UNIVERSITETET
I OSLO

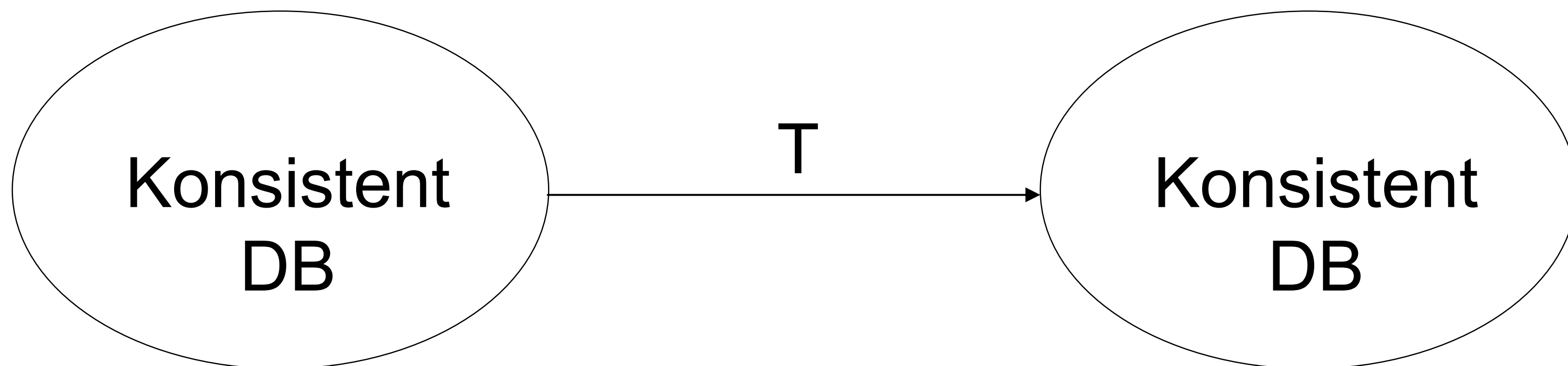
Transaksjonshåndtering og samtidighetskontroll

Ragnar Normann

Mange lysark er basert på en original laget av
Hector Garcia-Molina

Transaksjoner

En ***transaksjon*** er en sekvens av operasjoner som bevarer konsistens i databasen



ACID-egenskapene (repetisjon)

- A – Atomicity
  enten blir hele transaksjonen utført
  eller så blir ikke noe av den utført
- C – Consistency
  transaksjoner skal bevare konsistens
  (dette er en del av definisjonen av begrepet transaksjon)
- I – Isolation
  transaksjoner skal ikke merke at andre transaksjoner
  utføres samtidig med dem selv
- D – Durability
  når transaksjoner er avsluttet, skal effekten av dem
  være varig og ikke kunne påvirkes av systemfeil

Serialiserbarhet (repetisjon)

- En eksekvering av en mengde transaksjoner er ***seriell*** hvis eksekveringen fullføres fullstendig for én transaksjon før neste transaksjon eksekveres.
- Eksekveringen er ***serialiserbar*** hvis transaksjonseksekveringene er slik at det *fins en seriell eksekvering* som gir samme totalresultat

Atomær eksekvering av hver enkelt transaksjon og serialiserbar eksekvering av en samling transaksjoner sikrer at databasen forblir konsistent og at applikasjonen som initierte transaksjonen, opplever resultatet som forutsigbart (isolation).

Eksekveringsplaner

En ***eksekveringsplan*** (“plan”, “schedule”) S for en mengde transaksjoner $\{T_1, \ldots, T_n\}$ er en fletting av operasjonene i $T_1, \ldots, T_n$.

Egenskaper S:

- Hvert element i S er en operasjon i nøyaktig én av transaksjonene
- Hver operasjon i en transaksjon er element i S nøyaktig én gang
- S bevarer rekkefølgen på operasjonene fra hver enkelt transaksjon

Transaksjonseksempel

Integritetsregel: A = B

	T1	T2
	Read(A);	Read(A);
	$A \leftarrow A+100$;	$A \leftarrow A \times 2$;
	Write(A);	Write(A);
	Read(B);	Read(B);
	$B \leftarrow B+100$;	$B \leftarrow B \times 2$;
	Write(B);	Write(B);

Eksekveringsplan S_A

T1	T2	A	B
		25	25
Read(A); A ← A+100;			
Write(A);		125	
Read(B); B ← B+100;			
Write(B);			125
	Read(A); A ← A×2;		
	Write(A);	250	
	Read(B); B ← B×2;		
	Write(B);		250
		250	250

