner serielt, trenger vi kriterier for å sikre korrekt utførelse av ikke-serielle planer. Et slikt kriterium er *serialiserbarhet*:

La $\approx$ betegne en ekvivalensrelasjon som sikrer at hvis en plan p er ekvivalent med en seriell plan i henhold til $\approx$, er p korrekt med hensyn på isolasjonskravet. Dersom det finnes en slik ekvivalensrelasjon, er p serialiserbar.

Vi skal ta for oss to slike ekvivalenskriterier, *viewekvivalens* og *konfliktekvivalens*.

2.4.1 Konfliktekvivalens og konfliktserialiserbarhet

Konfliktekvivalens er basert på operasjoner i *konflikt*:

La t_i og t_j være to ulike transaksjoner i en mengde transaksjoner T. Da er to operasjoner o_i og o_j, $o_i \in t_i$ og $o_j \in t_j$, i *konflikt* dersom de leser eller skriver et felles objekt x, og minst én av dem er en skriveoperasjon.

Konfliktekvivalens

To planer p og q er konfliktekvivalente dersom de inneholder de samme operasjonene, og i tillegg oppfyller følgende krav for alle par av operasjoner (o_k, o_l) i konflikt:

$o_k <_p o_l \Leftrightarrow o_k <_q o_l$
$o_l <_p o_k \Leftrightarrow o_l <_q o_k$

Med andre ord må alle par av operasjoner som er i konflikt utføres i samme rekkefølge i p og q.

Konfliktserialiserbarhet

En plan p er konfliktserialiserbar hvis og bare hvis det finnes en seriell plan p_s som er konfliktekvivalent med p.

Eksempel 2.3

Vi har følgende plan for utførelsen av to transaksjoner $t_i = r_i(y)\ w_i(x)$ og $t_j = r_j(x)\ w_j(x)$:

$p = r_i(y)\ r_j(x)\ w_j(x)\ w_i(x)\ c_i\ c_j$

Vi har også en seriell plan $p_s = r_j(x)\ w_j(x)\ c_j\ r_i(y)\ w_i(x)\ c_i$, hvor $(r_j(x), w_i(x))$ og $(w_j(x), w_i(x))$ inngår i $<_p$ og $<_{p_s}$. Dermed er p konfliktekvivalent med p_s, og p er konfliktserialiserbar.

Konfliktgrafer

La p være en plan for en mengde transaksjoner T. En konfliktgraf for p, også kalt serialiseringsgraf, illustrerer hvilke transaksjoner som er i konflikt i p, og i hvilken rekkefølge konfliktoperasjonene utføres. Vi definerer konfliktgrafen slik:

La p være en plan for en mengde transaksjoner T. Konfliktgrafen for p er en graf $G = (N, E)$, hvor N og E representerer henholdsvis nodene og kantene i G. N består av alle transaksjoner som committes i p, og det inngår en kant (t_i, t_j) i E hvis og bare hvis $i \neq j$ og det finnes to operasjoner o_i, o_j, hvor $o_i <_p o_j$ og o_i, o_j er i konflikt.

En viktig egenskap ved konfliktgrafer er at en plan p er serialiserbar hvis og bare hvis konfliktgrafen for p er asyklisk:

Eksempel 2.4

Vi har tre transaksjoner $t_i = r_i(x)\ w_i(x)$, $t_j = r_j(x)\ r_j(y)\ w_j(y)$ og $t_k = r_k(x)\ r_k(y)$. Da er p en mulig plan for utførelsen av t_i, t_j og t_k:

$p = r_i(x)\ r_k(x)\ w_i(x)\ c_i\ r_j(x)\ r_j(y)\ w_j(y)\ c_j\ r_k(y)\ c_k$

Konfliktgrafen for p er illustrert i figur 2.1.

Siden konfliktgrafen er syklisk, er ikke p serialiserbar.

2.4.2 Viewekvivalens og viewserialiserbarhet

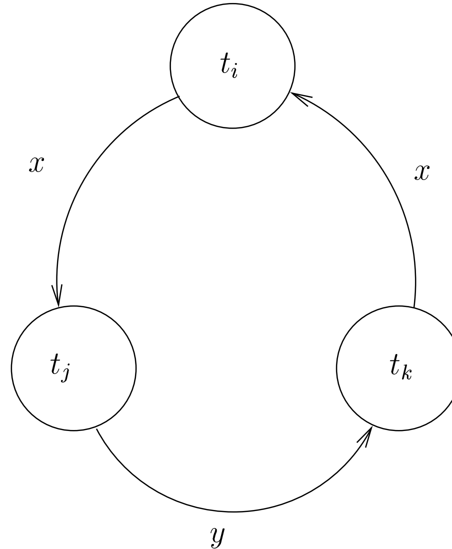
For å definere viewekvivalens trenger vi å innføre *lese-fra-relasjonen* for en plan p:

La p være en plan for en mengde transaksjoner T, hvor t_i og t_j er to transaksjoner i T, og hvor $r_j(x)$ og $w_i(x)$ inngår i p. Transaksjon t_j leser x fra t_i i p hvis og bare hvis t_j leser verdien av x som ble skrevet av t_i.

Lese-fra-relasjonen LF for en plan p er definert slik:

$LF(p) = \{\ (t_i, x, t_j) \mid \exists (r_j(x), w_i(x))\ \wedge\ r_j(x)$ leser fra $w_i(x)$ i $p\}$

Definisjonen av lese-fra-relasjonen krever at det, for hvert objekt som leses i planen, finnes en transaksjon i p som tidligere skriver det aktuelle objektet. Vi innfører derfor en *initialiseringstransaksjon* for en plan, dvs. en transaksjon t_0 som fullføres før noen andre transaksjoner startes, og hvor alle objekter som leses av operasjoner i planen, skrives i t_0.



Figur 2.1: Konfliktgraf for p

For å kunne definere viewserialiserbarhet trenger vi også en *avslutningstransaksjon*. Avslutningstransaksjonen t_∞ for en plan p utføres etter alle andre transaksjoner i p, og leser alle objekter som er skrevet av operasjoner i p. En plan $p = r_i(x)\ r_j(y)\ w_i(x)\ c_i\ c_j$ kan da skrives slik:

$p = w_0(x)\ w_0(y)\ c_0\ r_i(x)\ r_j(y)\ w_i(x)\ c_i\ c_j\ r_\infty(x)\ r_\infty(y)\ c_\infty$

$LF(p)$ blir: $\{(t_0, x, t_i), (t_0, y, t_j), (t_i, x, t_\infty)\}$.

Initialiserings- og avslutningstransaksjonene blir vanligvis ikke vist eksplisitt.

Viewekvivalens og viewserialiserbarhet

To planer p og q er viewekvivalente dersom de inneholder de samme operasjonene og $LF(p) = LF(q)$.

En komplett plan p er *viewserialiserbar* hvis og bare hvis det finnes en seriell plan p_s som er viewekvivalent med p.

Eksempel 2.5

Vi bruker samme eksempel som for konfliktserialiserbarhet, og p er da en plan for utførelsen av to transaksjoner $t_i = r_i(y)\ w_i(x)$ og $t_j = r_j(x)\ w_j(x)$:

$p = r_i(y)\ r_j(x)\ w_j(x)\ w_i(x)\ c_i\ c_j$

$LF(p) = \{(t_0, x, t_j), (t_0, y, t_i), (t_i, x, t_\infty)\}$

Dersom vi misbruker notasjonen litt, kan vi la $p_s = t_j \; t_i$ representere den serielle planen hvor t_j i sin helhet utføres før t_i. Lese-fra-relasjonen for p_s blir da:

$LF(p_s) = \{(t_0, x, t_j), (t_0, y, t_i), (t_i, x, t_\infty)\}$

Siden $LF(p) = LF(p_s)$, er p viewekvivalent med p_s, og p er viewserialiserbar.

Det lar seg vise at dersom en plan p er konfliktekvivalent med en seriell plan p_s, er også lese-fra-relasjonene for de to planene like. Følgelig er alle konfliktserialiserbare planer også viewserialiserbare. At det motsatte ikke gjelder, ser vi av følgende plan:

$p = w_i(x) \; w_j(x) \; w_j(y) \; c_j \; w_i(y) \; c_i \; w_k(x) \; w_k(y) \; c_k$

p er ikke konfliktserialiserbar, siden $w_i(x) <_p w_j(x)$ og $w_j(y) <_p w_i(y)$. $LF(p) = \{(t_k, x, t_\infty), (t_k, y, t_\infty)\}$, og p er derfor viewekvivalent med to serielle planer $p_{s1} = t_i \; t_j \; t_k$ og $p_{s2} = t_j \; t_i \; t_k$.

Testing av serialiserbarhet

For en gitt plan p kan vi alltid avgjøre i polynomisk tid om p er konfliktserialiserbar, mens en slik generell test for viewserialiserbarhet er et NP-komplett problem [WV01, side 86]. I praksis vil derfor en planlegger nøye seg med å generere konfliktserialiserbare planer. Vi skal imidlertid se at viewserialiserbarhet er et nyttig teoretisk hjelpemiddel, særlig når vi skal presentere kriterier for multiversjonsserialiserbarhet.

2.5 Algoritmer for samtidighetskontroll

Det finnes mange ulike strategier for samtidighetskontroll, og vi begrenser oss derfor til algoritmer som:

- har *generell anvendelse*, dvs. at de ikke er konstruert for å støtte bestemte aksessmønstre.
- tilbyr serialiserbare transaksjoner.

For mange applikasjoner er serialiserbarhet et unødig strengt isolasjonskrav. I sin kritikk av ANSI SQL-standardens behandling av samtidighetskontroll identifiserer [BBG+95] i alt åtte ulike *samtidighetsanomalier*, dvs. situasjoner som bare kan oppstå når flere transaksjoner utføres samtidig. Ut fra dette defineres seks ulike *isolasjonsnivåer*, og algoritmer for samtidighetskontroll kan da klassifiseres ut fra hvilket isolasjonsnivå de tilbyr. Så lenge transaksjoner kan avbrytes, er ikke en planlegger uten videre immun mot alle slike samtidighetsanomalier selv om den garanterer serialiserbar eksekvering:

Eksempel 2.6

La p være en plan for to transaksjoner t_i og t_j:

$p = r_i(x)\ w_i(x)\ r_j(x)\ w_j(x)\ c_j\ a_i$

I henhold til vår definisjon er p konfliktserialiserbar, men eksekveringen kan på ingen måte sees som korrekt. Siden t_j leser x fra t_i, som senere avbrytes, må vi anta at x-verdien som skrives avhenger av en transaksjon som i henhold til atomisitetskravet aldri er utført.

Dette kalles en *dirty read*, dvs. en situasjon hvor en transaksjon leser et objekt fra en ikke-committet transaksjon. Vi kan likevel garantere korrekt utførelse dersom vi ikke tillater en transaksjon å committes før alle transaksjoner den leser fra også er committet. Ulempen med dette er muligheten for *kaskadeavbrudd*, dvs. at man, idet en skrivetransaksjon t_k avbrytes, også må avbryte alle andre transaksjoner som har lest fra t_k.

En algoritme kalles *strikt* hvis den krever at en transaksjon må være committet for at andre transaksjoner skal kunne lese fra den. Dette gir også beskyttelse mot kaskadeavbrudd, og selv om vi her antar at serialiserbarhet er et tilstrekkelig krav til korrekt eksekvering, kan alle algoritmene som presenteres modifiseres til å tilby strikthet.

2.5.1 Tofase-låsing

Tofase-låsing er basert på at transaksjonene må låse objekter de leser eller skriver, og at en transaksjon ikke kan «låse opp» objekter før den har fått alle låsene den trenger.

Vi har to typer lås:

- En transaksjon ber om *leselås* idet den skal lese verdien av et objekt, og vilkårlig mange transaksjoner kan ha leselås på et objekt samtidig.

- En transaksjon ber om en *skrivelås* idet den skal skrive en ny verdi for et objekt. Ingen andre transaksjoner kan ha noen form for låser på et objekt x så lenge det finnes en skrivelås på x, og en transaksjon kan både lese og skrive alle objekter den har låst for skriving. En konsekvens av dette er at vi kan gå ut fra at en transaksjon aldri har mer enn én lås på et objekt.

Dersom en transaksjon t_j ber om en lås, og det allerede finnes en annen transaksjon t_i som har en lås som er *inkompatibel* (se tabell 2.5.1) med låsen t_i ber om, settes t_i til å vente:

	$rlock_i(x)$	$wlock_i(x)$
$rlock_j(x)$	kompatibel	*inkompatibel*
$wlock_j(x)$	*inkompatibel*	*inkompatibel*

Tabell 2.1: Låsekompatibilitet

Eksempel 2.7

Vi innfører to låseoperasjoner, $rlock$ og $wlock$, for å angi at en transaksjon ber om henholdsvis lese- og skrivelås på et objekt. I tillegg innfører vi en operasjon for fjerning av låser, $unlock$. Notasjonen $rlock_i(x)$ skal da tolkes slik: «t_i ber om en leselås på objekt x», mens $unlock_i(x)$ betyr «transaksjon t_i fjerner sin lås på x». Vi inkluderer disse operasjonene i eksekveringsplanene, slik at p er en mulig plan for to transaksjoner $t_i = r_i(x)\ r_i(y)$ og $t_j = w_j(x)$:

$$p = rlock_i(x)\ r_i(x)\ wlock_j(x)\ r_i(y)\ unlock_i(x)\ c_i\ w_j(x)\ c_j$$

t_i's leselås på x gjør at t_j må vente med å skrive til t_i fjerner sin lås.

En transaksjon kan aldri fjerne en lås på et objekt før den er ferdig med å lese eller skrive det aktuelle objektet. I tillegg kan, som nevnt, ingen låser fjernes før transaksjonen har låst alle objekter den trenger i løpet av eksekveringstiden. Alle transaksjoner får derfor en *økefase* og en *minkefase* med hensyn på låser de innehar, derav navnet *tofase-låsing*.

Eksempel 2.8

La p være en plan for utførelsen av tre transaksjoner $t_i = r_i(x)\ w_i(x)$, $t_j = r_j(x)\ r_j(y)\ w_j(y)$ og $t_k = r_k(x)\ r_k(y)$:

$$p = rlock_i(x)\ r_i(x)\ rlock_k(x)\ r_k(x)\ rlock_j(x)\ r_j(x)\ wlock_i(x)\ rlock_j(y)\ r_j(y)\\ wlock_j(y)\ w_j(y)\ c_j\ rlock_k(y)\ r_k(y)\ c_k\ w_i(x)\ c_i$$

p er konfliktekvivalent med den serielle planen $p_s = t_j\ t_k\ t_i$, og er følgelig serialiserbar.

Konfliktserialiserbarhet

Tofase-låsing garanterer konfliktserialiserbare planer. Hovedideene i et bevis for dette er som følger:

La t_i være en transaksjon som inngår i en plan p. Transaksjonene i p kontrolleres ved hjelp av tofase-låsing. Hvis vi antar at t_i committer, vil det finnes et tidspunkt hvor t_i har en lås på alle objektene i som leses eller skrives av operasjoner i t_i. Vi kaller dette t_i's *låsepunkt*[3], *LP*$_i$.

[3]Dette kalles også t_i's serialiseringspunkt.

La K_{ij} betegne *konfliktmengden* for t_i og t_j, hvor et objekt x inngår i K_{ij} hvis og bare hvis det finnes et par av operasjoner $o_i(x)$ og $o_j(x)$ i konflikt.

Da har vi for alle transaksjoner t_j i konflikt med en transaksjon t_i i p, to muligheter:

1. t_j må ha låst, og således aksessert, alle objektene i K_{ij} *før LP*$_i$. Følgelig serialiseres t_j foran t_i.

2. t_j har ingen låser på objekter i K_{ij} før *LP*$_i$. Transaksjon t_i må derfor aksessere alle objekter i K_{ij} før t_i kan aksessere noen, og t_i serialiseres foran t_j.

Dette gir en total ordning av alle transaksjoner i konflikt, og konfliktgrafen for en plan generert med tofase-låsing kan følgelig aldri bli syklisk.

For å oppnå strikt eksekvering benytter man i praksis vanligvis *strikt* tofase-låsing, som krever at en transaksjon beholder alle skrivelåser inntil den avsluttes. Det er lett å se at dette forhindrer *dirty read*, og dermed kaskadeavbrudd.

I en annen variant, kalt *sterk* tofase-låsing, holder en transaksjon både lese- og skrivelåser inntil den avsluttes. Dette sikrer at dersom det finnes en kant fra en transaksjon t_i til en transaksjon t_j i konfliktgrafen, må t_j committes etter t_i. Et eksempel på en situasjon hvor vi utnytter denne egenskapen finnes i seksjon 2.7.2.

2.5.2 Vranglåser

En ulempe med tofase-låsing er faren for *vranglås*:

Eksempel 2.9

Anta vi har to transaksjoner $t_i = r_i(y)\ w_i(x)$ og $t_j = r_j(x)\ w_j(y)$. Da er p begynnelsen på en mulig plan for utførelsen av t_i og t_j:

$p = rlock_i(y)\ r_i(y)\ rlock_j(x)\ r_j(x)\ wlock_i(x)\ wlock_j(y)$

Transaksjon t_i venter på t_j's leselås på x, mens t_j må vente på t_i's leselås på y. t_i og t_j er da i vranglås. Etter at en vranglås er oppstått, er den eneste utveien å avbryte, og evt. restarte, en av de involverte transaksjonene.

Hvordan oppdage vranglåser?

Vranglåser kan oppdages ved hjelp av en *vente-på-graf* (engelsk. *wait-for-graph*), definert slik:

Vente-på-grafen for en mengde transaksjoner T er en rettet graf hvor hver node representerer en AKTIV transaksjon, og hvor det går en kant fra en transaksjon t_i

til en transaksjon t_j dersom t_i venter på at t_j skal låse opp et objekt. Vente-på-grafen for T vil da inneholde en sykel hvis og bare hvis to eller flere av transaksjonene i T er i vranglås.

Vi kan ha vilkårlig store sykler i en slik graf, og selv om eksempel 2.9 viser en vranglås som involverer to transaksjoner, kan et vilkårlig antall transaksjoner inngå. Det vil likevel alltid være nok å fjerne én av transaksjonene for å bryte opp den aktuelle vranglåsen.

Ved hjelp av en vente-på-graf kan vranglåser oppdages, i prinsippet idet de oppstår, dersom vi sjekker grafen ved hver låseforespørsel. Siden låsing av objekter forekommer relativt hyppig, kan dette være for kostbart. Et alternativ er å gå gjennom grafen periodisk, med et passende valgt tidsintervall.

Fjerning av vranglåser

Etter at vranglåsen er oppdaget, må vi velge hvilken transaksjon vi skal avbryte. Aktuelle utvalgskriterier er bl.a. prosesseringstid, oppstartstid eller hvor mange sykler en transaksjon deltar i i vente-på-grafen. Det er også mulig å velge en tilfeldig transaksjon. Det viktigste er uansett å unngå *utsulting*, dvs at en eller flere transaksjoner gang på gang blir restartet.