Eksekveringsplan S_B

T1	T2	A	B
		25	25
	Read(A); A ← A×2;		
	Write(A);	50	
	Read(B); B ← B×2;		
	Write(B);		50
Read(A); A ← A+100;			
Write(A);		150	
Read(B); B ← B+100;			
Write(B);			150
		150	150

S_A og S_B er ***serielle*** planer **–** de tar en transaksjon om gangen

Eksekveringsplan S_C

T1	T2	A	B
		25	25
Read(A); A ← A+100;			
Write(A);		125	
	Read(A); A ← A×2;		
	Write(A);	250	
Read(B); B ← B+100;			
Write(B);			125
	Read(B); B ← B×2;		
	Write(B);		250
		250	250

Eksekveringsplan S_D

T1	T2	A	B
		25	25
Read(A); A ← A+100;			
Write(A);		125	
	Read(A); A ← A×2;		
	Write(A);	250	
	Read(B); B ← B×2;		
	Write(B);		50
Read(B); B ← B+100;			
Write(B);			150
		250	150

Eksekveringsplan S_E

T1	T2'	A	B
		25	25
Read(A); A ← A+100;			
Write(A);		125	
	Read(A); A ← A×1;		
	Write(A);	125	
	Read(B); B ← B×1;		
	Write(B);		25
Read(B); B ← B+100;			
Write(B);			125
		125	125

Samme som plan S_D,men med ny T2'

«Gode» eksekveringsplaner

- Vi vil ha eksekveringsplaner som er «gode»
- Begrepet «god» skal være uavhengig av
  - initialtilstanden
  - transaksjonssemantikken
- Begrepet skal bare være avhengig av lese- og skriveoperasjonene og deres innbyrdes rekkefølge
- Det finnes flere mulige definisjoner av «god»
  Formålet er å garantere serialiserbare eksekveringer
  Vi skal først se på «konfliktserialiserbarhet»