Andre metoder

I stedet for å vedlikeholde en eksplisitt vente-på-graf og håndtere vranglåser etter at de har oppstått, kan man eliminere situasjoner som fører til vranglås. Anta at alle transaksjoner gis et unikt, monotont voksende tidsstempel ved oppstart, og la $ts(t_i)$ betegne tidsstempelet til en transaksjon t_i. Vranglåseliminering kan da gjøres slik:

1. *Wait-die*: hvis en låseforespørsel for en transaksjon t_i vil medføre en kant i vente-på-grafen fra t_i til en transaksjon t_j, håndteres dette slik: Hvis $ts(t_i) < ts(t_j)$, venter t_i på t_j. Hvis derimot $ts(t_j) < ts(t_i)$, restartes t_i. Med andre ord kan transaksjoner bare vente på yngre transaksjoner, og vente-på-grafen kan aldri bli syklisk.

2. *Wound-wait*: hvis en låseforespørsel for en transaksjon t_i vil medføre en kant i vente-på-grafen fra t_i til en transaksjon t_j, håndteres dette slik: Hvis $ts(t_i) < ts(t_j)$, restartes t_j. Hvis derimot $ts(t_j) < ts(t_i)$, venter t_i. En transaksjon kan bare bli satt til å vente av eldre transaksjoner, og vente-på-grafen blir aldri syklisk.

En annen, svært enkel, strategi for vranglåshåndtering er å starte en nedtelling idet en transaksjon settes til å vente på en lås. Dersom nedtellingen er ferdig før lå-

sen kan tildeles, antar databasesystemet at transaksjonen er involvert i en vranglås, og restarter den. En slik strategi vil være særlig attraktiv i distribuerte systemer, siden den ikke krever noen form for kommunikasjon mellom noder.

Det er bare vedlikehold av en eksplisitt vente-på-graf som garanterer at vi kun avbryter transaksjoner som faktisk er involvert i en vranglås, både vranglåseliminering og nedtellingsur vil kunne fjerne transaksjoner som ikke er i vranglås. Dette må imidlertid veies mot ressurskravene for vente-på-grafer.

2.5.3 Samtidighetskontroll med tidsstempelordning

I stedet for låser, kan vi kontrollere utførelsen av transaksjonene ved hjelp av tidsstempler. Tidsstempelordning krever at enhver transaksjon t_i tilordnes et unikt, monotont voksende tidsstempel ved oppstart. Vi lar også her *ts*(i) betegne transaksjon t_i's tidsstempel. En planlegger kan garantere serialiserbarhet ved å håndheve følgende krav:

For alle par o_i, o_j av operasjoner i konflikt, skal o_i utføres før o_j hvis og bare hvis *ts*$(t_i) <$ *ts*(t_j).

At dette garanterer konfliktserialiserbar eksekvering er lett å se, siden tidsstemplene bestemmer serialiseringsrekkefølgen:

La t_k og t_l være to transaksjoner som inngår i en eksekveringsplan p. Da kan aldri konfliktgrafen for p inneholde en vei fra t_l til t_k dersom $ts(t_k) < ts(t_l)$.

Implementasjon

En tidsstempelbasert algoritme kan realiseres som følger:

Hvert databaseobjekt x har til enhver tid to tidstempler: et *lesetidsstempel*, $rts(x)$, som inneholder tidsstempelet til transaksjonen som sist leste objektet, og et *skrivetidsstempel*, $wts(x)$, som på samme måte spesifiserer hvilken transaksjon som sist skrev objektet.

Planleggeren må da, for alle operasjoner $o_i(x)$, sikre følgende:

- Hvis $o_i(x)$ er en leseoperasjon, kan t_i bare fortsette dersom $wts(x)$ er lavere enn $ts(t_i)$.
- Hvis $o_i(x)$ er en skriveoperasjon, kan t_i bare fortsette dersom $rts(x)$ og $wts(x)$ er lavere enn $ts(t_i)$.

Eksempel 2.10

La $t_i = r_i(y)\ w_i(y)$ og $t_j = r_j(x)\ r_j(y)$ være to transaksjoner, hvor $ts(t_i)$ er lavere enn $ts(t_j)$. Da er p en mulig plan for utførelsen av t_i og t_j:

$p = r_i(x)\ r_j(x)\ r_j(y)\ c_j\ w_i(y)\ a_i$

Vår algoritme krever at transaksjon t_i avbrytes, siden t_i forsøker å skrive y etter at t_j allerede har lest samme objekt.

En viktig fordel med tidsstempelordning er at man unngår vranglåser, siden transaksjonene aldri venter. Ulempen er illustrert i eksempelet over – med tofase-låsing ville t_i fått fortsette eksekveringen uforstyrret.

2.5.4 Samtidighetskontroll basert på konfliktgraf-testing

En tredje strategi, som garanterer konfliktserialiserbar eksekvering, er å vedlikeholde en konfliktgraf som definert i seksjon 2.4.1. Vi lar da alle transaksjoner representeres med en node i grafen, og kan, for hver operasjon en transaksjon t_i utfører, legge til evt. nye kanter og sjekke grafen for sykler. Dersom grafen fremdeles er asyklisk fortsetter t_i eksekveringen, hvis ikke, må t_i avbrytes og t_i's node, samt alle kanter inn til denne, fjernes fra grafen.

En algoritme som vedlikeholder en eksplisitt konfliktgraf er for kostbar til å være praktisk anvendbar. Slike grafer kan bli svært store, særlig fordi vi først kan fjerne en committet transaksjon t_i når vi vet at det ikke finnes noen aktiv transaksjon t_j hvor det går en vei fra t_j til t_i. I tillegg til plassbehovet vil tiden det tar å sjekke en slik graf for sykler forlenge transaksjonene, og en algoritme basert på f.eks. tofaselåsing vil vanligvis kunne utføre transaksjonene langt mer effektivt.

2.6 Multiversjons-samtidighetskontroll

Vi har implisitt antatt at når en skriveoperasjon oppdaterer et objekt x, slettes den tidligere verdien til objektet. Det er ingen grunn til at det må være slik, og svært mange moderne databasesystemer tar vare på tidligere versjoner, slik at alle skriveoperasjoner i praksis oppretter et nytt objekt.

Vi avgrenser diskusjonen til databaser som tilbyr *versjonstransparens*, dvs. at i motsetning til systemer for f.eks. versjonskontroll kan ikke brukeren eksplisitt be om bestemte versjoner av objekter. I den videre diskusjonen begrenser vi termen *multiversjonsdatabase* til å gjelde slike databaser, og formålet er utelukkende å gi bedre ytelse ved at transaksjoner som leser «for sent» kan lese eldre versjoner, illustrert ved følgende eksempel:

Eksempel 2.11

Vi har to transaksjoner $t_i = r_i(x)\ r_i(y)\ w_i(x)$ og $t_j = r_j(y)\ w_j(y)\ r_j(x)$, som begge utføres i en multiversjonsdatabase. Vi identifiserer en bestemt versjon ved hjelp av transaksjonen som opprettet den. Dersom en transaksjon t_k skriver et objekt

x, representeres denne versjonen med x_k i notasjonen. En mulig plan for utførelsen av t_i og t_j er da:

$p = r_j(y_0)\ w_j(y_j)\ r_i(x_0)\ r_i(y_j)\ w_i(x_i)\ r_j(x_0)$

Legg merke til at t_j leser x fra t_0 i den siste operasjonen. Lese-fra-relasjonen $LF(p)$ for p blir: $\{(t_0, y, t_j), (t_j, y, t_i), (t_0, x, t_i)\}$, og p er derfor viewekvivalent med den serielle planen $p_s = r_j(y_0)\ w_j(y_j)\ r_j(x_0)\ r_i(x_0)\ r_i(y_j)\ w_i(x_i)$. Utførelsen er derfor intuitivt korrekt. Det forutsetter imidlertid at transaksjon t_j har mulighet til å lese x_0 – siden t_i leser y fra t_j, kan ikke planen serialiseres dersom t_j leser x fra t_i.

2.6.1 Multiversjonsplaner og multiversjonsserialiserbarhet

For å definere multiversjonsserialiserbarhet trenger vi en formell definisjon av *multiversjonsplaner*. Vi innfører først begrepet *versjonsfunksjon*:

La p være en plan. En versjonsfunksjon for p er en funksjon h som assosierer alle leseoperasjoner i p med en foregående skriveoperasjon, mens alle skriveoperasjoner assosieres med seg selv.

Som tidligere lar vi O_T og A_T representere henholdsvis operasjonene og avslutningsoperasjonene i en mengde transaksjoner T.

Vi lar mengden H_T representere en mengde hvor h er utført på hver operasjon i O_T.

Multiversjonsplaner kan da defineres slik:

Multiversjonsplaner

En multiversjonsplan m, for en mengde transaksjoner T, består av en mengde $VA_T \subseteq (H_T \cup A_T)$, og en partiell ordning $<_m$ over VA_T.

For alle par av operasjoner $q <_i r$ i en transaksjon t_i har vi at dersom t_i inngår i m, må $h(q) <_m h(r)$. Dersom avslutningsoperasjonen for t_i inngår i m, må i tillegg alle operasjoner o_i i m ordnes slik at $o_i <_m \{c_i, a_i\}$.[4]

En multiversjonsplan m er komplett dersom den, for alle transaksjoner t_i som utfører operasjoner i p, oppfyller følgende:

- c_i eller a_i inngår i m.
- Det finnes ingen operasjoner i t_i som ikke utføres i m.

[4] [WV01] krever også at hvis en operasjon $r_j(x)$ assosieres med en skriveoperasjon $w_i(x_i)$ i en multiversjonsplan m og c_j inngår i m, må c_i inngå i m og $c_i <_m c_j$. Selv om dette i praksis er nødvendig for å sikre korrekt eksekvering ved transaksjonsavbrudd, utelater vi kravet i definisjonen av multiversjonsplaner.

Vi kan også kalle en komplett multiversjonsplan en *multiversjonshistorie*.

Vi forutsetter også her at alle planer innledes av en initialiseringstransaksjon, slik at det alltid vil finnes en skriveoperasjon som leseoperasjoner kan lese fra.

Et spesialtilfelle av multiversjonsplaner er *monoversjonsplaner*. En monoversjonsplan er en multiversjonsplan hvor alle leseoperasjoner må assosieres med den transaksjonen som sist skrev objektet som leses. Alle ordinære planer kan sees som monoversjonsplaner.

Å kreve at en multiversjonsplan m er ekvivalent med en seriell *multiversjonsplan* er ikke tilstrekkelig til å garantere korrekt utførelse, siden vi da ikke setter noen begrensninger for hvilke versjoner leseoperasjonene i m faktisk leser. Multiversjonsserialiserbarhet er derfor definert slik:

Multiversjonsserialiserbarhet

Dersom en komplett multiversjonsplan m, for et passende valgt ekvivalenskrav, er ekvivalent med en seriell *monoversjonsplan* m_s, er m multiversjonsserialiserbar.

2.6.2 Konflikter i multiversjonsplaner

Det finnes dessverre ikke noen fullgod definisjon av konfliktserialiserbarhet for en multiversjonsplan. Konfliktekvivalens med en seriell plan fungerer som serialiserbarhetskrav i monoversjonsplaner fordi rekkefølgen av operasjoner i konflikt også definerer et krav til serialiseringsrekkefølge for transaksjonene.

Selv om et par av operasjoner representerer en konflikt i en monoversjonsplan, gjelder ikke dette uten videre for multiversjonsplaner:

Eksempel 2.12

La m være en multiversjonsplan definert for tre transaksjoner t_i, t_j og t_k:

$m = w_i(x_i)\ c_i\ w_j(x_j)\ c_j\ r_k(x_i)\ c_k$

I prinsippet kan t_k «velge» hvor den vil serialiseres, og m er viewekvivalent med to serielle monoversjonsplaner, $p_{s1} = t_j\ t_i\ t_k$ og $p_{s2} = t_i\ t_k\ t_j$. Hverken $(w_i(x_i), w_j(x_i))$ eller $(w_j(x_j), r_k(x_i))$ begrenser serialiseringsrekkefølgen for de involverte transaksjonene, og de kan derfor ikke kalles konflikter.

Den eneste situasjonen som faktisk representerer en tradisjonell konflikt i en multiversjonsplan, er når en transaksjon t_j leser en *versjon* av et objekt som en annen transaksjon t_i har skrevet. Dette krever selvsagt at t_j serialiseres etter t_i.

Dessverre kan vi ikke garantere konsistent eksekvering ved å kreve at alle operasjoner som er i konflikt i en multiversjonsplan skal ordnes i samme rekkefølge som i en seriell plan:

Eksempel 2.13

La m være en multiversjonsplan:

$m = w_i(x_i)\ w_i(y_i)\ c_i\ w_j(x_j)\ w_j(y_j)\ c_j\ r_k(x_j)\ r_k(y_i)\ c_k$

Siden t_k leser x fra t_i og y fra t_j, mens t_i og t_j skriver både x og y, kan ikke m være viewekvivalent med noen seriell plan.

Dersom vi imidlertid bare ser på rekkefølgen i konfliktene i m, $(w_i(y_i), r_k(y_i))$ og $(w_j(x_j), r_k(x_j))$, ville m være konfliktekvivalent med de serielle planene $p_{s1} = t_i\ t_j\ t_k$ og $p_{s2} = t_j\ t_i\ t_k$.

Hvis vi i stedet ser bort fra versjonene, og bare krever en serialiserbar ordning av operasjoner som opererer på samme objekt, er dette heller ikke nok til å garantere serialiserbar eksekvering: operasjonene i m utføres serielt, og ut fra dette kravet ville m være serialiserbar.

Vi har ikke noen definisjon av konflikt som gir oss mulighet til å avgjøre om en gitt multiversjonsplan faktisk er serialiserbar. I stedet vil man, som vi uformelt har gjort flere ganger, vanligvis bruke viewserialiserbarhet som korrekthetskrav for multiversjonsplaner.[5]

2.6.3 Multiversjons-viewserialiserbarhet

Siden lese-fra-relasjonen for en multiversjonsplan kan defineres på samme måte som for monoversjonsplaner (se seksjon 2.4.2), er viewekvivalens for multiversjonsplaner definert som for monoversjonsplaner:

La m og m' være to multiversjonsplaner for samme mengde transaksjoner. m og m' er $viewekvivalente$ hvis $LF(m) = LF(m')$.

Vi har allerede vist at det ikke er nok å kreve ekvivalens med en multiversjonsplan for å sikre konsistent eksekvering, og vi har derfor følgende krav til multiversjonsviewserialiserbare planer:

En multiversjonsplan m, for en mengde transaksjoner T, er multiversjons-viewserialiserbar dersom det eksisterer en seriell monoversjonsplan m_s for T, hvor m er viewekvivalent med m_s.

[5] [HP85] kaller alle lese-skrive-konflikter *multiversjonskonflikter*, og viser at det for en multiversjonsplan m finnes en eller flere versjonsfunksjoner som gir multiversjonsserialiserbarhet hvis og bare hvis m er multiversjons-konfliktekvivalent med en seriell plan. Men det er fremdeles et NP-komplett problem å avgjøre *hvilke* versjonsfunksjoner som gir serialiserbar eksekvering, og dette er derfor ikke noen praktisk anvendbar test for serialiserbarhet.

2.6.4 Multiversjons-serialiseringsgrafer

[Had88] definerer en en *multiversjons-serialiseringsgraf*, dvs. en graf som garanterer serialiserbarhet for en gitt multiversjonsplan dersom grafen er asyklisk. Det finnes imidlertid ikke bare én slik serialiseringsgraf for en gitt multiversjonsplan, og vi må derfor innføre begrepet *versjonsordning*:

For en multiversjonsplan m kan vi, for hvert objekt x som skrives i m, definere en versjonsordning $\ll$, dvs. en total ordning av alle versjonene av x som opprettes i m. En versjonsordning for m, kalt $\ll_m$, er definert som unionen av versjonsordningene til alle objektene som oppdateres i m.

Versjonsordningen for en multiversjonsplan er ikke unik, og har i utgangspunktet ingen sammenheng med hvilken rekkefølge versjonene opprettes i. I stedet representerer versjonsordningen for en multiversjonsplan m et «forslag» til en serialiseringsrekkefølge for transaksjonene i m. Anta vi har tre transaksjoner t_i, t_j og t_k som alle skriver et objekt x, og la en fjerde transaksjon t_l lese x fra t_j. En mulig versjonsordning for x er $x_k \ll x_j \ll x_i$, og denne krever at t_k serialiseres foran t_j, som igjen serialiseres foran t_i. Versjonsordningen vil da også plassere t_l: siden t_l leser fra t_j, må den serialiseres etter t_k og t_j, men før t_i.

Vi kan da, for en plan m og en versjonsordning $\ll_m$, definere en multiversjons-serialiseringsgraf:

Multiversjons-serialiseringsgrafen, for en komplett multiversjonsplan m og en multiversjonsordning $\ll_m$, består av en mengde noder N og en mengde kanter E, hvor N inneholder alle transaksjonene som committes i m. Kantene i E er definert slik:

1. For hvert par av operasjoner $w_i(x_i), r_j(x_i)$ i m går det en kant fra t_i til t_j.
2. For hvert par av operasjoner $r_k(x_j), w_i(x_i)$ i m, hvor k, i og j er forskjellige og $x_i \ll_m x_j$, går det en kant fra t_i til t_j.
3. For hvert par av operasjoner $r_k(x_j), w_i(x_i)$ i m, hvor k, i og j er forskjellige og $x_j \ll_m x_i$, går det en kant fra t_k til t_j.

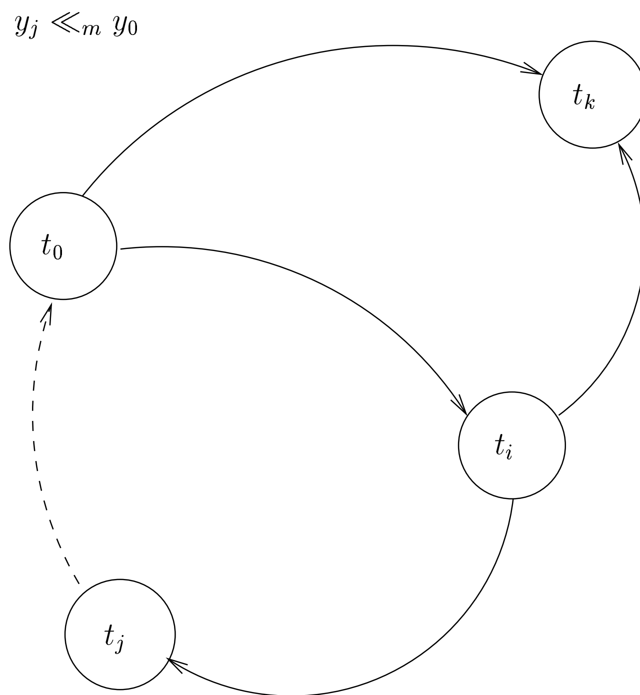
Eksempel 2.14

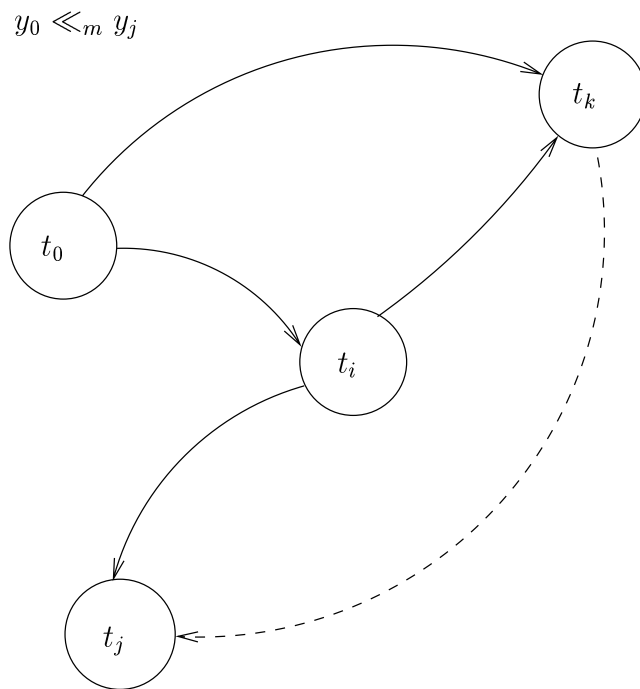
Anta vi har følgende multiversjonsplan:

$m = r_i(x_0)\ w_i(x_i)\ c_i\ r_j(x_i)\ w_j(y_j)\ c_j\ r_k(x_i)\ r_k(y_0)$

I en multiversjonsplan kan vi ikke gå ut fra at initialiseringstransaksjonen t_0 serialiseres først, og figur 2.2 viser multiversjons-serialiseringsgrafen for to ulike versjonsordninger av y. Den stiplede kanten avhenger av denne ordningen, og vi ser at grafen bare er asyklisk dersom $y_0 \ll_m y_j$.

En multiversjonsplan m er multiversjons-viewserialiserbar hvis og bare hvis det finnes en versjonsordning $\ll_m$ som er slik at multiversjons-serialiseringsgrafen for m og $\ll_m$ er asyklisk.





Figur 2.2: Multiversjons-serialiseringsgrafer

2.7 Algoritmer som tilbyr multiversjonsserialiserbarhet

På samme måte som for monoversjonsplaner har vi ingen kjent metode for å kunne avgjøre om en multiversjonsplan m er multiversjons-viewserialiserbar i polynomisk tid. Dette følger direkte av at en monoversjonsplan er et spesialtilfelle av en multiversjonsplan.

Men å avgjøre om multiversjons-serialiseringsgrafen er asyklisk, for en multiversjonsplan m og en bestemt versjonsordning $\ll_m$, lar seg gjøre i polynomisk tid. Planleggere som tilbyr multiversjonsserialiserbarhet benytter derfor en på forhånd definert versjonsordning. Det finnes flere slike algoritmer[WV01, kapittel 5], og her presenteres to av dem:

2.7.1 Multiversjons-tidsstempelordning

I likhet med ordinær tidsstempelordning sikrer multiversjons-tidsstempelordning at eksekveringen blir ekvivalent med en seriell monoversjonsplan hvor transaksjonene utføres i tidsstempelrekkefølge. Alle transaksjoner tildeles et unikt, monotont voksende tidsstempel ved oppstart, og alle versjoner må merkes med tidsstempelet til transaksjonen som opprettet den.

Algoritmen utfører operasjonene i en transaksjon t_i slik:

1. En operasjon $r_i(x)$ assosieres med en versjon x_k, hvor x_k er den versjonen av x med det høyeste tidsstempelet blant de versjonene som har tidsstempel lavere enn $ts(t_i)$.
2. En skriveoperasjon $w_i(x)$ utføres slik:
   (a) Hvis det allerede er utført en operasjon $r_j(x_k)$, hvor $ts(t_k) < ts(t_i) < ts(t_j)$, må t_i avbrytes.
   (b) Hvis ikke, opprettes en ny versjon x_i.

Bevisskisse for serialiserbarhet

Som nevnt definerer algoritmen en versjonsordning hvor $x_i \ll x_j$ hvis og bare hvis $ts(t_i) < ts(t_j)$.

La m være en multiversjonsplan, og la t_i og t_j være to transaksjoner i m, hvor $ts(t_i) < ts(t_j)$. Vi antar at det går en kant i multiversjons-serialiseringsgrafen fra t_j til t_i.

En slik kant må da skyldes én av følgende:

- Transaksjon t_i har lest en versjon x_j av et objekt x. Men dette strider mot krav (1) i algoritmen.

- Transaksjon t_i skriver et objekt x, men transaksjon t_j leser x fra en transaksjon t_k, hvor $ts(t_k) < ts(t_i)$. Men dette krever enten at t_j ikke oppfyller krav (1), eller at t_i bryter krav (2).

Vi ser at tidsstemplene definerer en total ordning for serialiseringen av transaksjonene, og algoritmen garanterer global serialiserbarhet.

2.7.2 Tidsstempelordning med snapshot

En enkel form for multiversjons-samtidighetskontroll innebærer at alle transaksjoner som ikke på forhånd er deklarert som lesetransaksjoner koordineres ved hjelp av en konvensjonell samtidighetskontrollalgoritme, for eksempel tofaselåsing. Alle rene lesetransaksjoner leser derimot fra et *snapshot*, hvor vi ved hjelp av tidsstempler sikrer at dette bare inneholder committede transaksjoner.

En slik algoritme, basert på sterk tofase-låsing (se seksjon 2.5.1), utfører en transaksjon t_i som følger:

Dersom t_i er en *skrivetransaksjon*:

- Operasjonene i t_i kontrolleres ved hjelp av sterk tofase-låsing, dvs. at alle låser t_i holder først kan frigis idet t_i committes. Alle skriveoperasjoner oppretter en ny versjon, mens leseoperasjoner, på samme måte som i en monoversjonsplan, alltid assosieres med den siste versjonen som ble opprettet.
- Idet t_i committes, tildeles den et unikt, monotont voksende tidsstempel.

Dersom t_i er en *lesetransaksjon*:

- Ved oppstart tildeles t_i et tidsstempel $ts(t_i)$, hvor dette er identisk med tidsstempelet til den skrivetransaksjonen som sist ble committet
- Kravet til leseoperasjonene er da det samme som for ordinær multiversjonstidsstempelordning: En operasjon $r_i(x)$ assosieres med en versjon x_k, hvor x_k er den versjonen av x med det høyeste tidsstempelet blant de versjonene som har tidsstempel mindre enn eller lik $ts(t_i)$.

Bevisskisse for serialiserbarhet

Algoritmen er basert på en versjonsordning hvor skrivetransaksjonene, og dermed versjonene de oppretter, ordnes i commitrekkefølge.

En lesetransaksjon t_j plasseres i serialiseringsrekkefølgen ut fra tidsstempelet, slik at den serialiseres etter alle skrivetransaksjoner t_i, hvor $ts(t_i) \leq ts(t_j)$, men før alle skrivetransaksjoner t_k, hvor $ts(t_j) < ts(t_k)$.

Hvis vi antar at vi har en sykel s i multiversjons-serialiseringsgrafen for en multiversjonsplan m, må det finnes en kant i s fra en transaksjon t_l til en transaksjon t_k, hvor $ts(t_k) < ts(t_l)$. En slik kant må skyldes en av følgende:

- Transaksjon t_k har lest en versjon x_l av et objekt x. Siden lesetransaksjoner aldri leser fra transaksjoner med høyere tidsstempel, må t_k være en skrivetransaksjon. Men sterk tofase-låsing krever at en transaksjon beholder alle låser til den committer, og siden t_k committes først, kan ikke t_k lese fra t_l.
- Transaksjon t_k skriver et objekt x, mens transaksjon t_l leser x fra en transaksjon t_m, hvor $ts(t_m) < ts(t_k)$. Heller ikke her kan t_l være en lesetransaksjon, siden den da ikke ville lese den versjonen av x som har det høyeste tidsstempelet som likevel er lavere enn $ts(t_l)$. Dersom t_l er en skrivetransaksjon, må t_k vente med å skrive x til t_l har fjernet sin leselås. Siden vi benytter sterk tofase-låsing innebærer dette at t_k ikke kan committes før t_l, og $ts(t_k)$ kan følgelig ikke være lavere enn $ts(t_l)$.

Snapshotalgoritmen og strikt tofase-låsing

Beviset for serialiserbarhet er basert på sterk tofase-låsing, siden denne garanterer at alle skrivetransaksjoner committes i serialiseringsrekkefølge. *Strikt* tofase-låsing, hvor bare skrivelåser må beholdes til transaksjonen committes, er ikke nok til å garantere dette, illustrert ved følgende monoversjonsplan:

$m = rlock_i(x)\ r_i(x)\ wlock_i(y)\ w_i(y)\ unlock_i(x)\ wlock_j(x)\ w_j(x)\ c_j\ c_i$

Siden t_i leser x før t_j's skriveoperasjon serialiseres t_i foran t_j, men t_j committes likevel først.

[WV01] krever bare strikt tofase-låsing for å koordinere skrivetransaksjonene i denne algoritmen. Dette betyr at man tillater følgende eksekveringsplan:

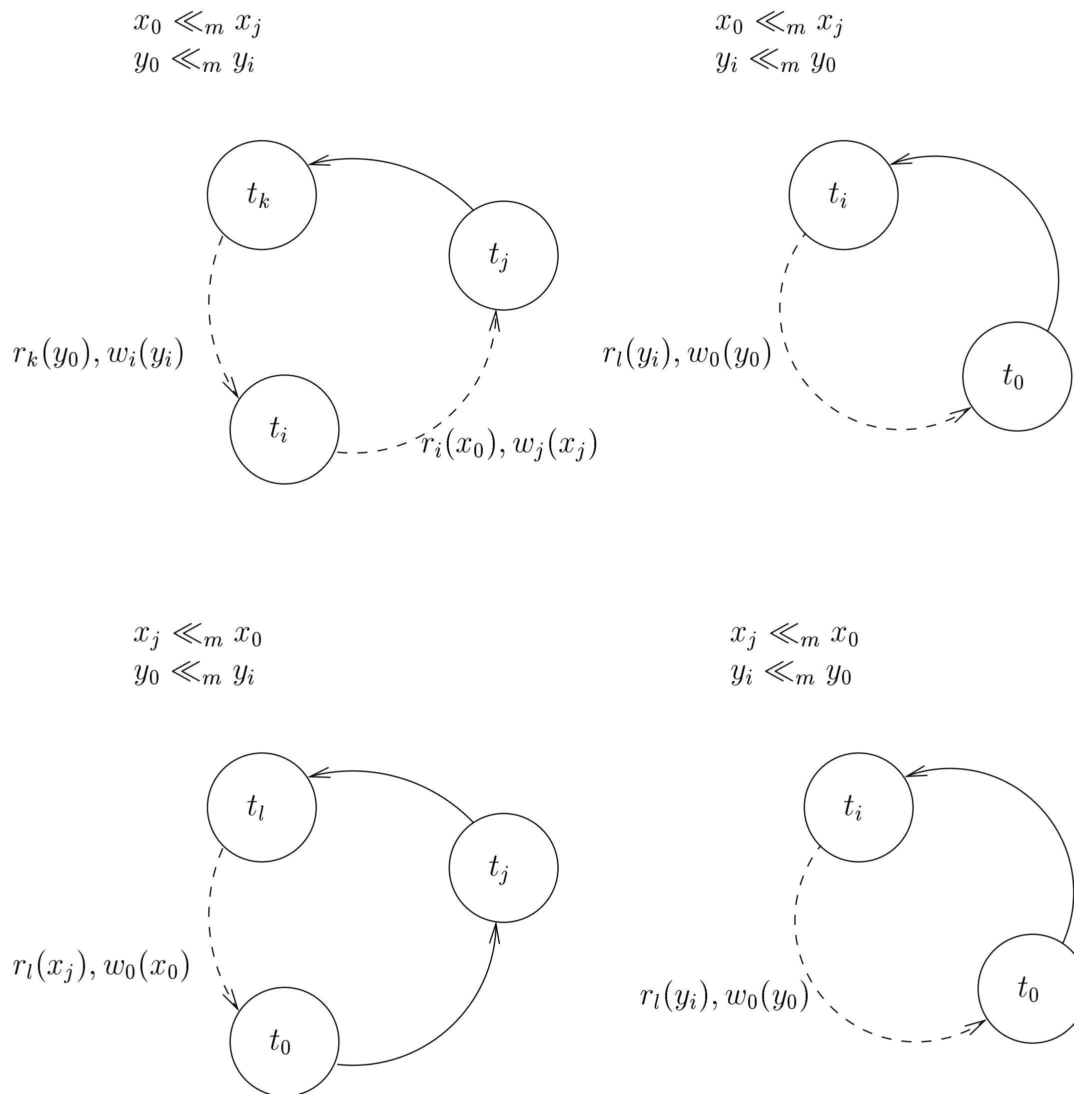
$m = r_i(x_0)\ w_i(y_i)\ unlock_i(x)\ w_j(x_j)\ c_j\ r_k(x_j)\ r_k(y_0)\ c_k\ c_i\ r_l(x_j)\ r_l(y_i)\ c_l$

t_i leser x fra t_0, og t_0 må derfor serialiseres før t_i. I tillegg betyr dette at t_j enten må serialiseres før t_0 eller etter t_i. Siden transaksjon t_l leser x fra t_j og y fra t_i, kan ikke t_j serialiseres før t_0. Men t_k leser x fra t_j og y fra t_0, og dette krever at t_j serialiseres før t_i (ellers skal t_k lese y fra t_i). Men vi har allerede eliminert dette alternativet, og m kan ikke være mulitversjons-viewserialiserbar.

Figur 2.3 illustrerer dette ved å vise at alle de fire mulige versjonsordningene for m gir en sykel i multiversjons-serialiseringsgrafen. De stiplede kantene er kanter som avhenger av versjonsordningen, mens de heltrukne representerer skrive-lese-konflikter.

Problemet oppstår sannsynligvis fordi [WV01] bruker [BHG87] som kilde, men i [BHG87] er strikt tofase-låsing definert annerledes: der er dette identisk med det som her, og i [WV01], kalles *sterk* tofase-låsing.[6]

[6] Et skille mellom det vi her kaller strikt og sterk tofase-låsing ble først innført i [BGRS91].



Figur 2.3: Multiversjons-serialiseringsgrafer for m

Snapshot for skrivetransaksjoner

Flere databasesystemer, deriblant Oracle, implementerer en variant av tidsstempelordning med snapshot. Denne algoritmen definerer to «nivåer»[BBG+95]:

- *Les-committet*, hvor en leseoperasjon leser den siste versjonen som er skrevet av en transaksjon som er committet idet operasjonen utføres, mens skriveoperasjoner kontrolleres med strikt tofase-låsing.
- *Snapshot-isolering*, hvor en leseoperasjon i en transaksjon t_i leser den siste versjonen som er skrevet av en transaksjon som var committet ved t_i's oppstart. Skrivemengdene til samtidige transaksjoner må være disjunkte, dvs. at for alle par av transaksjoner t_i og t_j som skriver et felles objekt x, må enten t_i være committet før t_j's oppstart, eller motsatt, dvs. at t_j er committet før t_i's oppstart.

Ingen av disse algoritmene garanterer multiversjonsserialiserbarhet, men snapshot-isolering er mer restriktiv, og gir noe bedre beskyttelse mot samtidighetsanomalier.

2.8 Optimistisk samtidighetskontroll

Alle algoritmene vi har presentert antar implisitt at konflikter skjer ofte, og at vi, idet en operasjon skal utføres, bør avgjøre hvorvidt eksekveringen forblir serialiserbar også etter at den er utført. Slike algoritmer kalles *pessimistiske*.

En annen mulighet er å utføre transaksjonene *optimistisk*:

- Operasjonene utføres uten noen form for samtidighetskontroll.
- Før en transaksjon kan committes må utførelsen *valideres*, dvs. at planleggeren må se om den tentative eksekveringsplanen er serialiserbar. Hvis valideringen feiler må transaksjonen restartes.

Det er vanlig å dele optimistisk utførelse av transaksjoner i tre faser: en *lesefase*, en *valideringsfase* og en *skrivefase*:

1. *Lesefase:* Transaksjonen utføres, men ingen oppdateringer lagres i databasen.
2. *Valideringsfase:* Før en transaksjon skal committes, utfører planleggeren en test for å se om eksekveringen forblir serialiserbar dersom transaksjonen committes. Hvis denne testen er negativ, restartes transaksjonen. Hvis den derimot er positiv, fortsetter transaksjonen til skrivefasen.

3. *Skrivefase:* alle eventuelle oppdateringer gjøres permanente, og transaksjonen committes.

[KR81] introduserer optimistisk samtidighetskontroll, og presenterer bl.a. følgende algoritme som garanterer konfliktserialiserbarhet:

Transaksjonene går gjennom nevnte tre faser. Alle transaksjoner tilordnes ved slutten av lesefasen et unikt, monotont voksende tidsstempel. Konfliktserialiserbarhet kan da sikres med følgende betingelse for validering av en transaksjon t_j: for alle transaksjoner t_j, hvor $ts(t_i) < ts(t_j)$, må ett av følgende krav være innfridd:

1. t_i fullfører sin skrivefase før t_j påbegynner sin lesefase.
2. Skrivemengden til t_i overlapper ikke med lesemengden til t_j, og t_i fullfører sin skrivefase før t_j påbegynner sin skrivefase.

Siden valideringen skjer *før* skrivefasen, må vi sikre at ikke krav (2) brytes under eller etter valideringen. Vi anta for enkelhets skyld at ingen andre transaksjoner utfører operasjoner mens en transaksjon er i validerings- og skrivefasen, selv om dette i praksis kan kreve mer sofistikerte løsninger.

2.8.1 Bevisskisse for serialiserbarhet

La t_i og t_j være to transaksjoner som utføres i en plan p, og anta $ts(t_i) < ts(t_j)$.

En kant fra t_j til t_i, krever én av følgende:

- Det finnes et par av operasjoner $r_j(x), w_i(x)$ i p, hvor $r_j(x) <_p w_i(x)$. Men siden t_i skriver x etter at t_j har påbegynt sin lesefase og t_j's lesemengde i tillegg overlapper med t_i's skrivemengde, kan ikke t_j valideres.
- Det finnes et par av operasjoner $w_j(x), w_i(x)$ eller $w_j(x), r_i(x)$, hvor $w_j(x) <_p w_i(x)$ eller $w_j(x) <_p r_i(x)$. Men dermed kan ikke t_i ha fullført sin skrivefase når t_j's skrivefase begynner, og t_j kan ikke valideres.

Betingelsene ovenfor sikrer at tidsstemplene definerer en total ordning av transaksjonene, og garanterer derfor global serialiserbarhet.

Kapittel 3

Samtidighetskontroll i distribuerte databaser

I kapittel 1 viste vi formålet med distribuerte databasesystemer og replikering. Her innfører vi serialiserbarhetskrav og strategier for samtidighetskontroll i slike systemer, og vi presenterer noen av utfordringene som oppstår når vi utfører distribuerte transaksjoner.

Vi er særlig interessert i replikerte databaser, men definisjonen av serialiserbare transaksjoner i distribuerte databaser er uavhengig av om dataene er replikert.[1] Vi begynner derfor med en generell presentasjon av distribuert samtidighetskontroll, men avslutter kapittelet med en egen seksjon om replikering. Mye av kapittelet omhandler velkjente begreper og strategier som omtales i [WV01] og [OV99]. Dersom ikke annet er angitt, er presentasjon basert på [WV01]. For replikering og replikeringsalgoritmer er den viktigste kilden [GHOS96].

3.1 Distribuerte databasesystemer

I denne sammenhengen forutsetter vi at et distribuert databasesystem består av en samling frittstående noder som utelukkende kommuniserer over et nettverk. Vi ser dermed bort fra flerprosessorsystemer.