Noen nødvendige begreper

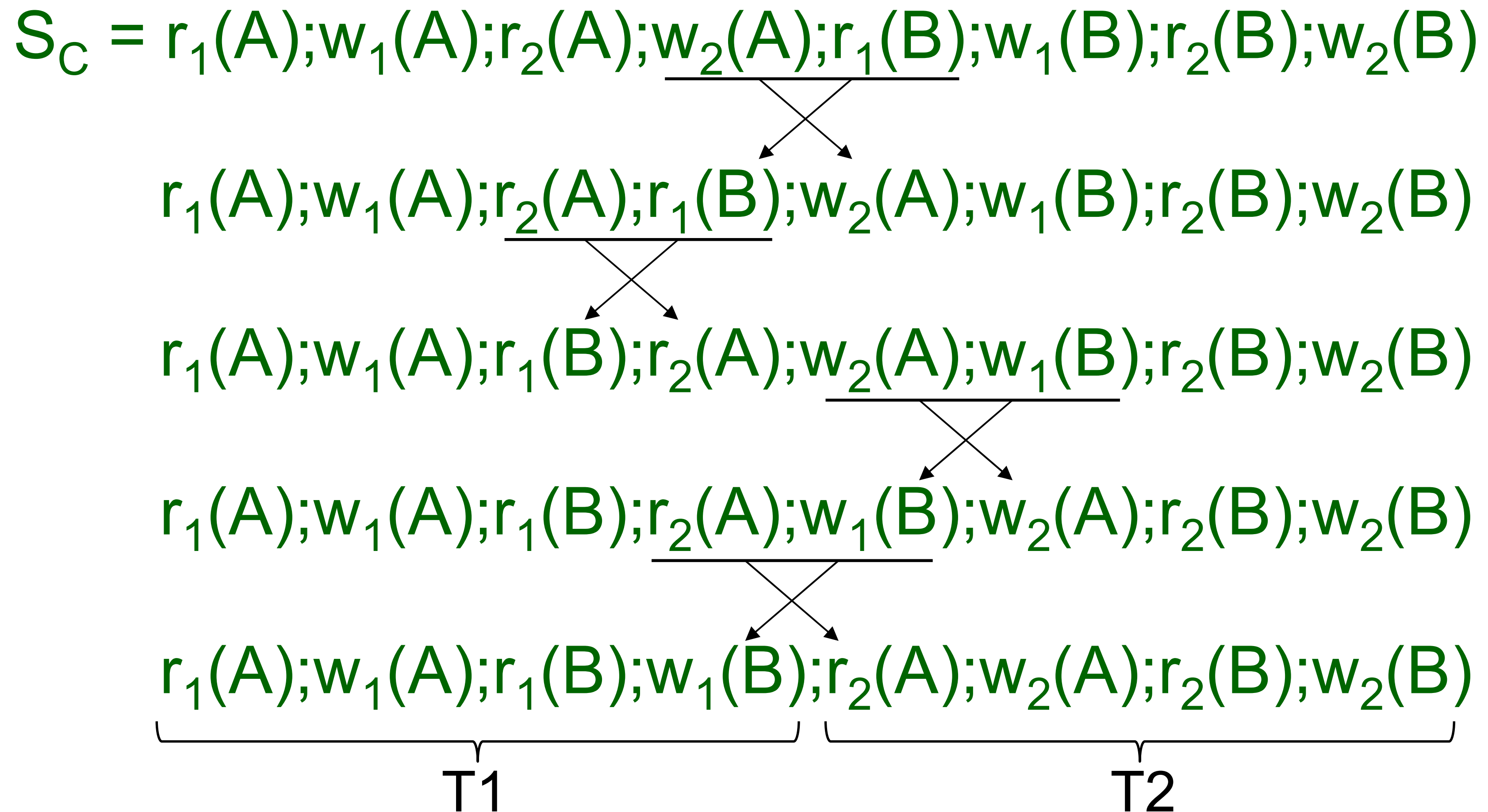
- ***Transaksjon***:
  En sekvens av leseoperasjoner $r_i(A)$ og skriveoperasjoner $w_i(B)$
- ***Eksekveringsplan*** for transaksjonene $\{T_1, \ldots, T_n\}$:
  En fletting av $T_1, \ldots, T_n$
- ***Seriell eksekveringsplan***:
  Plan hvor alle operasjonene i en transaksjon fullføres før neste transaksjon startes
- ***Konflikt i en eksekveringsplan***:
  1. ***Lese-skrive-konflikt***: Et par av operasjoner av formen $...r_i(A)...w_k(A)...$ eller $...w_i(A)...r_k(A)...$ (hvor $i \neq k$)
  2. ***Skrive-skrive-konflikt***: Et par av operasjoner av formen $...w_i(A)...w_k(A)...$ (hvor $i \neq k$)
  3. ***Intratransaksjonskonflikt***: Et par av operasjoner av formen $...o_i(A)...o_i(B)...$
     hvor o_i er w_i eller r_i

Konfliktserialiserbarhet

- To eksekveringsplaner S_1 og S_2 kalles ***konfliktekvivalente*** hvis S_1 kan omformes til S_2 ved en serie ombyttinger av nabooperasjoner som *ikke* er i konflikt med hverandre

- En eksekveringsplan er ***konfliktserialiserbar*** hvis den er konfliktekvivalent med en seriell eksekveringsplan

Eksempel



S_C kan omformes til en seriell eksekveringsplan

Altså er S_C ***konfliktserialiserbar***

En «dårlig» eksekveringsplan

La oss så se på plan S_D:

$S_D = r_1(A);w_1(A);r_2(A);w_2(A);r_2(B);w_2(B);r_1(B);w_1(B)$

- Her har vi en konflikt mellom $w_2(B)$ og $r_1(B)$
  Disse kan ikke bytte plass, så S_D kan ikke være konfliktekvivalent med den serielle planen $T_1;T_2$
- Vi har også en konflikt mellom $w_1(A)$ og $r_2(A)$ som følgelig heller ikke kan bytte plass
  Dermed kan S_D heller ikke være konfliktekvivalent med den serielle planen $T_2;T_1$
- S_D er altså ikke konfliktserialiserbar

En «dårlig» eksekveringsplan (forts.)

$$S_D = r_1(A);w_1(A);r_2(A);w_2(A);r_2(B);w_2(B);r_1(B);w_1(B)$$

Alle konflikter mellom operasjoner i T1 og T2 er tegnet inn.

- Det at T1 må håndtere A før T2 gjør det, kalles at T1 har ***presedens*** over T2 (på A), og vi skriver det slik:
  $T1 \rightarrow T2$ i S_D
- Men vi har også at T2 har presedens over T1 (på B), så vi har både $T2 \rightarrow T1$ og $T1 \rightarrow T2$ i S_D
- Det er denne sykel-avhengigheten som gjør at S_D ikke kan omarrangeres til en seriell eksekveringsplan

En «dårlig» eksekveringsplan (forts.)