Vi forutsetter også at vi ønsker å oppnå *distribusjonstransparens*, dvs. at brukeren av databasesystemet ikke trenger å forholde seg til hvor dataene faktisk finnes. Vi antar at vi har mulighet til å kontrollere planleggeren ved alle noder, noe som ekskluderer såkalte multidatabasesystemer, hvor man forsøker å utføre distribuerte transaksjoner ved hjelp av standardgrensesnittet til de involvert databasesy-

[1] Konseptuelt er replikering bare en integritetsregel som uttrykker at en gitt mengde objekter alltid skal ha samme verdi.

stemene, og hvor nodene ikke engang nødvendigvis benytter samme databasesystem.

3.2 Globale eksekveringsplaner

Vi har tidligere uformelt omtalt transaksjoner hvor operasjonene utføres ved mer enn én node som *distribuerte* transaksjoner. Siden vi forutsetter distribusjonstransparens, krever vi ikke at operasjonene i en transaksjon selv spesifiserer hvilken node de utføres ved. Dette vil være vanligvis være opp til planleggeren idet transaksjonen utføres.

For å beskrive utførelsen av én eller flere transaksjoner i distribuerte databaser, benytter vi en *global plan*:

Anta vi har en mengde transaksjoner T som utføres i en database distribuert over en mengde noder N. O_T betegner unionen av alle operasjonene til transaksjonene i T. Vi har l være en funksjon som assosierer en operasjon med en node i N, og L_T betegner l utført på alle operasjonene i O_T. L_T består av elementer på formen *(operasjonstype, transaksjon, objekt, node)*, og en operasjon $r_i(x_a)$ i L_T skal da leses slik: «transaksjon t_i leser x ved n_a».

Vi definerer mengden av avslutningsoperasjoner, A_{TL}, slik:

For en transaksjon t_i i T inneholder A_{TL}, for hver node n_c hvor t_i utfører operasjoner, c_{ic} eller a_{ic}, dvs. henholdsvis *commit* og *abort* for t_i ved node n_c. Vi forutsetter her at en transaksjon bare utføres én gang ved hver node, og A_{TL} kan derfor, for hver transaksjon, bare inneholde én avslutningsoperasjon pr. node.[2]

En global plan g for T består da av en mengde $LA_T \subseteq (L_T \cup A_{TL})$ og en partiell ordning $<_g$ over LA_T. Alle par av konfliktoperasjoner i O_T må være ordnet i $<_g$, og dersom c_{ic} eller a_{ic} inngår i g, må alle operasjoner o_i, som t_i utfører ved n_c, ordnes slik at $o_i <_g \{c_{ic}, a_{ic}\}$.

En global plan g er komplett dersom den, for alle transaksjoner t_i som utfører operasjoner i g, oppfyller følgende:

- For alle noder n_c som utfører operasjoner i t_i, inngår enten c_{ic} eller a_{ic} i g.
- Det finnes ingen operasjoner i t_i som ikke utføres i g.

[2]Dette er ikke opplagt for globale planer. I praksis vil samme transaksjon kunne avbrytes og restartes flere ganger lokalt ved én node uten at transaksjonen avbrytes globalt. En komplett definisjon av globale planer bør håndtere dette, selv om de må være et krav at en transaksjon til slutt får samme sluttstatus ved alle noder. For å unngå å gjøre planbegrepet unødig komplisert, forutsetter vi imidlertid i denne sammenhengen at hver transaksjon bare utføres én gang ved hver node.

3.3 Global serialiserbarhet

Vi definerer global serialiserbarhet slik:

En global plan g er globalt serialiserbar hvis og bare hvis det, for et passende ekvivalenskrav, finnes en seriell, global plan som er ekvivalent med g.

Ut fra dette har vi følgende observasjon:

En global plan g er globalt serialiserbar hvis og bare hvis transaksjonene i g serialiseres i samme rekkefølge ved alle noder.

Projeksjonen av en global plan g på en node n_a, kalt $\Pi_g(n_a)$, er en plan som inneholder alle operasjoner i g som utføres ved n_a. For alle par av operasjoner o_k, o_l i $\Pi_g(n_a)$ er $o_k <_{\Pi_g(n_a)} o_l$ hvis $o_k <_g o_l$.

En global konfliktgraf er definert slik:

Konfliktgrafen for en global plan g representerer unionen av konfliktgrafene til projeksjonen av g på alle noder som utfører operasjoner i g. Alle transaksjoner som utfører operasjoner i g er representert med en grafnode i denne konfliktgrafen, og det går en kant fra en transaksjon t_i til en transaksjon t_j hvis og bare hvis det finnes en node n_c hvor konfliktgrafen for $\Pi_g(n_c)$ inneholder en kant fra t_i til t_j.

Eksempel 3.1

Anta vi har to transaksjoner $t_i = r_i(x)\ w_i(x)\ r_i(y)$ og $t_j = r_j(x)\ w_j(y)$. La t_i og t_j utføres i en distribuert database, hvor x lagres ved node n_a og y ved n_c. Da er g en mulig plan for utførelsen av t_i og t_j:

$g = r_i(x_a)\ r_j(x_a)\ w_i(x_a)\ r_i(y_b)\ c_{ia}\ c_{ib}\ w_j(y_b)\ c_{ja}\ c_{jb}$

Konfliktgrafene for $\Pi_g(n_a)$, $\Pi_g(n_b)$ og g er vist i figur 3.1.

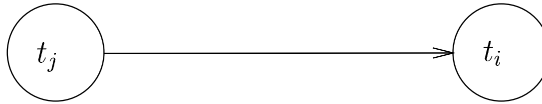
Vi ser at selv om projeksjonen av g er serialiserbar ved begge nodene, serialiseres t_i og t_j i ulik rekkefølge , og den globale konfliktgrafen blir syklisk. g er følgelig ikke globalt serialiserbar.

Vi får ofte bruk for å spesifisere om en transaksjon som utføres i en distribuert database er aktiv ved mer enn én node. Transaksjoner som utføres i en slik database er derfor enten *globale* eller *lokale*, og en transaksjon er global hvis og bare hvis den utfører operasjoner ved mer enn én node.

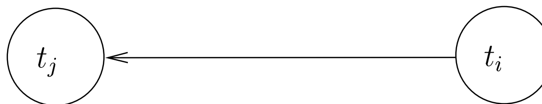
3.4 Algoritmer som sikrer global serialiserbarhet

I prinsippet kan alle strategier for samtidighetskontroll som garanterer serialiserbarhet i ikke-distribuerte databaser, også benyttes i distribuerte databaser. Her skisserer vi kort tre slike algoritmer, basert på to-fase-låsing, tidsstempler og multiversjons-snapshot.

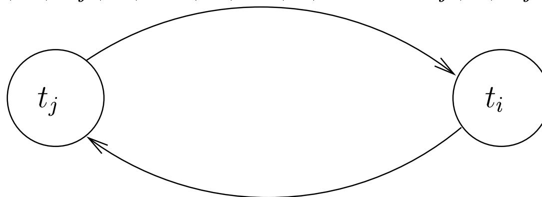
$\Pi_g(n_a) = r_i(x)\ r_j(x)\ w_i(x)\ c_i\ c_j$



$\Pi_g(n_b) = r_i(y)\ c_i\ w_j(y)\ c_j$



$g = r_i(x_a)\ r_j(x_a)\ w_i(x_a)\ r_i(y_b)\ c_{ia}\ c_{ib}\ w_j(y_b)\ c_{ja}\ c_{jb}$



Figur 3.1: Konfliktgrafer for g

Fordelene med distribuering av dataene er nevnt tidligere: gode skaleringsmuligheter og økt pålitelighet. Ulempen er at transaksjonshåndteringen blir mer kompleks, særlig fordi vi har flere, helt uavhengige prosesseringsenheter og variabel forsinkelse i nettverket. Noen av problemene dette medfører er:

- Forsinkelser som i et ikke-distribuert databasesystem kan neglisjeres, for eksempel tiden det tar å sette en lås, øker med flere størrelsesordener dersom det kreves meldingsutveksling over nettverk.
- Vi ønsker vanligvis ikke at hele systemet skal bli inoperativt selv om én node blir utilgjengelig. Men siden nodene bare kommuniserer via nettverk, har vi ingen pålitelig metode for å avgjøre årsaken til manglende respons. Dette er særlig viktig siden vi ved en node n_a ikke kan gå ut fra at en node n_b er utilgjengelig for alle, selv om n_b ikke svarer på n_a's forespørsler.
- Siden vi mangler en «allvitende» kontrollenhet, krever operasjoner som tildeling av globale tidsstempler, søk etter vranglåser og avslutning av transaksjoner spesiell håndtering.

Vi begrenser oss til problemer knyttet til samtidighetskontroll, og konsekvensene av vilkårlige nettverksfeil eller feil på enkeltnoder er i liten grad omtalt. Dersom ikke annet er angitt, vil en robust implementasjon ikke kreve konseptuelle endringer i algoritmen.

Vi forutsetter at vi for alle transaksjoner har én bestemt node som har ansvaret for utførelsen av denne transaksjonen, og vi kaller denne noden transaksjonens *utgangsnode*. Vanligvis antar vi at noden som starter en transaksjon t_i også er t_i's utgangsnode.

3.4.1 Global tofase-låsing

Dersom hver node implementerer tofase-låsing, og alle operasjoner kontrolleres lokalt av det enkelte databasesystemet, kan vi oppnå global serialiserbarhet ved å sikre at tofase-kravet også håndheves globalt. En transaksjon kan derfor ikke fjerne noen av sine låser før den har låst objektene den trenger ved *alle* noder hvor den utføres, og vi kaller dette t_i's globale låsepunkt. Global tofase-låsing kan implementeres slik:

Enhver lokal node n_c, som utfører operasjoner for en global transaksjon t_i, informerer t_i's utgangsnode idet t_i har låst alle objekter den trenger ved n_c (dvs. at t_i er nådd *låsepunktet* ved n_c). t_i kan først låse opp objekter når dette punktet er nådd ved alle noder hvor t_i utføres.

Argumentet for at global tofase-låsing garanterer global serialiserbarhet er det samme som i ikke-distribuerte databaser:

K_{ij} betegner konfliktmengden for to globale transaksjoner t_i og t_j. t_i må enten ha aksessert, og derfor låst opp, alle objektene i K_{ij} *før* t_j's globale låsepunkt, eller t_i må aksessere alle objektene i K_{ij} *etter* t_j. Om alle objektene i K_{ij} lagres ved samme node er i denne sammenhengen irrelevant.

Vranglåser i distribuerte databaser

I seksjon 2.5.2 viste vi at algoritmer basert på tofase-låsing er sårbare for vranglåser, og skisserte mulige strategier for håndtering av slike.

En utfordring i distribuerte databaser er *distribuerte* vranglåser: dersom transaksjon t_i venter på t_j ved node n_c, mens t_j venter på t_i ved node n_d, er t_i og t_j i en distribuert vranglås.

Å vedlikeholde en sentral vente-på-graf ved én av nodene er mulig, men kan gi en uakseptabel flaskehals.

En alternativ løsning er at en transaksjon t_i, idet den settes til å vente, traverserer vente-på-grafen ved å sende ut en melding til transaksjonen den venter på, kalt t_j. Dersom t_j også venter, sender denne meldingen videre på samme måte. En vranglås vil medføre at t_i til slutt mottar meldingen den opprinnelig sendte ut, og vranglåsen kan f.eks. brytes ved at t_i avbryter seg selv.

Det finnes flere metoder for å unngå vranglåser i distribuerte databaser, inkludert wound-wait og wait-die-strategiene vi skisserte i seksjon 2.5.2. En løsning med nedtellingsur, som startes i det transaksjonene settes til å vente, kan selvsagt implementeres helt uavhengig av om databasen er distribuert eller ikke.

3.4.2 Global tidsstempelordning

Utgangspunktet for tidsstempelordning er at for alle par av operasjoner o_i, o_j i konflikt på et objekt x, skal o_i utføres før o_j hvis og bare hvis $ts(t_i) < ts(t_j)$.

Forutsatt at alle transaksjoner får et globalt unikt tidsstempel ved oppstart, garanterer denne algoritmen global serialiserbarhet. Alle transaksjoner i konflikt vil da ordnes i tidsstempelrekkefølge ved alle noder hvor de utføres.

Å tildele slike tidsstempler krever omtanke i en distribuert database, siden vi må unngå at to transaksjoner som starter ved ulike noder får samme tidsstempel. Vi skisserer her to mulige løsninger:

- *Sentralisert utdeling av tidsstempler*

  Én node har ansvaret for all tildeling av tidsstempler. Ulempen er at tildeling av tidsstempler alltid krever utveksling av meldinger, og at hele systemet blir avhengig av denne noden.

- *Distribuert utdeling av tidsstempler*

  Alle noder tildeles en unik id, og de har sin egen, lokale teller. Vi lar $id(n_c)$ betegne id-en til en node n_c, og et globalt unik tidsstempel for en transaksjon t_i som startes ved n_c, kalt $ts(t_i)$, består da av et par $(ts_c(t_i), id(n_c))$, hvor $ts_c(t_i)$ betegner tidsstempelet som tildeles t_i ved n_c.

  Vi forutsetter at nodenes id-er kan sorteres. La t_k og t_l være to transaksjoner, hvor t_k og t_l startes ved henholdsvis n_a og n_b. Da er $ts(t_k) < ts(t_l)$ hvis $ts_a(t_k) < ts_b(t_l)$, eller hvis $ts_a(t_k) = ts_b(t_l)$ og $id(n_a) < id(n_b)$.

3.4.3 Tidsstempelordning med globale snapshot

I seksjon 2.7.2 introduserte vi en algoritme som lar lesetransaksjoner lese fra et *snapshot*, mens alle skrivetransaksjoner koordineres ved hjelp av en samtidighetskontrollalgoritme som ikke tar hensyn til eksistensen av flere versjoner.

En slik framgangsmåte kan, i henhold til [BHG87], også fungere i distribuerte databaser. [BHG87]'s gjennomgang av tidsstempelordning med snapshot er imidlertid svært lite formell, og vi presenterer her tidsstempelordning med globale snapshot ved hjelp av globale multiversjonsplaner.

Globale multiversjonsplaner

La T være en mengde transaksjoner som utføres i en distribuert multiversjonsdatabase. Vi forutsetter at alle oppdateringstransaksjoner har en globalt unik id, og at alle versjoner kan identifiseres ved hjelp av id-en til transaksjonen som skrev den. O_T representerer unionen av operasjonene i T, og vi definerer en mengde LH_T, hvor hver operasjon i O_T er assosiert med en versjon og en node. Alle elementer i LH_T er på formen *(operasjonstype, transaksjon, objekt, versjon, node)*, og vi lar $r_j(x_{ia})$ betegne en operasjon hvor t_j leser t_i's versjon av x ved node n_a. Vi krever at for alle operasjoner på formen $r_l(x_{ka})$, hvor t_l altså leser en versjon x_k ved en node n_c, må t_k ha skrevet x ***før*** $r_l(x_{ka})$.

Vi lar som før A_{LT} betegne avslutningsoperasjonene for operasjonene i T, og en global multiversjonsplan mg for T består da av en mengde $MA_T \subseteq (LH_T \cup A_{LT})$ og en partiell ordning $<_{mg}$ over LH_T. Alle par av konfliktoperasjoner i O_T må være ordnet i $<_{mg}$, og dersom c_{ic} eller a_{ic} inngår i mg, må alle operasjoner o_i som t_i utfører ved n_c ordnes slik at $o_i <_{mg} \{c_{ic}, a_{ic}\}$.

Multiversjons-serialiseringsgrafen for en global multiversjonsplan mg representerer unionen av multiversjons-serialiseringsgrafene for projeksjonen av mg på alle noder som utfører operasjoner i mg. Projeksjonen av en global multiversjonsplan på en node er definert nøyaktig som projeksjonen av ordinære globale planer: en operasjon inngår i projeksjonen av mg på en node hvis og bare hvis den utføres ved den aktuelle noden.

En global multiversjonsplan mg er globalt multiversjonsserialiserbar hvis og bare hvis den globale multiversjons-serialiseringsgrafen for mg er asyklisk.

Globale snapshot

I seksjon 2.7.2 definerte vi et snapshot for lesetransaksjoner slik:

En operasjon $r_i(x)$ skal assosieres med den versjonen av x som har det høyeste tidsstempelet blant de versjonene med tidsstempel mindre enn eller lik $ts(t_i)$.

En versjon tidsstemples med commit-tidspunktet til transaksjonen som skrev den. For en global lesetransaksjon må vi da sikre at alle noder den leser fra definerer samme snapshot:

La t_k være en transaksjon som skriver to objekter x og y. x skrives ved node n_a og y ved node n_b. For en lesetransaksjon t_i som leser x_{ka}, krever vi at dersom t_i også leser y ved node n_b, skal den også lese y_{kb}. Vi må derfor unngå følgende eksekvering:

Eksempel 3.2

La $t_i = r_i(x)\ w_i(x)\ w_i(y)$ og $t_j = r_j(x)\ r_j(y)$ være to transaksjoner. x lagres ved en node n_a, og y ved node n_b. Da er følgende en mulig plan for utførelsen av t_i og t_j i en distribuert multiversjonsdatabase:

$mg = r_i(x_0)\ w_i(x_{ia})\ w_i(y_{ib})\ c_{ia}\ r_j(x_{ia})\ r_j(y_{0b})\ c_{ib}\ c_{ja}\ c_{jb}$

Siden t_i's commit ikke er registrert ved n_b idet t_j leser y, leser t_j i stedet fra t_0, og eksekveringen er ikke serialiserbar.

Den enkleste løsningen for å unngå slik eksekvering er å tidsstemple en lesetransaksjon så lavt at vi kan garantere at alle transaksjoner den leser fra er committet ved alle noder. I dette tilfellet ville vi da måtte sikre at $ts(t_j) < ts(t_k)$. Vi kommer nærmere inn på dette i kapittel 4, hvor vi viser en snapshotalgoritme hvor skrivetransaksjonene synkroniseres ved hjelp av en replikeringsgraf.

3.4.4 Tofase-commit

Som nevnt, vil avslutningsoperasjonene trenge spesiell håndtering i et distribuert system. Vi skisserer her en enkel algoritme for å committe globale transaksjoner[OV99]:

Korrekt utførelse av globale transaksjoner krever at dersom én node committer den aktuelle transaksjonen, committes den ved alle. Tofase-commit er basert på en avstemning blant nodene om transaksjonens sluttstatus, hvor *abort*-stemmer gir veto.

Commit av en global transaksjon t_i utføres slik:

1. t_i's utgangsnode sender en commitforespørsel til alle noder som utfører operasjoner i t_i.
2. En node som mottar en slik forespørsel, svarer først når den vet om t_i kan committes lokalt. En global commit krever derfor at alle operasjonene i transaksjonen er utført.
3. Dersom utgangsnoden får positivt svar fra alle nodene, committes t_i. Hvis én node svarer negativt, avbrytes transaksjonen overalt.
4. Utgangsnoden sender ut melding om resultatet, og transaksjonen avsluttes.

Feil på nettverk eller enkeltnoder kan oppstå når som helst, og i praksis vil man derfor kreve at dersom utgangsnoden ikke får svar på commitforespørselen innen en fastsatt tid, må transaksjonen avbrytes. Imidlertid må vi kreve at en node som har stemt *commit*, ikke avslutter transaksjonen før den er kjent med resultatet.

3.5 Replikerte databaser

En konsekvens av vårt mål om distribusjonstransparens er at vi tilbyr *replikeringstransparens*, dvs. at brukeren ikke trenger å forholde seg til om et objekt er replikert eller ikke. En leseoperasjon på et replikert objekt tilordnes en vilkårlig kopi, mens en skriveoperasjon vil oppdatere alle kopier.