Se på plan S_E:

$S_E = r_1(A);w_1(A);r_2(A);w_2(A);r_2(B);w_2(B);r_1(B);w_1(B)$

- Når transaksjonssemantikken er abstrahert bort, ser vi ikke forskjell på S_D og S_E.
- Vi vet at S_E i motsetning til S_D er serialiserbar (se ark 11 der semantikken til transaksjonene i S_E beskrives), men den er *ikke* konfliktserialiserbar.

Konfliktserialiserbarhet $\Rightarrow$ serialiserbarhet

- Enhver konfliktserialiserbar eksekveringsplan er serialiserbar
  - Ombytting av ikke-konflikterende operasjoner vil aldri kunne endre resultatet av eksekveringen
  - Derfor er det tilstrekkelig å bare tillate eksekveringsplaner som er konfliktserialiserbare
- Det fins eksekveringsplaner som ikke er konfliktserialiserbare, men likevel serialiserbare
  - Hvis vi forkaster planer som ikke er konfliktserialiserbare, vil vi derfor kanskje forkaste noen planer som ville gått bra – men det er for dyrt/umulig å sjekke serialiserbarhet generelt
- Hvordan kan man praktisk sjekke konfliktserialiserbarhet?

Presedensgrafer

La S være en eksekveringsplan, og la $p_i(A)$ og $q_k(B)$ være to (vilkårlige) operasjoner i S. Notasjonen

$$p_i(A) <_S q_k(B)$$

betyr at $p_i(A)$ gjøres før (kommer foran) $q_k(B)$ i S
Da defineres ***presedensgrafen til S***, ***P(S)***, slik:

- Noder: Transaksjonene i S
- Kanter: Presedensene i S

  $T_i \rightarrow T_k$ (der $i \neq k$) dersom
  1) $p_i(A) <_S q_k(A)$ og
  2) minst en av p_i og q_k er en skriveoperasjon

<u>Oppgave</u>:

- Tegn P(S) for
  $S = w_3(A);w_2(C);r_1(A);w_1(B);r_1(C);w_2(A);r_4(A);w_4(D)$
- Er S serialiserbar?

Presedensgrafer (forts.)

Lemma

S_1 og S_2 er konfliktekvivalente planer $\Rightarrow P(S_1) = P(S_2)$

Bevis: [Vi viser at $P(S_1) \neq P(S_2) \Rightarrow S_1$ og S_2 ikke er konfliktekvivalente]

Anta at S_1 og S_2 begge er flettinger av transaksjonene $\{T_1, ..., T_n\}$, men at $P(S_1) \neq P(S_2)$.

Da finnes i og k ($i \neq k$) slik at $T_i \rightarrow T_k$ er kant i $P(S_1)$, men ikke i $P(S_2)$.

Det betyr at det finnes operasjoner p_i og q_k som er i konflikt på et dataelement A slik at

$S_1 = \ldots p_i(A) \ldots q_k(A) \ldots$ (derav kanten $T_i \rightarrow T_k$ i $P(S_1)$)

$S_2 = \ldots q_k(A) \ldots p_i(A) \ldots$ (så det er også en kant $T_k \rightarrow T_i$ i $P(S_2)$)

Dette viser at S_1 og S_2 ikke er konfliktekvivalente. q.e.d.

Presedensgrafer (forts.)

Merk!

Vi kan **ikke** slutte motsatt, dvs. fra $P(S_1) = P(S_2)$ at S_1 og S_2 er konfliktekvivalente.

Bevis (Moteksempel):

$S_1 = w_1(A);r_2(A);w_2(B);r_1(B)$

$S_2 = r_2(A);w_1(A);r_1(B);w_2(B)$

S_1 og S_2 er åpenbart ikke konfliktekvivalente.

$P(S_1)$ og $P(S_2)$ har begge de to nodene T_1 og T_2 og de to kantene $T_1 \rightarrow T_2$ og $T_2 \rightarrow T_1$, så $P(S_1) = P(S_2)$.

Presedensgrafer (forts.)