3.5.1 Replikeringsalgoritmer

Transparent replikering krever at databasesystemet implementerer en mekanisme for å sikre korrekt oppdatering av kopiene. Vi kaller dette en *replikeringsalgoritme*.

Definisjonen av serialiserbarhet er, som nevnt tidligere, uendret, siden kopiene av et replikert objekt kan sees som en samling forskjellige objekter med en bestemt integritetsregel, nemlig at alle disse objektene skal ha samme verdi.[3]

Siden pålitelighet vanligvis er et viktig mål for replikeringen, ønsker vi også at transaksjoner skal kunne utføres selv om én eller flere noder blir utilgjengelige. Et vanlig krav er da at replikeringsalgoritmen bare trenger å oppdatere *tilgjengelige* noder. Nettverkspartisjoner kan imidlertid skape problemer:

[3] Både [BHG87], [OV99] og [WV01] benytter *enkopi-serialiserbarhet* som korrekthetskrav for transaksjoner i replikerte databaser, og dette kravet uttrykker at eksekveringen skal være ekvivalent med en seriell utførelse av de samme transaksjonene i en ikke-replikert database. Men jeg kan ikke se at det finnes noen planer som er enkopi-serialiserbare uten å være globalt serialiserbare, og heller ingen planer som er globalt serialiserbare uten å samtidig være enkopi-serialiserbare.

Eksempel 3.3

Anta vi har et replikert objekt x som lagres ved tre noder n_a, n_b og n_c.

La $t_i = w_i(x)$ og $t_j = w_j(x)$ være to transaksjoner. t_i har n_a som utgangsnode, mens t_j har n_c som utgangsnode.

Vi har en *nettverkspartisjon* dersom noen av nodene er ute av stand til å kommunisere med hverandre, men likevel er operative. Vi kan heller ikke utelukke at slike noder, selv om de ikke kan kommunisere innbyrdes, kan kommunisere med brukere av databasen.

Hvis vi har en nettverkspartisjon hvor n_a og n_b kan kommunisere seg i mellom, men ikke med n_c, er følgende en mulig plan for utførelsen av t_i og t_j:

$g = w_i(x_a)\ w_i(x_b)\ c_{ia}\ c_{ib}\ w_j(x_c)\ c_{jc}$

Databasen er da inkonsistent, siden vi krever at verdien av x må være den samme ved alle noder.

En vanlig løsning på dette er å forlange *quorum*, dvs. at operasjoner bare kan utføres dersom de er i den største partisjonen. Hvis vi forutsetter at alle noder til enhver tid kjenner det totale antallet noder som lagrer kopier av et objekt x, samt hvilke av disse som er tilgjengelige, er dette enkelt. I praksis vil vi sjelden kunne vite nøyaktig hvilke noder som til enhver tid er tilgjengelige, men dette kan løses ved hjelp av avstemninger, enten i forkant av operasjonen eller før transaksjonen committes.[4]

3.5.2 Skaleringsevne

Nettverkspartisjoner er et problem som skyldes behovet for økt pålitelighet. Et annet formål med replikering er gjerne økt skalerbarhet, men ulempen er at å tilby effektiv samtidighetskontroll i replikerte databaser er vanskelig.

Hele denne seksjonen er basert på [GHOS96], som klassifiserer replikeringsalgoritmer ut fra to kriterier:

- *Spredning av oppdateringer*

  Iherdig replikering krever at alle kopier må være oppdatert før en skrivetransaksjon kan committes. *Doven* replikering tillater at skrivetransaksjoner committes så lenge én node er oppdatert.

- *Objekteierskap*

[4] Flere quorumstrategier er presentert i [OV99], kapittel 12.

> I replikeringsalgoritmer som benytter *primæreierskap* tilordnes alle replikerte objekter en *primærnode*, og en skrivetransaksjon kan bare oppdatere den kopien av et replikert objekt som lagres ved primærnoden. Vi kaller denne kopien en *primærkopi*, mens alle andre kopier er *sekundærkopier*.
>
> Ved *gruppeeierskap* kan transaksjonene oppdatere kopier av et replikert objekt ved alle noder.

Hvis vi forutsetter iherdig replikering, vil en global skrivetransaksjon kunne utføres slik:

Eksempel 3.4

La $t_i = w_i(x)\ w_i(y)\ w_i(z)$ være en transaksjon som utføres i en distribuert database, hvor x, y og z er replikert ved tre noder n_a, n_b og n_c.

En mulig plan for utførelsen av t_i er da:

$g = w_i(x_a)\ w_i(y_a)\ w_i(z_a)\ w_i(x_b)\ w_i(y_b)\ w_i(z_b)\ w_i(x_c)\ w_i(y_c)\ w_i(z_c)\ c_{ia}\ c_{ib}\ c_{ic}$

Den største utfordringen ved replikering er, som eksempelet viser, at antallet skriveoperasjoner, og dermed lengden på transaksjonene, øker dramatisk. Selv om vi forutsetter at oppdatering av ulike noder parallelliseres, kan ikke en skrivetransaksjon committes før alle kopier er oppdatert. I tillegg vil spredning av oppdateringene over nettverk medføre en viss forsinkelse.

[GHOS96] forutsetter at vi benytter en algoritme basert på *les-committet* (se seksjon 2.7.2), hvor alle leseoperasjoner leser fra et snapshot, mens skriveoperasjonene kontrolleres med strikt tofase-låsing.

Dermed kan vi utelukke lese-skrive- og skrive-lese-konflikter, og vi antar i denne sammenhengen at alle operasjoner er skriveoperasjoner, og alle transaksjoner er derfor skrivetransaksjoner.

Vi forutsetter at det hvert sekund starter P transaksjoner ved hver node. Siden målet i denne sammenhengen er skalerbarhet, forutsetter vi at et større antall noder innebærer høyere transaksjonsbelastning, og i en database replikert over tre noder antar vi derfor at det startes $3 * P$ transaksjoner pr. sekund. Videre antar vi at databasen alltid er fullreplikert, dvs. at alle objekter er replikert ved alle noder.

Hvis hver transaksjon inneholder K operasjoner, vil vi i en ikke-replikert database utføre $P * K$ skriveoperasjoner pr. sekund. Dersom vi replikerer databasen over to noder, skal det utføres $2 * P * K$ skriveoperasjoner i sekundet. Men siden dataene er replikert, må alle skriveoperasjoner utføres ved begge noder. Hver node må derfor utføre $2 * P * K$ operasjoner pr. sekund, og totalt blir belastningen $4 * P * K$ operasjoner i sekundet.

Generelt har vi at ved en økning i antallet noder med en faktor på N vil antallet skriveoperasjoner øke med N^2, og replikerte databaser skalerer ikke lineært.

3.5.3 Vranglåser

Hvis vi setter sannsynligheten for at en transaksjon må vente på en annen lik D, hvor $D < 1$, vil sannsynligheten for at to transaksjoner venter på hverandre være proporsjonal med D^2.

Siden databasesystemer vanligvis lagrer et stort antall objekter, vil sannsynligheten for at to samtidige transaksjoner må vente på hverandre være relativt lav, dvs. at $D \ll 1$, og slike vranglåser vil ikke representere noen alvorlig belastning. Sannsynligheten for at vi får vranglåser som involverer flere enn to transaksjoner er enda lavere, siden sannsynligheten for en vranglås som involverer i transaksjoner er proporsjonal med D^i.

Dette endrer seg imidlertid dersom databasen er replikert. Stigningen i antallet skriveoperasjoner, sammen med lengre transaksjoner, gir en kraftig økning i sannsynligheten for at skrivetransaksjoner må vente på hverandre.

Iherdig replikering

[GHOS96] forsøker å estimere forsinkelsen ved meldingsutveksling ved å anta at lengden på transaksjonene i en iherdig replikert database øker proporsjonalt med antall noder. At transaksjonene ved N ganger så mange noder blir N ganger så lange, gjør at sannsynligheten for at en gitt transaksjon blir involvert i en vranglås stiger proporsjonalt med N^2. Siden antallet transaksjoner også øker med N, vil den totale frekvensen av vranglåser, dvs. hvor hyppig en eller annen vranglås oppstår i databasen, stige med $N^2 * N = N^3$.

Dette er en betydelig økning. Siden det i tillegg kan være kostbart å oppdage vranglåser i distribuerte systemer, begrenser dette skaleringsevnen til replikerte databaser, og [GHOS96] går langt i antyde at databaser som forsøker å kombinere serialiserbarhet og iherdig replikering ikke skalerer: «*simple replication (transactional update-anywhere-anytime-anyway) cannot be made to work with global serializability*».

Merk at dette problemet er uavhengig av om objektene bare oppdateres ved en primærnode, siden økningen i antall samtidige skriveoperasjoner vil være det samme.

Doven replikering

Hvis vi i stedet benytter doven replikering, vil vi i stedet for én lang transaksjon dele oppdateringene opp i flere, kortere oppdateringstransaksjoner, én ved hver node. Det totale antallet operasjoner som må utføres er det samme, men transaksjonene kan nå committes uten at alle kopiene er oppdatert. Dette reduserer lengden på transaksjonene, og vi får følgelig færre transaksjoner som utføres samtidig.

Doven replikering og gruppeeierskap er i utgangspunktet uforenlig med globalt serialiserbar eksekvering, og [GHOS96] forutsetter at konflikter i så fall håndteres i etterkant, f.eks. med kompenserende transaksjoner.

Vi kan oppnå serialiserbarhet sammen med doven replikering ved å begrense oppdatering til primærkopier og samtidig kreve at alle leseoperasjoner setter en leselås ved objektets primærnode. Vi forutsetter strikt tofase-låsing ved primærnoden, og kan derfor benytte *Thomas' Write Rule*[WV01, BHG87] for å synkronisere skrive-skrive-konflikter på sekundærkopier[GHOS96]:

Alle transaksjoner som skriver samme objekt tidsstemples i serialiseringsrekkefølge, og et objekt lagrer til enhver tid tidsstempelet til den transaksjonen som sist skrev objektet. Thomas' Write Rule sier da at dersom en transaksjon t_i ber om å få oppdatere et objekt x, men t_i's tidsstempel er *lavere* enn tidsstempelet til x, kan skriveoperasjonen ignoreres, mens t_i fortsetter eksekveringen. Dersom t_i's tidsstempel derimot er høyere enn x', utføres operasjonen, og t_i fortsetter som vanlig.

Kravet om tidsstempling i serialiseringsrekkefølge gjør at vi tidsstempler transaksjonene idet de committes ved primærnoden,[5] og sekundærkopiene kan derfor ikke oppdateres før transaksjonen er committet der. Til gjengjeld kan vi se bort fra skrive-skrive-konflikter på sekundærkopier, og vi ser derfor bare på frekvensen av vranglåser ved primærnoder:

Siden transaksjonene ikke trenger å vente på oppdatering av sekundærkopier, er disse like lange som transaksjoner i ikke-replikerte databaser. Vi kan også se bort fra transaksjoner som oppdaterer sekundærkopier, siden disse aldri må vente (vi ser som nevnt bort fra leseoperasjoner, siden disse antas å lese fra et snapshot).

Vi antar fremdeles at en økning av antallet noder med en faktor N gir N flere transaksjoner, og alle disse transaksjonene vil da utføres ved objektenes primærnoder. Siden lengden ikke øker, vil vranglåsfrekvensen imidlertid bare øke proporsjonalt med N^2, noe som gir bedre skaleringsmuligheter enn for iherdig replikering. Kostnadene ved algoritmen er imidlertid betydelige, særlig siden alle leseoperasjoner må be om leselås ved utgangsnoden.

Konklusjonen til [GHOS96]

Ingen av algoritmene [GHOS96] presenterer kombinerer global serialiserbarhet og god skaleringsevne, og forfatterne konkluderer med at man i praksis må velge mellom skalerbarhet og serialiserbarhet. Skaleringsevnen kan imidlertid bedres ved å utnytte semantisk kjennskap til applikasjonens integritetsregler. Et eksempel på en slik database er systemet for håndtering av domenenavn på Internett, DNS (*Domain Name System*):

[5] Ved å benytte strikt tofase-låsing sikrer vi at alle par av transaksjoner som skriver et felles objekt committes i serialiseringsrekkefølge.

Eksempel 3.5

DNS er en dovent replikert database med primæreierskap, og alle objekter merkes med et varighets-tidsstempel. Dersom en node n_c trenger et objekt x, og n_c ikke er x' primærnode, hentes verdien av x fra primærnoden hvis og bare hvis et av følgende er oppfylt:

- n_c har ikke lest x tidligere
- n_c har lest x, men varighets-tidsstempelet viser at denne verdien nå er foreldet.

3.5.4 Andre replikeringsstrategier

Det finne flere andre metoder for å oppnå global serialiserbarhet i replikerte databaser. I [AAS97] og [KA00] garanteres global serialiserbarhet ved hjelp av varianter av tidsstempelordning. Algoritmene i [CRR96] og [BKR$^+$99] garanterer global serialiserbarhet sammen med doven oppdatering og primæreierskap, men unngår låser ved å kreve at dataene replikeres i henhold til en rettet, asyklisk graf. I denne grafen går det en kant fra en node n_a til en node n_b hvis og bare hvis n_b lagrer en kopi av et objekt som har n_a som primærnode. Objekter kan bare oppdateres ved primærnoden, og en transaksjon kan bare lese objekter som er tilgjengelige lokalt.

Mest interessant for oss er imidlertid algoritmer basert på en replikeringsgraf[BK97]. I det neste kapittelet defineres slike grafer, og vi presenterer tre algoritmer som garanterer global serialiserbarhet ved hjelp av en replikeringsgraf.

Kapittel 4

Replikeringsgrafer

[BK97] introduserer *replikeringsgrafer* som en metode for å bedre ytelsen på transaksjoner utført i replikerte databaser.

Ideen er at dem globale serialiseringsgrafen for en mengde transaksjoner kan representeres ved hjelp av en replikeringsgraf, og at alle ikke-serialiserbare globale eksekveringsplaner vil resultere i en sykel i denne grafen. Vi forutsetter at alle de lokale databasesystemene garanterer lokal serialiserbarhet. Replikeringsgrafen vil vanligvis være flere størrelsesordener mindre enn serialiseringsgrafen, og det virker derfor praktisk gjennomførbart å basere en replikeringsalgoritme på en slik graf. Både [BK97] og [ABKW98] presenterer algoritmer som tilbyr global serialiserbarhet sammen med *doven* replikering, og [ABKW98] presenterer testresultater som indikerer at en slik algoritme basert på en replikeringsgraf tilbyr konkurransedyktig ytelse.

4.1 Definisjon av replikeringsgrafer

En replikeringsgraf for en global plan g er en urettet, bipartitt graf, dvs. en graf hvor nodene kan deles i to distinkte mengder T og V. Alle globale transaksjoner tilordnes en egen node i T, mens V representerer g's *virtuelle noder*:

En virtuell node vn_{ia}, for en transaksjon t_i, inneholder alle objektene t_i har *aksessert* ved en fysisk node n_a, og det går en kant mellom vn_{ia} og t_i i replikeringsgrafen. La transaksjonene t_i og t_j utføre hver sin operasjon ved n_a. Dersom dette medfører en kant mellom t_i og t_j i konfliktgrafen for g, skal t_i og t_j dele virtuell node ved n_a. Notasjonen $vn_{ia} = vn_{ja}$ brukes for å illustrere at to transaksjoner t_i og t_j må være tilknyttet samme virtuelle node ved fysisk node n_a. Dette betyr at vn_{ia} ofte vil inneholde objekter som ikke er aksessert av t_i.

Aksessere defineres her slik:

Anta vi har en database replikert over en mengde noder N. En transaksjon t_i aksesserer et objekt x ved en node n_a dersom:

- t_i leser x ved n_a, **eller**
- n_a replikerer x, og t_i skriver en kopi av x ved *én* av nodene i N.[1]

Med andre ord må t_i, idet den skriver et objekt x, tilordnes en virtuell node ved alle noder i N som replikerer x. Det kan selvsagt skje at t_i allerede er tilknyttet en virtuell node ved én eller flere av disse nodene, og i så fall legges bare x inn i de allerede eksisterende virtuelle nodene.

Replikeringsgrafen for en gitt plan er ikke unik. En replikeringsgraf hvor hver fysisk node inneholder én virtuell node, som da består av alle objektene, vil være korrekt, men ikke særlig effektiv. Generelt, dvs. når vi snakker om replikeringsgrafer utenfor konteksten av en gitt algoritme, har vi derfor flere mulige replikeringsgrafer for en gitt plan. Dersom vi i forutsetter en algoritme som sikrer at de virtuelle nodene er så små som mulig, er replikeringsgrafen for én bestemt plan likevel entydig.

4.2 Replikeringsgrafer og global serialiserbarhet

En global plan g vil være globalt serialiserbar dersom replikeringsgrafen for g er asyklisk under hele utførelsen av g. [BK97] begrunner denne påstanden slik:

Anta at vi har en global plan g definert for en mengde transaksjoner T, hvor T inneholder både globale og lokale transaksjoner.

Dersom g ikke er globalt serialiserbar, må den globale konfliktgrafen SG_g inneholde en sykel. La s være en slik sykel. Vi forutsetter som nevnt at alle lokale noder garanterer lokal serialiserbarhet for projeksjonen av g på seg selv, og s må derfor involvere to eller flere globale transaksjoner (ellers ville sykelen være lokal ved en fysisk node).

La t_i og t_j være globale transaksjoner slik at $i < j$, og anta vi har en vei $t_i \rightarrow t_{i+1} \rightarrow \ldots \rightarrow t_{j-1} \rightarrow t_j$ i s, hvor $t_{i+1}, \ldots, t_{j-1}$ er lokale transaksjoner.

Siden det ikke går kanter i konfliktgrafen mellom lokale transaksjoner på tvers av fysiske noder, må $t_{i+1}, \ldots, t_{j-1}$ utføres ved samme fysiske node n_a. Siden t_{i+1} er lokal ved n_a, må det gå en kant fra t_i til t_{i+1} i den lokale projeksjonen av SG_g på n_a. Tilsvarende må det gå en kant fra t_{j-1} til t_j i den lokale projeksjonen av SG_g på n_a. Dermed vet vi (ut fra definisjonen av replikeringsgrafer) at $vn_{ia} = vn_{(i+1)a} = \ldots =$

[1] Det er nødvendig å regne en lokal skriveoperasjon som en global aksess for å kunne tilby doven replikering.

$vn_{(j-1)a} = vn_{ja}$. Vi kaller denne felles virtuelle noden vn_a, og det finnes en kant i replikeringsgrafen mellom vn_a og t_i, og mellom vn_a og t_j. Vi ser altså at dersom det finnes en vei fra en global transaksjon t_i til en annen global transaksjon t_j i SG_g, finnes det en vei fra t_i til t_j i replikeringsgrafen. Videre vet vi at dersom t_i og t_j er involvert i en sykel i SG_g, må det også finnes en vei fra t_j til t_i, og denne må involvere minst en annen fysisk node (siden de lokale nodene garanterer at projeksjonen av SG_g på seg selv er asyklisk). Dermed må det også finnes en annen vei mellom node t_j og t_i i replikeringsgrafen.

Vi kan da konkludere med at dersom det for en global plan g finnes en asyklisk replikeringsgraf, er g globalt serialiserbar.