<u>TEOREM</u>

P(S) er asyklisk (sykelfri) ⇔ S er konfliktserialiserbar

<u>Bevis (⇒)</u>:

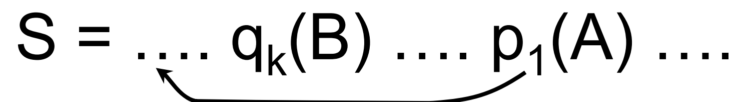
Anta at P(S) er asyklisk. Omform S på følgende måte:

T_1 → T_2, T_1 → T_3, T_2 → T_4, T_3 → T_4

1) Velg en transaksjon T_1 som ikke har noen inngående kanter i P(S)

2) Flytt alle operasjonene i T_1 til starten av S

$$S = \ldots.\ q_k(B)\ \ldots.\ p_1(A)\ \ldots.$$



3) Nå har vi S_1 = <operasjonene i T_1> <resten av S>

4) Gjenta 1–3 for å serialisere resten av S!

Presedensgrafer (forts.)

<u>TEOREM</u>

P(S) er asyklisk $\Leftrightarrow$ S er konfliktserialiserbar

<u>Bevis ($\Leftarrow$)</u>:

Anta at S er konfliktserialiserbar.
Da finnes en seriell plan S_S som er konfliktekvivalent med S.
Siden S_S er seriell, er $P(S_S)$ opplagt uten sykler.
Ifølge foregående lemma er $P(S) = P(S_S)$.
Følgelig er P(S) asyklisk.

q.e.d.

Håndheving av serialiserbarhet og serialiserbarhetsprotokoller

- Metode 1:
  Kjør systemet og registrer P(S)
  På slutten av dagen sjekker vi om P(S) er sykelfri, dvs. om alt gikk bra
- Metode 2:
  Kontroller på forhånd at eksekveringsplanen aldri kan føre til at det blir sykler i P(S)

- Et regelverk som understøtter metode 2, kalles en ***serialiseringsprotokoll***

Låseprotokoller

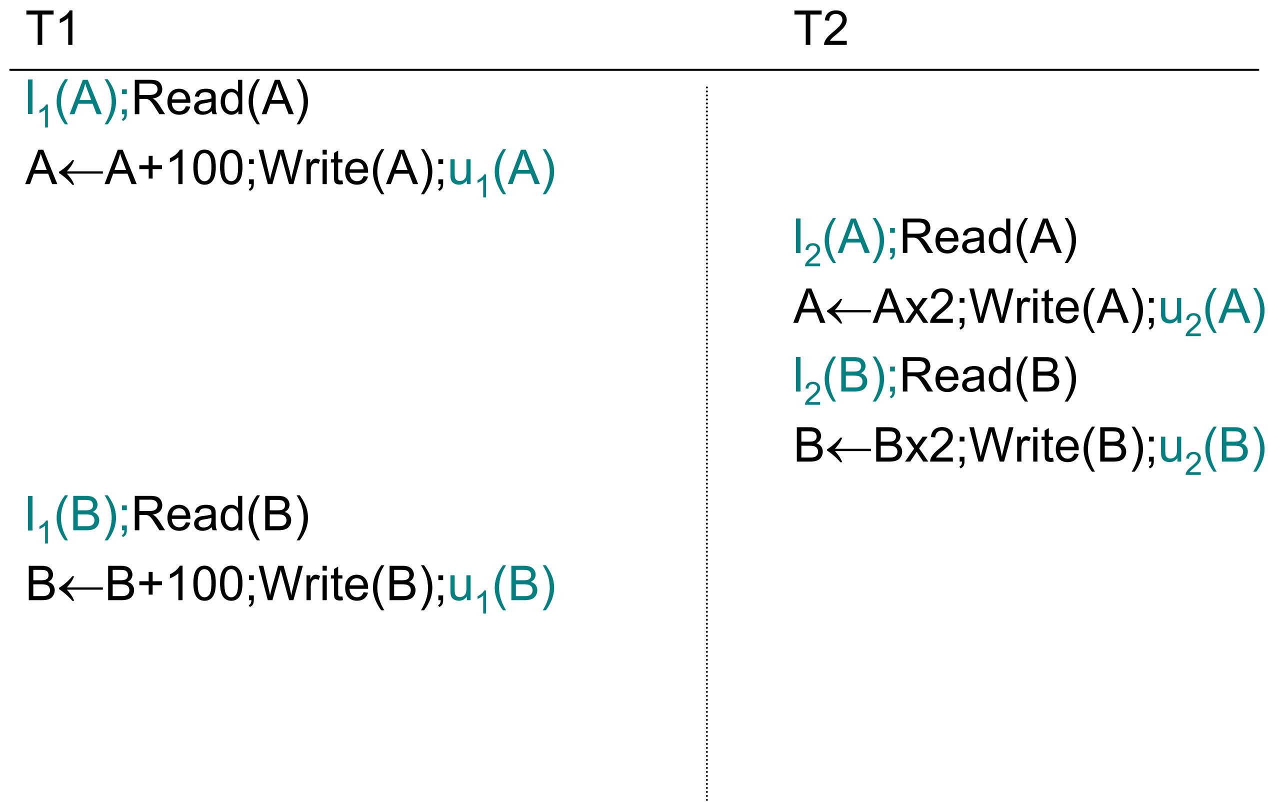
Vi innfører to nye operasjonstyper:

Lock: $l_i(A)$ T_i setter (eksklusiv) lås på A
Unlock: $u_i(A)$ T_i frigir låsen på A

I tillegg må DBMS vedlikeholde en låstabell som viser hvilke dataelementer som er låst av hvilke transaksjoner

De fleste DBMS har en egen «lock-manager»-modul som holder styr på låstabellen

Eksekveringsplan S_D med låser



A	B
25	25
125	
250	
	50
	150
250	150

Låser alene garanterer ikke serialiserbarhet

Låseregler i 2PL (2 Phase Locking)

- **Regel 1 - Velformede transaksjoner:**
  Før T_i gjør en operasjon $p_i(A)$, skal T_i ha gjort $l_i(A)$, og den skal gjøre $u_i(A)$ etter at den har gjort $p_i(A)$

  Eks.: T_i: … $l_i(A)$ … $r_i(A)$ … $w_i(A)$ … $u_i(A)$ …

- **Regel 2 - Lovlige eksekveringsplaner:**
  Eksekveringsplaner kan ikke tillate to transaksjoner å ha lås på samme dataelement samtidig

  Eks.: S: … $l_i(A)$ ……………. $u_i(A)$ …

ingen $l_k(A)$ (for $k \neq i$)

Låseregler i 2PL (forts.)

- **Regel 3 - 2-faselåsing:**
  En transaksjon som har utført en unlock-operasjon, har ikke lov til å utføre flere lock-operasjoner

  T_i = ………… $l_i(A)$ …… $u_i(A)$ ……

  ingen $u_i(B)$

  ingen $l_i(B)$

- Tiden frem til transaksjonens første unlock-operasjon kalles transaksjonens ***voksefase***
- Tiden fra og med transaksjonens første unlock-operasjon kalles transaksjonens ***krympefase***

Konfliktregler for lock/unlock

- $l_i(A)$, $l_k(A)$ gir konflikt
- $l_i(A)$, $u_k(A)$ gir konflikt

- Merk at følgende to situasjoner ***ikke*** gir konflikt:
  - $u_i(A)$, $u_k(A)$
  - $l_i(A)$, $r_k(A)$

Start av krympefasen

- En midlertidig hjelpedefinisjon:
  - $Sh(T_i)$ = første unlock-operasjon som T_i gjør
- Lemma: Hvis $T_i \rightarrow T_k$ i S, så er $Sh(T_i) <_S Sh(T_k)$

  Bevis: At $T_i \rightarrow T_k$ betyr at

  S = … $p_i(A)$ … $q_k(A)$ …; der p_i og q_k er i konflikt

  Regel 1 sier at $u_i(A)$ må komme etter $p_i(A)$ og $l_k(A)$ før $q_k(A)$

  Regel 2 sier at $l_k(A)$ må komme etter $u_i(A)$. Altså har vi

  S = … $p_i(A)$ … $u_i(A)$ … $l_k(A)$ … $q_k(A)$ …;

  Regel 3 sier at $Sh(T_i)$ ikke kan komme etter $u_i(A)$ og at $Sh(T_k)$ må komme etter $l_k(A)$

  Dermed har vi bevist at $Sh(T_i)$ må komme før $Sh(T_k)$ i S

  q.e.d.

2PL sikrer konfliktserialiserbarhet

TEOREM Hvis en plan S overholder regel 1, 2 og 3,
så er S konfliktserialiserbar

Bevis: Ifølge forrige teorem er det nok å vise at
hvis en plan S overholder regel 1, 2 og 3,
så er presedensgrafen P(S) sykelfri

Anta derfor (ad absurdum) at P(S) har en sykel

$$T_1 \to T_2 \to \ldots \to T_n \to T_1$$

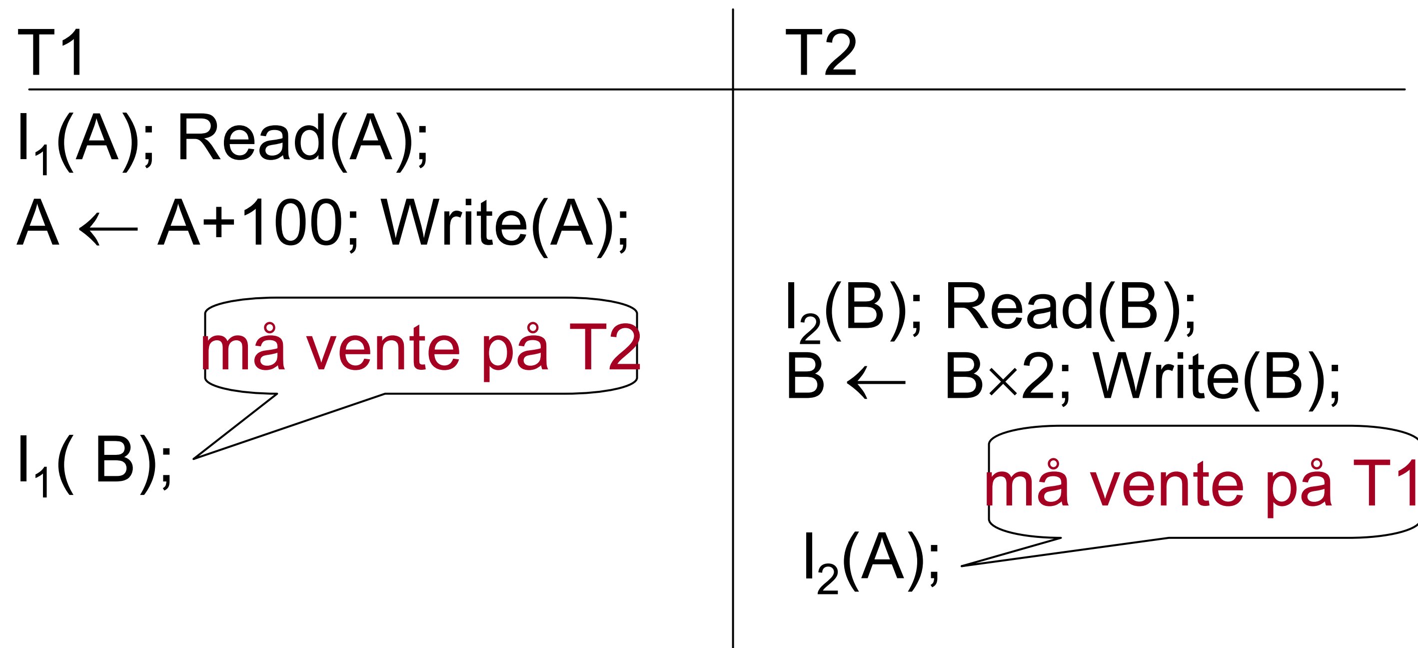
Ifølge lemmaet er da

$$Sh(T_1) <_S Sh(T_2) <_S \ldots <_S Sh(T_n) <_S Sh(T_1)$$

Men det er umulig, så P(S) er sykelfri

q.e.d.

Vranglås



Dette viser at 2PL ***ikke*** sikrer mot vranglås (deadlock)!

Lese- og skrivelåser

- For å få bedre samtidighet kan vi bruke to låsetyper:
  - ***Leselås (Shared lock)*** som tillater andre transaksjoner å lese dataelementet, men ikke å skrive det
  - ***Skrivelås (eXclusive lock)*** som hverken tillater andre transaksjoner å lese eller skrive dataelementet
- Notasjon:
  - $sl_k(A)$ – T_k setter leselås på A
  - $xl_k(A)$ – T_k setter skrivelås på A
  - $u_k(A)$ – T_k sletter sin(e) lås(er) på A
- $sl_k(A)$ blir ikke utført hvis noen annen transaksjon enn T_k har skrivelås på A
- $xl_k(A)$ blir ikke utført hvis en annen transaksjon enn T_k har låst A (det spiller ingen rolle om det er lese- eller skrivelås)

Regler for lese- og skrivelåser

- ***Velformede transaksjoner***
  Enhver transaksjon T_k må overholde disse tre reglene:
  - En $r_k(A)$ må komme etter en $sl_k(A)$ eller $xl_k(A)$ uten at det er noen $u_k(A)$ mellom dem
  - En $w_k(A)$ må komme etter en $xl_k(A)$ uten at det er noen $u_k(A)$ mellom dem
  - Etter en $sl_k(A)$ eller $xl_k(A)$ må det komme en $u_k(A)$
- ***2-faselåsing***
  Enhver 2PL-transaksjon T_k må i tillegg overholde:
  - Ingen $sl_k(A)$ eller $xl_k(A)$ kan komme etter en $u_k(B)$ (uavhengig av hva A og B er)

Regler for lese- og skrivelåser (forts.)