Eksempel 4.1

La t_i, t_j og t_k være tre transaksjoner:

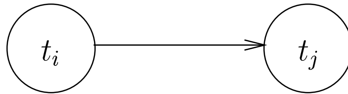
$t_i = r_i(x)\ r_i(y)\ w_i(x)$
$t_j = r_j(y)\ w_j(y)$
$t_k = r_k(x)\ r_k(y)$

Transaksjonene t_i, t_j og t_k utføres i en database replikert over to noder, n_a og n_b, hvor n_a er primærnode for både x og y. En mulig plan for t_i, t_j og t_k er:

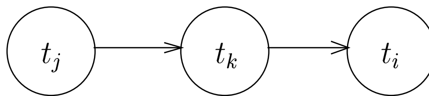
$g = r_i(x_a)\ r_i(y_a)\ w_i(x_a)\ c_{ia}\ r_k(x_b)\ w_i(x_b)\ c_{ib}\ r_j(y_a)\ w_j(y_a)\ c_{ja}\ w_j(y_b)\ c_{jb}\ r_k(y_b)\ c_{kb}$

At g ikke er globalt serialiserbar ser vi av konfliktgrafene:

$$\Pi_g(n_a) = r_i(x)\ r_i(y)\ w_i(x)\ c_i\ r_j(y)\ w_j(y)\ c_j$$



$$\Pi_g(n_b) = r_k(x)\ w_i(x)\ c_i\ w_j(y)\ c_j\ r_k(y)\ c_k$$

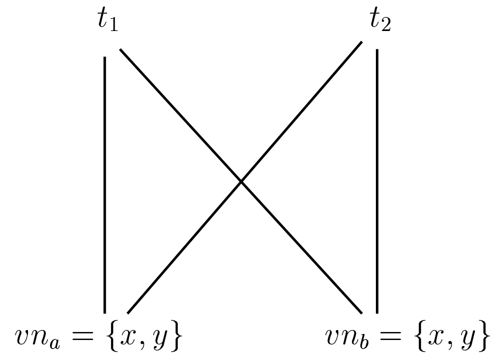


Siden transaksjonene serialiseres i ulik rekkefølge ved de to nodene, inneholder den globale konfliktgrafen en sykel, og g er ikke globalt serialiserbar.

Figur 4.1 illustrerer hvordan replikeringsgrafen for g, RG_g, vil se ut etter at t_k har utført $r_k(y_b)$. Vi ser at t_i og t_j deler virtuell node både ved n_a og n_b, og replikeringsgrafen for g er derfor syklisk.

Det er særlig to ting som er verdt å legge merke til:

- Lokale transaksjoner påvirker replikeringsgrafen selv om de ikke har egne noder. Transaksjonene t_i og t_j ville ikke trenge å dele virtuell node ved n_b

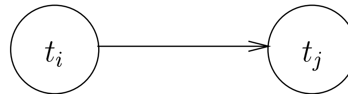


Figur 4.1: Replikeringsgraf for g

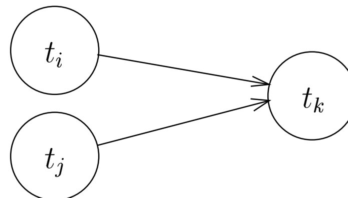
hvis ikke begge hadde vært i konflikt med t_k ved n_b, og i så fall ville heller ikke RG_g vært syklisk. De to konfliktene $r_k(x_b), w_i(x_b)$ og $w_j(y_b), r_k(y_b)$ medfører imidlertid at $vn_{ib} = vn_{jb} = vn_{kb}$.

- Eksempelet kan lett modifiseres til å vise at ikke alle serialiserbare planer gir en asyklisk replikeringsgraf. Dersom operasjonene $r_k(x_b)$ og $w_i(x_b)$ utføres i motsatt rekkefølge, dvs. at $w_i(x_b)$ utføres før $r_k(x_b)$, blir serialiseringsgrafene slik:

$$\Pi_g(n_a) = r_i(x)\ r_i(y)\ w_i(x)\ c_i\ r_j(y)\ w_j(y)\ c_j$$



$$\Pi_g(n_b) = w_i(x_b)\ c_i\ r_k(x)\ w_j(y)\ c_j\ r_k(y)\ c_k$$



Siden vi ikke lenger har noen vei fra t_j til t_i ved n_b, er planen globalt serialiserbar. I og med at t_i, t_j og t_k likevel må dele virtuell node ved n_b, forblir likevel replikeringsgrafen syklisk.

4.3 Algoritmer basert på en replikeringsgraf

[BK97] beskriver en algoritme som tilbyr globalt konfliktserialiserbar eksekvering ved hjelp av en replikeringsgraf. I [ABKW98] presenterer forfatterne en forbedret utgave av algoritmen, sammen med en simuleringsstudie som sammenligner denne med en låsbasert replikeringsalgoritme.

[ABKW98] gir en svært kort presentasjon av selve algoritmen, men denne drøftes grundigere i [BK99]. Mens [BK97] viser at en replikeringsgraf kan brukes til å garantere global serialiserbarhet, er det bare [BK99] som faktisk beviser algoritmens korrekthet, og dette beviset er knyttet sterkt til varianten av algoritmen som presenteres i [ABKW98]. Jeg har derfor laget et nytt bevis som beviser korrektheten til algoritmen i [BK97], dette brukes også som basis for å vise korrektheten til algoritmen som omtales i [ABKW98] og [BK99].

Her presenterer vi først [BK97]. Deretter beskrives endringene som gjøres i [ABKW98] sammen med en kort presentasjon av ytelsesmålingene. Til slutt skisserer vi en framgangsmåte for å la rene lesetransaksjoner lese fra et globalt snapshot, mens alle skrivetransaksjoner koordineres ved hjelp av en replikeringsgraf.

4.4 Replikeringsalgoritme i [BK97]

[BK97] forutsetter at transaksjonene utføres i en database hvor det finnes helt eller delvis replikerte objekter.

Transaksjonene er begrenset til å lese objekter som finnes ved utgangsnoden. I tillegg er alle objekter tilordnet en primærnode, og algoritmen krever at en transaksjon bare kan oppdatere objekter som har transaksjonens utgangsnode som primærnode. Dette er alvorlige begrensninger, men de er introdusert for at algoritmen skal oppnå best mulig ytelse og robusthet mot vranglåser. Siden de ikke er nødvendige for å oppnå korrekt eksekvering, ville det være interessant å se hvor stor betydning det ville ha om de sløyfes, og vi kommer tilbake til dette i seksjon 4.6.

En transaksjon t_i er til enhver tid i én av følgende tilstander:

- *AKTIV*

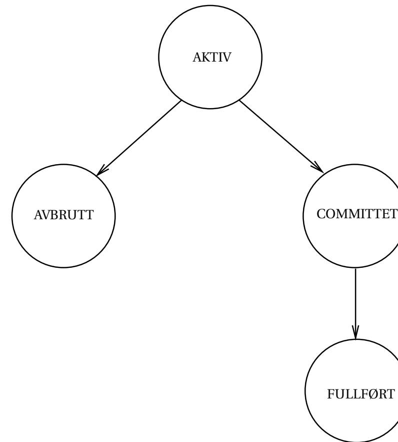
  Dersom t_i er aktiv ved sin utgangsnode.

- *COMMITTET*

  Dersom t_i er committet ved sin utgangsnode, men fremdeles ikke oppfyller kravene til FULLFØRT-tilstanden (definert på neste side).

- *AVBRUTT*

  Dersom t_i er avbrutt ved sin utgangsnode.



Figur 4.2: Lovlige tilstandsendringer for transaksjoner i replikeringsgrafalgoritmen

- *FULLFØRT*

  Dersom t_i er committet ved alle noder hvor den utføres, og det ikke finnes noen node n_a hvor projeksjonen av den globale konfliktgrafen på n_a inneholder en kant fra en transaksjon t_j (som ennå ikke er FULLFØRT) til t_i.

De tillatte tilstandsendringene er illustrert i figur 4.2.

Algoritmen vedlikeholder en replikeringsgraf. Denne kan lagres distribuert eller sentraliseres ved én node, hvilken variant man velger er et implementasjonsspørsmål.

[BK97] definerer følgende tre krav som må innfris for å vedlikeholde en sunn replikeringsgraf:

- *Lokalitetskravet*

  En transaksjon tilordnes nøyaktig én virtuell node ved hver fysisk node hvor den utfører operasjoner. Lokale transaksjoner utføres ved nøyaktig én fysisk node, og vil følgelig bare tilordnes én virtuell node.

- *Unionskravet*

  Til enhver tid må en virtuell node vn_{ia} inneholde mengden av dataobjekter t_i har aksessert ved fysisk node n_a så langt. Dersom to transaksjoner t_i og t_j aksesserer et felles objekt x ved node n_a slik at t_i og t_j kommer i konflikt, må de to virtuelle nodene vn_{ia} og vn_{ja} være identiske (og må dermed inneholde alle data aksessert både av t_i og t_j)

  En transaksjon aksesserer et objekt x ved node n_a dersom den leser n_a's lokale kopi av x eller skriver til en kopi av x, uavhengig av node.

- *Splittkravet*

  Splittkravet innebærer at idet en fysisk node n_a registrerer at en transaksjon t_i har nådd tilstanden FULLFØRT eller AVBRUTT, utføres følgende operasjon (heretter kalt *splittoperasjonen*):

  Alle dataobjekter t_i er alene om å ha aksessert i vn_{ia}, blant transaksjonene som deler denne virtuelle noden, skal fjernes fra vn_{ia}. Transaksjon t_i trenger nå ikke lenger tas hensyn til av replikeringsalgoritmen. Dersom det ikke finnes noen annen transaksjon som deler virtuell node med t_i ved n_a, betyr dette at vn_{ia} fjernes. Hvis en eller flere slike transaksjoner derimot finnes, beregnes de virtuelle nodene ved n_a på nytt for alle transaksjoner t_k, hvor $vn_{ia} = vn_{ka}$, i overensstemmelse med lokalitets- og unionskravene.

Lokalitets- og unionskravene er tilstrekkelige til å garantere serialiserbarhet, men uten noen mekanisme for å fjerne transaksjoner fra grafen vil en algoritme aldri være praktisk anvendbar.

Splittkravet er basert på kriteriet for fjerning av transaksjoner ved konfliktgraftesting, definert i seksjon 2.5.4: hvis en transaksjon t_i ikke skal utføre flere operasjoner, og det i konfliktgrafen ikke finnes noen vei fra en transaksjon t_j, som ennå er aktiv, til t_i, vil det aldri oppstå en sykel som involverer t_i.

4.4.1 Utførelse av transaksjoner

1. For hver lese- eller skriveoperasjon en transaksjon t_i utfører ved sin utgangsnode:

   - Oppdater replikeringsgrafen *tentativt* i henhold til lokalitets- og unionsregelen. Test om grafen forblir asyklisk.
   - Hvis grafen forblir asyklisk, utføres operasjonen, og endringene i replikeringsgrafen gjøres permanente.
   - Hvis grafen blir syklisk og t_i er en lokal transaksjon, avbrytes t_i.
   - Hvis grafen blir syklisk og t_i er en global transaksjon[2] skal t_i avbrytes dersom sykelen inneholder en transaksjon som er COMMITTET. Hvis ikke, settes transaksjonen til å vente.

2. For hver skriveoperasjon t_i ber om å utføre ved en annen node enn utgangsnoden, utføres operasjonen.

3. Dersom t_i ber om å committes, committes transaksjonen. Hvis t_i nå er FULLFØRT, utfør splittoperasjonen, og undersøk om det finnes ventende transaksjoner som nå kan aktiveres.

[2]Forutsetningen om at transaksjoner er begrenset til å lese objekter ved utgangsnoden, og til å skrive objekter som har transaksjonens utgangsnode som primærnode, gjør at en global transaksjon i denne sammenhengen alltid er en transaksjon som oppdaterer replikerte objekter.

4. Dersom t_i ber om å avbrytes ved utgangsnoden, avbrytes transaksjonen. Splittoperasjonen utføres, og algoritmen undersøker om det finnes ventende transaksjoner som nå kan aktiveres.

5. Dersom t_i ber om å avbrytes ved en annen node enn utgangsnoden, avbrytes transaksjonen ved denne noden. Den må imidlertid senere restartes.

4.4.2 Bevisskisse for global serialiserbarhet

Dersom vi forutsetter at splittoperasjonen ikke er utført, vil vi, som vist i seksjon 4.2, alltid ha en sykel i replikeringsgrafen for en global plan g dersom den globale konfliktgrafen for g er syklisk.

For å vise at algoritmen er korrekt, må vi i tillegg vise følgende:

I I enhver sykel i den globale konfliktgrafen må det finnes en transaksjon som er AKTIV, og vi må kunne bryte sykelen ved å fjerne denne transaksjonen.

II FULLFØRT-kravet må garantere at splittoperasjonen bare fjerner transaksjoner som aldri kan inngå i en sykel i den globale konfliktgrafen.

- **Bevis for I**

  Vi skal vise at idet en sykel s oppstår i den globale konfliktgrafen for en plan g, må s alltid inneholde minst én AKTIV transaksjon.

  La t_i og t_j være to committede transaksjoner i en global plan g, og anta at det oppstår en kant $t_i \rightarrow t_j$ i projeksjonen av g på en node n_c. Committede transaksjoner kan bare opprette nye kanter i den globale konfliktgrafen idet de oppdaterer sekundærkopier, og denne kanten må derfor skyldes at t_j oppdaterer en sekundærkopi ved n_c. Vi antar videre at denne kanten medfører en sykel s i konfliktgrafen for g, og at s bare inneholder committede transaksjoner.

  Opprettelsen av en slik kant krever at vi har et par av operasjoner $o_i(x), o_j(x)$ som utføres ved henholdsvis t_i's og t_j's utgangsnode, og hvor $o_i(x)$ medfører opprettelsen av en virtuell node $vn_{ic}, x \in vn_{ic}$, mens $o_j(x)$ gir en virtuell node $vn_{jc}, x \in vn_{jc}$.

  Uten tap av generalitet antar vi at $o_i(x) <_g o_j(x)$. Men da vil utførelsen av $o_j(x)$ kreve sammenslåing av vn_{ic} og vn_{jc}. Dermed ser vi at kanten $t_i \rightarrow t_j$ i konfliktgrafen for g allerede er representert i replikeringsgrafen før t_j committes ved utgangsnoden. Merk symmetrien, dersom $o_j(x) <_g o_i(x)$, vil dette medføre en sammenslåing av de virtuelle nodene for t_i og t_j ved n_c mens t_i fremdeles er AKTIV.

Dette kan generaliseres til at en committet transaksjon aldri utfører operasjoner som endrer replikeringsgrafen. Dermed må det, i en sykel i replikeringsgrafen som bare inneholder committede transaksjoner, finnes en transaksjon som deltok i s mens den ennå var AKTIV, noe algoritmen ikke tillater.

- **Bevis for II**

  For at en transaksjon t_i skal nå FULLFØRT-tilstanden, må to ting være oppfylt: den må være committet ved alle noder, og det må ikke finnes noen ikke-FULLFØRT transaksjon t_j hvor det går en vei fra t_j til t_i i den globale konfliktgrafen.

  Dette beviset er basert på at det bare kan oppstå en kant fra en transaksjon t_i til en transaksjon t_j dersom t_j ikke er FULLFØRT. Dette følger av at en slik kant krever at t_j utfører en operasjon som er i konflikt med en operasjon t_i allerede har utført.

  La s være en sykel i den globale konfliktgrafen for en global plan g, og la t_i være en FULLFØRT transaksjon som inngår i s.

  Vi har vist at en sykel i den globale konfliktgrafen alltid må involvere en AKTIV transaksjon, og det må derfor også finnes en AKTIV transaksjon t_j i s, og en vei v fra t_j til t_i. Et eller annet sted i v må det da finnes en kant fra en transaksjon t_k, som ikke er FULLFØRT, til en transaksjon t_l som er FULLFØRT. Men siden en FULLFØRT transaksjon er committet ved alle noder, og dermed ikke utfører operasjoner, må denne kanten ha eksistert før t_l ble FULLFØRT, og dette strider mot kravet til FULLFØRT-tilstanden.

4.4.3 Vranglåser

Siden globale transaksjoner kan bli satt til å vente, er algoritmen også sårbar for vranglåser. [BK97] presenterer et teorem som hevder at denne algoritmen eliminerer distribuerte vranglåser som bare involverer to globale transaksjoner. Siden vranglåser som bare involverer én global transaksjon alltid er lokale ved én node, må en distribuert vranglås derfor involvere minst tre globale transaksjoner. Dette vil redusere sannsynligheten for vranglåser betraktelig.

Følgende eksempel viser at dette teoremet ikke kan være riktig:

Eksempel 4.2

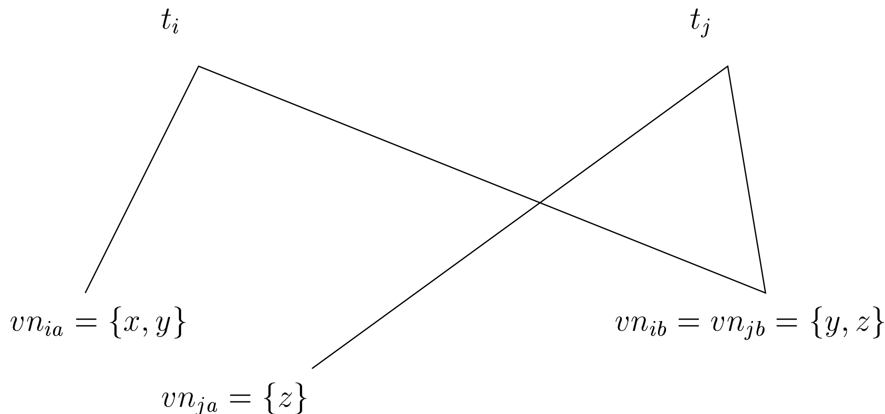
Anta vi har to transaksjoner t_i og t_j:

$t_i = r_i(x)\ w_i(y)\ r_i(z)$
$t_j = r_j(y)\ w_j(z)\ w_j(x)$

La t_i og t_j utføres i en replikert database med to noder, n_a og n_b, og la n_a være primærnode for y, mens n_b er primærnode for x og z. Alle objektene lagres ved begge nodene. Da er g starten på en mulig plan for utførelsen av t_i og t_j:

$g = r_i(x_a)\ r_j(y_b)\ w_i(y_a)\ w_j(z_b)$

Replikeringsgrafen for g ser slik ut:



Dersom t_i utfører $r_i(z)$ ved n_a vil t_i og t_j måtte dele virtuell node ved n_a, siden t_j allerede har skrevet z ved n_b. Dette vil gjøre replikeringsgrafen syklisk, og t_i må derfor vente på t_j. Siden t_i har lest x ved n_a, vil imidlertid vn_{ja} og vn_{ia} også måtte slås sammen dersom t_j utfører $w_j(x)$ ved n_b.

Følgelig må t_j vente på t_i, og vi har en global vranglås.

[BK97] skisserer et bevis for teoremet. [BK99] gir mer detaljert presentasjon av dette beviset, og her gjengis de sentrale momentene. Jeg mener også å ha identifisert feilen som gir rom for moteksempelet:

La t_i og t_j være to globale transaksjoner i vranglås. I replikeringsalgoritmen oppstår en vranglås dersom både t_i og t_j's neste operasjon vil medføre en sykel i replikeringsgrafen, og for at vranglåsen skal være distribuert, må t_i og t_j ha forskjellig utgangsnode, kalt henholdsvis n_a og n_b.

Anta at t_i og t_j til sammen bare er involvert ved to virtuelle noder. Uten tap av generalitet lar vi t_i og t_j være tilknyttet samme virtuelle node vn_b ved n_b, og t_i er i tillegg tilknyttet vn_a ved n_a. t_j vil måtte vente på t_i dersom t_j's neste operasjon er en skriveoperasjon på et replikert objekt som t_i allerede har lest ved vn_a, noe som vil medføre en kant fra t_j til vn_a, og dermed en sykel i replikeringsgrafen.

t_i's operasjoner kan selvsagt ikke medføre at det opprettes noen virtuell node for t_j ved n_b, og t_i fortsetter derfor eksekveringen uten å noensinne måtte vente på t_j.

Følgelig må en vranglås mellom t_i og t_j involvere mer enn to virtuelle noder. [BK99] innfører derfor en tredje virtuell node, vn_p, ved en node n_p.

[BK99] ser ut til å anta implisitt at n_p må være forskjellig fra n_a og n_b. Dette gjør det mulig å vise at man, ved hjelp av begrensningene i hvilke kopier en transaksjon leser og skriver, sikrer at t_i og t_j aldri kan komme i en distribuert vranglås. Replikeringsalgoritmen gir imidlertid ikke noe grunnlag for å forutsette at n_p ikke kan være lik n_a eller n_b. Moteksempelet vårt viser at dersom $n_p = n_a$ og $vn_p = vn_{ja}$, får vi en vranglås dersom t_i's neste operasjon ved n_a er en leseoperasjon på et replikert objekt som allerede inngår i vn_{ja}.

4.4.4 Algoritmens ytelse

Selv om [BK97] er svært optimistiske i forhold til denne algoritmens ytelse, er det ikke publisert målinger som bekrefter dette.

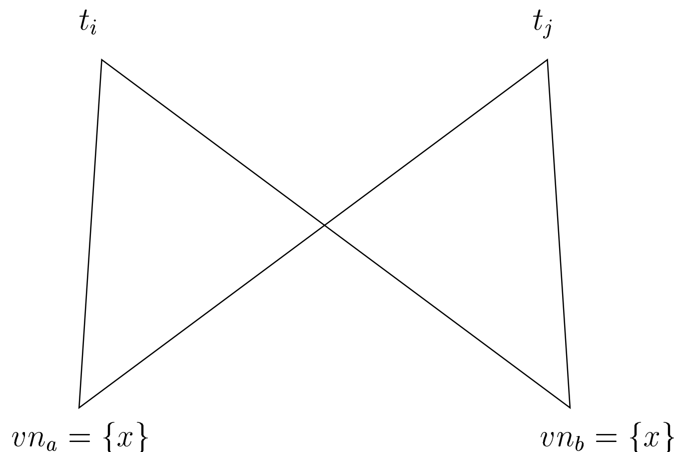
Algoritmen er *konservativ* i den forstand at den vil avvise mange eksekveringsplaner som egentlig er serialiserbare:

Eksempel 4.3

La $t_i = w_i(x)$ og $t_j = w_j(x)$ være to transaksjoner som utføres i en replikert database, og la x være replikert ved to noder, n_a og n_b. Hvis n_a er primærnode for x, er følgende en mulig eksekveringsplan for t_i og t_j:

$g = w_i(x_a)\ c_{ia}\ w_j(x_a)\ c_{ja}\ w_i(x_b)\ c_{ib}\ w_j(x_b)\ c_{jb}$

Replikeringsgrafen for g ser slik ut:



Siden replikeringsgrafen er syklisk, vil ikke denne algoritmen tillate g. Siden algoritmen konservativt må anta at t_j og t_i serialiseres i en annen rekkefølge ved n_b, må $w_j(x_a)$ vente til t_i er FULLFØRT.

I [ABKW98] presenteres derfor en variant av replikeringsgrafalgoritmen som tillater slike eksekveringsplaner:

4.5 Replikeringsalgoritme i [ABKW98]

Denne algoritmen benytter Thomas' Write Rule, introdusert i seksjon 3.5.3, for å synkronisere skrive-skrive-konflikter:

Alle transaksjoner tildeles et unikt tidsstempel, og et objekt lagrer til enhver tid tidsstempelet til den transaksjonen som sist skrev objektet. Thomas' Write Rule sier da at dersom en transaksjon t_i ber om å få oppdatere et objekt x, men t_i's tidsstempel er *lavere* enn tidsstempelet til x, kan skriveoperasjonen ignoreres, mens t_i fortsetter eksekveringen. Dersom t_i's tidsstempel derimot er høyere enn x', utføres operasjonen, og t_i fortsetter som vanlig.

Forutsetningene ellers er uendret, dvs. at transaksjonene fremdeles er begrenset til å lese lokalt, og en skrivetransaksjon kan bare skrive objekter som har transaksjonens utgangsnode som primærnode. Siden alle transaksjoner som oppdaterer et felles objekt derfor må ha felles utgangsnode, sikrer Thomas' Write Rule at vi har definert en serialiseringsrekkefølge for skrive-skrive-konflikter idet transaksjonene tidsstemples. Eksekveringsplaner som i eksempel 4.3 kan dermed tillates gjennom et nytt unionskrav:

- *Unionskravet i [ABKW98]*

  Til enhver tid må en virtuell node vn_{ia} inneholde mengden av dataobjekter t_i så langt har aksessert ved fysisk node n_a. Dersom to transaksjoner t_i og t_j aksesserer et felles objekt x ved node n_a slik at t_i og t_j kommer i konflikt, må de to virtuelle nodene vn_{ia} og vn_{ja} være identiske (og må dermed inneholde alle data aksessert både av t_i og t_j), dersom ett av følgende kriterier er oppfylt:

  - x er en primærkopi, og t_i og t_k er i konflikt på x.[3]
  - x er en sekundærkopi og t_i og t_j er i en lese-skrive- eller skrive-lese-konflikt på x.

  En transaksjon aksesserer et objekt x ved node n_a dersom den leser n_a's lokale kopi av x eller skriver til en kopi av x, uavhengig av node.

Legg merke til at alle par av transaksjoner i skrive-skrive-konflikt på et objekt x må dele virtuell node ved x' primærnode.

[3] [ABKW98] krever at de virtuelle nodene for to transaksjoner t_i og t_j skal slås sammen ved en node n_c dersom de er i direkte eller *indirekte* konflikt på en primærkopi ved n_c. To transaksjoner er i en indirekte konflikt hvis det finnes en vei mellom dem i konfliktgrafen, og denne veien inneholder minst én annen transaksjon. Dersom denne veien ikke inneholder kanter som skyldes skrive-skrive-konflikter på sekundærkopier, er dette kravet opplagt overflødig, siden alle par av sidestilte transaksjoner må dele virtuell node ved n_c, slik at alle transaksjoner i denne veien vil tilhøre samme virtuelle node ved n_c. Dersom veien inneholder kanter som skyldes skrive-skrive-konflikter på sekundærkopier, viser vi i seksjon 4.5.1 at en slik sammenslåing er unødvendig.

Et krav er, som nevnt tidligere, at tidsstemplene må tildeles i transaksjonenes serialiseringsrekkefølge. I [ABKW98, BK99] tildeles transaksjonene et tidsstempel idet de utfører sin *første* operasjon ved utgangsnoden. For at dette tidsstempelet skal tildeles i serialiseringsrekkefølge må man i så fall kreve at enhver lokal node som er primærnode for replikerte objekter, benytter en eller annen form for tidsstempelordning. Dette forutsettes hverken i [ABKW98] eller [BK99].

I vår presentasjon av algoritmen krever vi bare at dersom to transaksjoner t_i og t_j oppdaterer et replikert objekt x, og det finnes en vei fra t_i til t_j i den lokale konfliktgrafen ved deres felles utgangsnode, er $ts(t_i) < ts(t_j)$. Merk at de to transaksjonene må ha samme utgangsnode, siden de oppdaterer et felles objekt, og derfor vil alle transaksjoner som skriver samme objekt tidsstemples i serialiseringsrekkefølge.

Algoritmen i [ABKW98] presenteres i to utgaver: en pessimistisk variant, hvor replikeringsgrafen oppdateres og sjekkes for sykler ved hver operasjon en transaksjon utfører ved utgangsnoden, og en optimistisk variant, hvor replikeringsgrafen bare oppdateres og sjekkes idet transaksjonen skal committes ved utgangsnoden. Den pessimistiske varianten utfører operasjonene på samme måte som i [BK97] (se seksjon 4.4.1), med en ekstra forutsetning om at alle skrive-skrive-konflikter skal håndteres ved hjelp av Thomas' Write Rule.

I den optimistiske utgaven utføres operasjonene i en transaksjon t_i slik:

1. Vi forutsetter som nevnt at alle skrivetransaksjoner bærer et globalt unikt tidsstempel, og at alle par av transaksjoner som oppdaterer et felles, replikert objekt x tidsstemples i serialiseringsrekkefølge.

2. Dersom t_i ber om å få utført en lese- eller skriveoperasjon ved sin utgangsnode, utføres operasjonen.[4] Algoritmen krever at lese- og skrivemengden ved utgangsnoden lagres for å kunne oppdatere replikeringsgrafen senere (se pkt. 4).

3. Dersom t_i ber om å få utført en skriveoperasjon ved en annen node enn utgangsnoden, anvendes Thomas' Write Rule.

4. Dersom t_i ber om å committes ved utgangsnoden, oppdateres replikeringsgrafen *tentativt* ved hjelp av t_i's lese- og skrivemengde. Dersom den resulterende replikeringsgrafen er asyklisk, gjøres endringene permanente, og transaksjonen committes. Hvis ikke, kanselleres de tentative endringene, og t_i må avbrytes.

5. Dersom t_i ber om å committes ved en annen node enn utgangsnoden, tillates dette. Hvis t_i nå er i FULLFØRT-tilstanden, fjernes t_i's node fra replikeringsgrafen (dersom den finnes der), og splittoperasjonen utføres.

[4] [ABKW98] og [BK99] forutsetter at man også bruker Thomas' Write Rule for å håndtere skrive-skrive-konflikter på primærkopier. Så vidt jeg kan se er ikke dette påkrevd for algoritmens del, siden vi forutsetter at alle transaksjoner i skrive-skrive-konflikt tidsstemples i serialiseringsrekkefølge.

6. Dersom t_i ber om å avbrytes ved sin utgangsnode avbrytes den, og splitt-operasjonen utføres.

7. Dersom t_i ber om å avbrytes ved en annen node enn utgangsnoden, avbrytes transaksjonen ved denne noden. Den må imidlertid senere restartes.

4.5.1 Bevisskisse for global serialiserbarhet

Vi skal vise at algoritmen i [ABKW98] garanterer global serialiserbarhet. Vi har allerede vist korrektheten for den pessimistiske algoritmen fra [BK97], men dette beviset er basert på en mer konservativ unionsregel. La s være en sykel i konfliktgrafen for en global plan g. Vi må da vise at enhver replikeringsgraf for g inneholder en sykel, også dersom det finnes kanter i s som skyldes skrive-skrive-konflikter på sekundærkopier.

La g være en global plan, og la s være en sykel i konfliktgrafen for g. La t_i og t_j være to transaksjoner som begge skriver et replikert objekt x i g, og la en kant $t_i \rightarrow t_j$ inngå i s.

Thomas' Write Rule sikrer at t_i og t_j serialiseres i samme rekkefølge ved alle noder, og det vil dermed ikke finnes noen kant $t_j \rightarrow t_i$ i g. Unionsregelen sier at t_i og t_j må dele virtuell node ved x' primærnode, kalt n_c, som også er utgangsnode for begge transaksjonene. Hvis vi kaller denne felles virtuelle noden vn_c, har vi en vei $t_i - vn_c - t_j$ i enhver replikeringsgraf for g.

Generelt har vi at hvis det ved en node går en kant fra en transaksjon t_k til en transaksjon t_l som skyldes en skrive-skrive-konflikt på en sekundærkopi av et objekt y, må det finnes en kant fra t_k til t_l også ved y's primærnode. Kanter som skyldes skrive-skrive-konflikter på sekundærkopier vil derfor aldri være avgjørende i den globale konfliktgrafen, siden det alltid vil gå en tilsvarende kant ved det aktuelle objektets primærnode.

Beviset i seksjon 4.2, hvor vi viser at replikeringsgrafen alltid er syklisk dersom den globale konfliktgrafen er syklisk, kan derfor også brukes for å vise at dersom konfliktgrafen for en global plan g inneholder en sykel som ikke er lokal ved en node, vil denne sykelen være representert i enhver replikeringsgraf for g, selv om vi ser bort fra skrive-skrive-konflikter på sekundærkopier. Dette forutsetter korrekt anvendelse av Thomas' Write Rule, men viser da at den pessimistiske replikeringsalgoritmen i [ABKW98] garanterer global serialiserbarhet.

At den optimistiske varianten også gir global serialiserbarhet, kan vises slik:

Enhver transaksjon t_i oppdaterer og validerer replikeringsgrafen rett før den committes ved utgangsnoden. Etter at en transaksjon committes ved utgangsnoden kan den bare utføre oppdateringer på sekundærkopier. Siden vi forutsetter at de lokale nodene garanterer lokal serialiserbarhet, kan ikke t_i være involvert i noen

lokal sykel. I seksjon 4.4.2 viste vi at oppdatering av sekundærkopier aldri medfører endringer i replikeringsgrafen, og dersom t_i senere blir involvert i en sykel i den globale konfliktgrafen, vil denne alltid involvere en AKTIV transaksjon.

Det er verdt å merke seg at siden transaksjonene aldri venter i den optimistiske algoritmen, er den immun mot globale vranglåser.

4.5.2 Ytelsesmålinger i [ABKW98]

[ABKW98] presenterer simuleringer hvor den optimistiske og pessimistiske varianten av replikeringsalgoritmen (med Thomas' Write Rule) sammenlignes med den doven-primærnodebaserte algoritmen som omtales i [GHOS96] (se seksjon 3.5.3).

Simulator

Simuleringene er utført med en modell skrevet i C++, som sammen med simuleringspakken CSIM[5] er brukt for å måle algoritmenes ytelse i replikerte databaser av forskjellig størrelse.

Den største utfordringen med replikeringsgrafalgoritmen er å finne en effektiv håndtering av replikeringsgrafen. Simuleringen i [ABKW98] forutsetter en sentralisert replikeringsgraf hvor én dedikert node har ansvar for denne grafen, i praksis vil denne funksjonen også kunne utføres av en av databasenodene. Vranglåser håndteres for alle tre algoritmene ved hjelp av nedtellingsur, hvor man avbryter transaksjoner som venter lenger enn en fastsatt grense.

Algoritmene måles i ulike miljøer:

- I et ATM-basert WAN med 100 noder, som er knyttet sammen med et OC3-nettverk[6], dvs. 155 Mb/s og en latens på 0.0004 sekunder.
- I et mer geografisk spredt WAN, som fremdeles er ATM-basert, men hvor nodene er knyttet sammen via et OC1-nettverk, dvs. 55 Mb/s og latens på 0.1 sekunder. Her varieres antallet noder, resultatene presenteres først for 100 noder, deretter for 20 noder, og til slutt økes antallet noder gradvis fra 2 til 140 noder.

Transaksjonsraten for hele systemet varieres fra omkring 100 til 2500 pr. sekund, og hver transaksjon inneholder mellom 5 og 15 operasjoner. [ABKW98] benytter

[5] Mesquite Software, http://www.mesquite.com/

[6] *OC (Optical Carrier)*-skalaen klassifiserer nettverk ut fra kapasitet, hvor et OC1-nettverk har kapasitet på 55 Mb/s mens et OC3-nettverk kan bære 155 Mb/s.

en andel skrivetransaksjoner på 10%, hvor 30% av operasjonene i hver skrivetransaksjon er skriveoperasjoner. I målingen hvor antallet noder gradvis stiger, holdes transaksjonsraten fast på 15 transaksjoner i sekundet pr. node.

Målingen begrenser seg til «hot spots», dvs at man antar en applikasjon hvor en forholdsvis liten andel av den totale databasen er i aktiv bruk. Antallet objekter er derfor begrenset til 20 pr. node.

Resultater

Resultatene viser en oppsiktsvekkende stor gevinst ved den optimistiske replikeringsgrafalgoritmen. Det sentrale poenget er gjennomføringsgrad, dvs. hvor stor andel av transaksjonene algoritmen er i stand til å utføre. I alle de ulike miljøene er den optimistiske algoritmen i stand til å utføre opp mot 2500 transaksjoner, uavhengig av antall noder og uten at antallet transaksjoner som avbrytes er betydelig. Dette er, som ventet, i sterk motsetning til den låsbaserte algoritmen, hvor over halvparten av transaksjonene måtte avbrytes ved en belastning på 2000 transaksjoner i en database replikert over 100 noder i et OC-3 nettverk. I et OC-1 nettverk med 20 noder, dvs. hvor antallet objekter bare er 400 (noe som øker sannsynligheten for vranglåser), måtte mer enn halvparten av transaksjonene avbrytes allerede ved en belastning på 500 transaksjoner i sekundet.

Den pessimistiske replikeringsgrafalgoritmen gjør det langt bedre enn den låsbaserte, noe som ikke er overraskende siden den, på tross av moteksempelet i eksempel 4.2, unngår en rekke situasjoner som i den låsbaserte algoritmen fører til vranglås. Den største utfordringen for denne algoritmen ser ut til å være at grafnoden utgjør en flaskehals. Et stort antall samtidige transaksjoner øker belastningen på grafnoden, og dette er særlig tydelig dersom databasen er replikert over mange noder. Dette forsinker transaksjonene, og sannsynligheten for vranglåser øker. I forsøket med et variabelt antall noder i et OC-1 nettverk gir den låsbaserte algoritmen bedre ytelse enn den pessimistiske replikeringsgrafalgoritmen ved omkring 110 noder. Dette skjer imidlertid ikke for den optimistiske algoritmen, hvor transaksjonene bare belaster grafnoden idet de committes ved utgangsnoden. Selv om antallet noder økes fra 2 til 140 ligger avbruddsraten fast omkring to prosent, og algoritmen utfører altså så godt som alle transaksjonene.

4.5.3 Optimalisering av lesetransaksjoner

En interessant observasjon, som hverken nevnes i [BK97], [ABKW98] eller [BK99], er at FULLFØRT-tilstanden gjør at alle lesetransaksjoner som kun leser fra fullførte transaksjoner, umiddelbart oppfyller kravene til FULLFØRT. Slike transaksjoner kan committes uten å involvere replikeringsgrafen, og det er grunn til å tro at dette gir en ytelsesbedring, særlig for den pessimistiske replikeringsgrafalgoritmen hvor belastningen på grafnoden er en kritisk faktor.

I praksis kan dette implementeres ved å sette et flagg på hvert objekt idet det skrives av en transaksjon t_i, for deretter å fjerne flagget idet t_i registreres som FULLFØRT. I den optimistiske varianten er både skrive- og lesemengden kjent ved validering: dersom skrivemengden for en transaksjon er tom, og ingen av leseoperasjonene har lest fra transaksjoner som ikke er FULLFØRT, trenger ikke replikeringsgrafen oppdateres.

Den pessimistiske algoritmen kan gjøre følgende:

Så lenge en transaksjon bare utfører leseoperasjoner som leser fra fullførte transaksjoner, logges operasjonene. Idet transaksjonen leser fra en ikke-FULLFØRT transaksjon eller utfører en skriveoperasjon, tas alle slike operasjoner med i den tentative replikeringsgrafen, og utførelsen videre må valideres for hver operasjon. Dersom dette aldri inntreffer vil transaksjonen committes uten at replikeringsgrafen har vært involvert.

4.6 Replikeringsgrafer og globale snapshot

I seksjon 3.4.3 presenterte vi en algoritme hentet fra [BHG87], hvor globale lesetransaksjoner leser fra et *snapshot*, mens skrivetransaksjoner koordineres ved hjelp av en samtidighetskontrollalgoritme som ikke tar hensyn til eksistensen av flere versjoner. Som tidligere nevnt må man sikre at de involverte nodene har et felles bilde av hvilke transaksjoner som skal inngå i snapshotet, og her foreslås en replikeringsgrafalgoritme som benytter et slikt snapshot.

4.6.1 Globale snapshot

La t_i være en lesetransaksjon i en global plan g, og la t_i lese et objekt x fra en FULLFØRT transaksjon t_j, ved en fysisk node n_a. Vi har et konsistent snapshot dersom vi i en slik situasjon alltid kan sikre at det ikke finnes noen annen, fullført transaksjon t_k som også har skrevet x, og hvor det finnes en vei fra t_j til t_k i den globale multiversjons-serialiseringsgrafen. Dette kan gjøres slik:

Vi definerer kravet til FULLFØRT-tilstanden som før: en transaksjon t_i er FULLFØRT dersom den er committet ved alle noder hvor den utføres, og det heller ikke finnes noen node n_a, hvor projeksjonen av den globale konfliktgrafen på n_a inneholder en kant fra en transaksjon t_j, som ennå ikke er FULLFØRT, til t_i.

Idet en transaksjon t_i fullføres, tildeles den et globalt unikt, monotont voksende tidsstempel. Dette tilordnes også alle versjoner opprettet av t_i, men FULLFØRT-meldingen må uansett distribueres til alle noder som lagrer replikerte data.

Et potensielt problem er at dersom en lesetransaksjon t_i utføres ved en node n_a, har vi ingen garanti for at det er kjent ved n_a hvilke transaksjoner som er FULLFØRT ved t_i's oppstart. Dette kan gi rom for ikke-serialiserbar eksekvering, illustrert ved følgende eksempel:

Eksempel 4.4

Anta vi har tre transaksjoner t_i, t_j og t_k:

$t_i = r_i(y)\ w_i(x)$
$t_j = w_j(y)$
$t_k = r_k(x)\ r_k(y)$

Alle transaksjonene utføres i en multiversjonsdatabase replikert over en mengde noder N, hvor n_a og n_b inngår i N. t_i og t_j har n_a som utgangsnode, mens t_k utføres ved n_b.

t_i og t_j kan da utføres slik:

$m = w_j(y_{ja})\ c_{ja}\ r_i(y_{ja})\ w_i(x_{ia})\ c_{ia}\ w_j(y_{jb})\ c_{jb}\ w_i(x_{ib})\ c_{ib}$.

Vi ser at siden t_i leser y fra t_j, serialiseres t_j foran t_i ved n_a. Vi antar at en tredje node n_c senere registrerer t_i og t_j som FULLFØRT, men at n_b mottar t_i's FULLFØRT-melding før t_j's. m_b er da en plan for utførelsen av t_k ved n_b:

$m_b = r_k(x_i)\ r_k(y_0)\ c_k$

t_k serialiseres etter t_i, men foran t_j, noe som innebærer at t_j serialiseres etter t_i ved n_b, og eksekveringen er følgelig ikke globalt serialiserbar.

Eksekveringen i eksempelet kan unngås ved å kreve at et snapshot er begrenset til alle fullførte transaksjoner som inngår i en komplett rekke av FULLFØRT-tidsstempler, dvs. hvor det ikke finnes noe tidsstempel man (lokalt) ennå ikke har assosiert med en transaksjon. For en global lesetransaksjon, dvs. en transaksjon som leser ved flere noder, vil dette kreve koordinering mellom de involverte nodene for å sikre et felles bilde av FULLFØRT-tilstanden.

4.6.2 Replikeringsalgoritme

Vi skisserer her en algoritme hvor alle skrivetransaksjoner kontrolleres ved hjelp av en replikeringsgraf, mens lesetransaksjonene leser fra et globalt snapshot. Det er et krav at lesetransaksjonene deklareres på forhånd. Dette bør være en overkommelig begrensning, siden vi kan anta at en transaksjon skriver dersom ikke noe annet er eksplisitt angitt[WV01]. Et større offer er at vi mister replikeringstransparens: Hvis en klient først oppdaterer databasen (og transaksjonen committes), vil man forvente at en spørring fra den samme klienten gir oppdaterte data tilbake. Inntil skrivetransaksjonen er FULLFØRT vil man imidlertid bare få tidligere versjoner. I hvilken grad dette er til å leve med, er avhengig av applikasjonen. Man kan «emulere» iherdig replikering dersom klienten ikke får fortsette eksekveringen før transaksjonen er FULLFØRT.

Forutsetninger

[ABKW98] indikerer at den optimistiske replikeringsgrafalgoritmen skalerer godt, og vi konsentrerer oss derfor om en algoritme som benytter optimistisk validering. Et spørsmål er om vi kan sløyfe forutsetningene om at en transaksjoner ikke kan lese objekter som lagres ved andre noder enn utgangsnoden, og at en transaksjon heller ikke kan oppdatere objekter som ikke har transaksjonens utgangsnode som primærnode.

Algoritmen i [BK97] er korrekt uten disse begrensningene, og de er innført for å begrense antallet vranglåser. Siden transaksjonene aldri venter ved optimistisk validering, unngår vi vranglåser helt, og det virker derfor mulig å fjerne denne begrensningen.

Thomas' Write Rule krever at alle par av transaksjoner som er i skrive-skrive-konflikt på et replikert objekt, må tidsstemples i serialiseringsrekkefølge. Å oppnå dette uten at slike transaksjoner har samme utgangsnode krever svært mange nye forutsetninger. Vi ser derfor bort fra algoritmen i [ABKW98], og antar at vi benytter en unionsregel som krever sammenslåing av virtuelle noder for alle par av operasjoner i konflikt, også skrive-skrive-konflikter på sekundærkopier.

Siden [ABKW98] ikke presenterer målinger av algoritmer som ikke benytter Thomas' Write Rule, er det vanskelig å si om en slik algoritme fungerer i praksis. Vi velger likevel å anta at transaksjonene kan lese og skrive alle objekter, og objektene tilordnes derfor ikke lenger primærnoder. Vi trenger imidlertid én forutsetning: en transaksjon må ha skrevet én kopi av alle objekter den skriver, og lest alle objekter den leser, før den committes ved utgangsnoden.

Alle skrivetransaksjoner må ha en globalt unik id, og de må tidsstemples sekvensielt med et globalt unikt, monotont voksende tidsstempel idet de når FULLFØRT-tilstanden. Når en transaksjon t_i fullføres, må det sendes en melding til *alle* noder i den replikerte databasen, hvor t_i assosieres med FULLFØRT-tidsstempelet.

Ved oppstart gis en lesetransaksjon, som startes ved en node n_c, et tidsstempel identisk med det høyeste FULLFØRT-tidsstempelet som er slik at det ikke finnes noe lavere FULLFØRT-tidsstempel som ennå ikke er assosiert med noen transaksjon ved n_c.

Transaksjonsutførelse

Operasjonene i en transaksjon t_i utføres som følger:

1. Hvis t_i utfører en leseoperasjon $r_i(x)$ ved sin utgangsnode:
    - Dersom t_i er en lesetransaksjon, assosieres $r_i(x)$ med en versjon x_k, hvor x_k er den versjonen av x med det høyeste tidsstempelet blant de versjonene som har tidsstempel mindre enn eller lik $ts(t_i)$.

   - Dersom t_i er en skrivetransaksjon, skal t_i lese den siste versjonen av x som er opprettet. Operasjonen logges, for senere å brukes til å oppdatere replikeringsgrafen.

2. Hvis t_i utfører en skriveoperasjon ved sin utgangsnode logges operasjonen for senere å kunne oppdatere replikeringsgrafen.

3. Hvis t_i utfører en skriveoperasjon ved en annen node enn utgangsnoden, utføres operasjonen.

4. Hvis t_i utfører en leseoperasjon ved en annen node enn utgangsnoden:

   - Dersom t_i er en ren lesetransaksjon må vi sikre at den leser fra samme snapshot som ved utgangsnoden. Det enkleste vil være å kreve at dersom denne noden har «hull» i FULLFØRT-serien som kan true snapshotets konsistens, må t_i vente til informasjonen er mottatt (vi antar da at det tar relativt kort tid å spre FULLFØRT-informasjonen til alle involverte noder). Alternativt kan denne informasjonen hentes fra t_i's utgangsnode.
   - Dersom t_i er en skrivetransaksjon, utføres operasjonen.

5. Hvis t_i ber om å committes ved utgangsnoden:

   - Dersom t_i er en skrivetransaksjon oppdateres replikeringsgrafen *tentativt* i overensstemmelse med t_i's lese- og skrivemengde. Dersom den resulterende replikeringsgrafen er asyklisk gjøres endringene permanente og transaksjonen committes. Hvis ikke kanselleres de tentative endringene, og t_i må avbrytes.
   - Dersom t_i er en lesetransaksjon, committes den uten videre.

6. Hvis t_i ber om å committes ved en annen node enn utgangsnoden, committes operasjonen. Hvis t_i er en skrivetransaksjon og dette medfører at t_i er FULLFØRT fjernes den fra replikeringsgrafen, og splittregelen anvendes.

7. Hvis t_i ber om å avbrytes ved sin utgangsnode:

   - Dersom t_i er en lesetransaksjon avbrytes den uten videre.
   - Dersom t_i er en skrivetransaksjon avbrytes den, og splittoperasjonen utføres.

8. Hvis t_i ber om å avbrytes ved en annen node enn utgangsnoden, avbrytes transaksjonen ved denne noden. Den må imidlertid senere restartes.

4.6.3 Snapshotalgoritmens korrekthet

Vi skal vise at snapshotalgoritmen garanterer global serialiserbarhet:

Transaksjonenes tidsstempler definerer en versjonsordning, og i enhver sykel i multiversjons-serialiseringsgrafen for en global multiversjonsplan mg må det derfor inngå en kant fra en transaksjon t_j til en transaksjon t_i hvor $ts(t_i) < ts(t_j)$.

Replikeringsgrafalgoritmen sikrer sammen med FULLFØRT-kravet at alle skrivetransaksjonene fullføres i serialiseringsrekkefølge. Siden vi også tidsstempler skrivetransaksjonene idet de fullføres, vil en kant fra en skrivetransaksjon t_i til en annen skrivetransaksjon t_j alltid innebære at $ts(t_i) < ts(t_j)$. Derfor må enten t_i eller t_j være en lesetransaksjon.

En slik kant kan oppstå på en av følgende måter:

- Transaksjon t_i har lest en versjon x_j av et objekt x. Siden t_j skriver x, må t_i være en lesetransaksjon. Men lesetransaksjoner kan bare lese fra transaksjoner med lavere tidsstempel, og $ts(t_j)$ kan derfor ikke være høyere enn $ts(t_i)$.

- Transaksjon t_i skriver et objekt x, men transaksjon t_j leser x fra en transaksjon t_k, hvor $ts(t_k) < ts(t_i)$. Siden t_i skriver x, må t_j være en lesetransaksjon. Men lesetransaksjoner tidsstemples slik at det ikke kan finnes noen skrivetransaksjon med lavere tidsstempel som ennå ikke er FULLFØRT, og siden t_j alltid skal lese den versjonen med det høyeste tidsstempelet som er mindre enn eller lik $ts(t_j)$, kan ikke $ts(t_i)$ være lavere enn $ts(t_j)$.

Algoritmen garanterer følgelig global serialiserbarhet.

Kapittel 5

Avslutning

5.1 Oppsummering

[GHOS96] viser at vi ikke kjenner noen generelt anvendbar strategi for å utføre transaksjoner i replikerte databasesystemer. Skaleringsproblemene kan langt på vei løses ved å utnytte kjennskap til applikasjonenes integritetsregler, men man mister da noe av gevinsten med generelle systemer for databasehåndtering.

[BK97] presenterer en algoritme basert på *replikeringsgrafer*. Denne tilbyr serialiserbarhet for replikerte databaser, og bør i følge forfatterne gi bedre ytelse enn noen av algoritmene i [GHOS96]. En simuleringsstudie publisert i [ABKW98] bekrefter en markant ytelsesforbedring.

Denne algoritmen krever at man vedlikeholder en global replikeringsgraf, og i modellen som simuleres i [ABKW98] lagres denne i sin helhet ved én enkelt node. Dette gjør systemet mindre robust. En annen viktig begrensning er at alle objekter tildeles en *primærnode*, og en transaksjon som oppdaterer et objekt må starte ved det aktuelle objektets primærnode, noe som krever at alle objekter som skal oppdateres i én transaksjon tilhører samme primærnode.

I denne rapporten har vi vist følgende:

- [BK97], og den etterfølgende [BK99], presenterer et teorem som hevder at en distribuert vranglås som genereres av replikeringsgrafalgoritmen alltid må involvere mer enn to globale transaksjoner. Et moteksempel, presentert i seksjon 4.4.3, viser at dette teoremet må være galt.

- Replikeringsgrafalgoritmen definerer en FULLFØRT-tilstand, hvor en transaksjon bare når denne tilstanden dersom den ikke lenger kan inngå i en sykel i den globale konfliktgrafen. Slike transaksjoner kan da fjernes fra grafen, og vi påpeker i seksjon 4.5.3 at lesetransaksjoner som utelukkende leser fra fullførte transaksjoner, ikke trenger å oppdatere replikeringsgrafen. Dersom

ikke andelen skrivetransaksjoner er svært høy vil dette gjelde de fleste lesetransaksjonene, og det bør derfor gi en merkbar forbedring.

- I seksjon 4.6 presenterer vi en variant av replikeringsgrafalgoritmen som bygger på *multiversjonssnapshot*[BHG87]. Alle lesetransaksjoner leser fra et slikt snapshot, og bare skrivetransaksjoner trenger derfor å kontrolleres ved hjelp av en replikeringsgraf. I denne algoritmen har vi også foreslått å fjerne begrensningene for hvilke objekter en transaksjon kan lese og skrive.

5.2 Videre arbeid

Flere ting kan undersøkes dersom man ønsker å avgjøre om noe av dette har praktisk anvendelse:

- Ingen av de skisserte endringene i replikeringsgrafalgoritmen er utprøvd i praksis. En ny simuleringsstudie, gjerne sammen med en implementasjon i et eksisterende databasesystem, f.eks. MySQL eller PostgreSQL, bør gi mer informasjon om hvorvidt noen av bidragene her er praktisk anvendbare.

- Vi har ikke drøftet replikeringsgrafalgoritmers motstandsdyktighet mot feil. Et opplagt svakt punkt er at grafen lagres ved en sentralisert node, og det vil være interessant å forsøke å distribuere denne grafen. Dersom de virtuelle nodene lagres ved sine respektive fysiske noder, mens transaksjonenes grafnoder alltid lagres ved utgangsnoden, bør man teoretisk kunne vedlikeholde en slik graf ved å benytte en passende algoritme for å serialisere hver transaksjons bruk av denne grafen. Man må anta at en slik distribuert graf vil gi en betydelig forsinkelse, men det er ikke gitt at den er uakseptabel.

- Man bør forsøke å finne en optimal implementasjon av splittoperasjonen. Å gjenoppbygge deler av replikeringsgrafen er kostbart, og det vil trolig være mye å hente på å gjøre denne så effektiv som mulig.

- [GHOS96] viser at sannsynligheten for at en transaksjon deltar i en vranglås avhenger av antallet samtidige transaksjoner sammen med sannsynligheten for at transaksjonene må vente på hverandre. Dette er igjen avhengig av antallet operasjoner pr. transaksjon, eksekveringstiden for hver operasjon og antallet objekter i databasen.

  Sannsynligheten for at en transaksjon må avbrytes ved optimistisk validering er en funksjon av sannsynligheten for at to transaksjoner må vente på hverandre ved pessimistisk validering. I den pessimistisk algoritmen vil valideringen gi en fast forsinkelse pr. operasjon, og resultatene i [ABKW98] indikerer at denne forsinkelsen er relativt stor. Det kan være interessant å

forsøke å finne et forhold mellom den optimistiske og den pessimistiske algoritmen, og dermed undersøke teoretisk under hvilke betingelser den optimistiske algoritmen yter bedre enn den pessimistiske.

Bibliografi

[AAS97] Divyakant Agrawal, Amr El Abbadi og R. Steinke. Epidemic algorithms in replicated databases (extended abstract). I *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Tucson, Arizona*, side 161–172. ACM Press, 1997.

[ABKW98] Todd Anderson, Yuri Breitbart, Henry F. Korth og Avishai Woo. Replication, consistency, and practicality: are these mutually exclusive? I *Proceedings of ACM SIGMOD international conference on Management of data, Seattle, USA*, side 484–495. ACM, 1998.

[BBG+95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil og Patrick O'Neil. A critique of ansi sql isolation levels. I *Proceedings of the 1995 ACM SIGMOD international conference on Management of data, San Jose, California*, side 1–10. ACM Press, 1995.

[BGRS91] Yuri Breitbart, Dimitrios Georgakopoulos, Marek Rusinkiewicz og Abraham Silberschatz. On rigorous transaction scheduling. *IEEE Transactions On Software Engineering*, 17(9):954–960, 1991.

[BHG87] Phillip A. Bernstein, Vassos Hadzilacos og Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[BK97] Yuri Breitbart og Henry F. Korth. Being lazy helps sometimes. I *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Tucson, Arizona*, side 173–184. ACM, 1997.

[BK99] Yuri Breitbart og Henry F. Korth. Replication and consistency in a distributed environment. *Journal of Computer and System Sciences*, 59(1):29–69, 1999.

[BKR+99] Yuri Breitbart, Raghavan Komondoor, Rajeev Rastogi, S. Seshadri og Avi Silberschatz. Update propagation protocols for replicated databates. I *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, side 97–108, 1999.

[CRR96] Parvathi Chundi, Daniel J. Rosenkrantz og S. S. Ravi. Deferred updates and data placement in distributed databases. I Stanley Y. W. Su, redaktør, *Proceedings of the Twelfth International Conference on Data Engineering, New Orleans, Louisiana*, side 469–476. IEEE Computer Society, 1996.

[GHOS96] Jim Gray, Pat Helland, Patrick O'Neil og Dennis Shasha. The dangers of replication and a solution. I *Proceedings of ACM-SIGMOD 1996 International Conference on Management of Data, Montreal, Qeuebeq*, side 173–182. ACM, 1996.

[Had88] Thanasis Hadzilacos. Serialization graph algorithms for multiversion concurrency control. I *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Austin, Texas*. ACM, 1988.

[HP85] Thanasis Hadzilacos og Christos H. Papadimitriou. Algorithmic aspects of multiversion concurrency control. I *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Portland, Oregon*. ACM, 1985.

[KA00] Bettina Kemme og Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems (TODS)*, 25(3):333–379, 2000.

[KR81] H. T. Kung og John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[OV99] M. Tamer Özsu og Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall International, Inc, andre utgave, 1999.

[WV01] Gerhard Weikum og Gottfried Vossen. *Concurrency Control and Recovery in Database Systems*. Morgan Kaufman Publishers, 2001.