- ***Lovlige eksekveringsplaner***
  Hvert dataelement er enten ulåst, eller det har én skrivelås, eller det har en eller flere leselåser

  Dette sikres ved at alle planer S følger disse reglene:
  - Hvis $xl_i(A)$ forekommer i S, må den etterfølges av en $u_i(A)$ før det kan komme en $xl_k(A)$ eller $sl_k(A)$ med $k \neq i$
  - Hvis $sl_i(A)$ forekommer i S, må den etterfølges av en $u_i(A)$ før det kan komme en $xl_k(A)$ med $k \neq i$
- Merk at det er tillatt for en transaksjon å ha både lese- og skrivelås på samme dataelement

  Dette gir større fleksibilitet og *kan* bidra til mer samtidighet (dette er situasjonsavhengig)

Konfliktserialiserbarhet av S/X-planer

<u>TEOREM</u> Hvis en plan S overholder reglene for lese- og skrivelåser på de to forrige lysarkene, så er S konfliktserialiserbar

<u>Bevis:</u> Nær identisk med beviset for at planer som bare bruker skrivelåser, sikrer konflikt-serialiserbarhet

Den eneste forskjellen er at vi får bruk for at hverken

$ls_i(A)$ fulgt av $ls_k(A)$

eller

$ls_i(A)$ fulgt av $u_k(A)$

er en konflikt

Kompatibilitetsmatriser

- Kompatibilitetsmatriser brukes for å lagre regelverket for tildeling av låser når vi benytter flere låstyper
- Matrisene har en rad og en kolonne for hver låstype
- Kompatibilitetsmatriser tolkes slik:

  Hvis T_i ber om å få satt en lås av type K på data-element A, får den det bare hvis det står 'Ja' i kolonne K i alle rader R i matrisen hvor noen annen T_k har en lås av type R på A
- Eksempel: Kompatibilitetsmatrise for S/X-låser

		S	X	(lås T ber om å få på A)
(lås som A har)	S	Ja	Nei	
	X	Nei	Nei	

Oppgradering av låser

- For å gi bedre samtidighet kan vi tillate T å først sette leselås og så oppgradere den til skrivelås ved behov
- <u>Eksempel</u>:

T_1	T_2
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$;
	$sl_2(B)$; $r_2(B)$;
$sl_1(B)$; $r_1(B)$;	
$xl_1(B)$; ***Avslått***	
	$u_2(A)$; $u_2(B)$;
$xl_1(B)$; $w_1(B)$;	
$u_1(A)$; $u_1(B)$;	

Oppgradering av låser (forts.)

- En ulempe er at å oppgradere låser gir økt risiko for vranglås
- Eksempel:

T_1	T_2
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$;
$xl_1(A)$; ***Avslått***	
	$xl_2(A)$; ***Avslått***

- Eksempelet illustrerer at protokoller som bruker oppgradering av låser, bare egner seg hvis det er mange flere lese- enn skrivetransaksjoner

Oppdateringslåser

- En oppdateringslås er en leselås som senere skal oppgraderes til en skrivelås
- Oppdateringslåser betegnes med U (Update lock)
- Kompatibilitetsmatrisen for S/X/U-låser finnes i to varianter (hvor den usymmetriske er mest vanlig):
  - en usymmetrisk som prioriterer skrivetransaksjoner
  - en symmetrisk som prioriterer lesetransaksjoner

Komp.matrise		S	X	U	(ønsket lås)
	S	Ja	Nei	Ja	
(holdt lås)	X	Nei	Nei	Nei	
	U	J/N	Nei	Nei	

Oppdateringslåser (forts.)

- Planer som gir vranglås på grunn av oppgraderinger av lese- til skrivelåser, gjør det ikke med bruk av oppdateringslåser
  (**NB** Det kan være andre årsaker til vranglås!)
- <u>Eksempel</u> (Det som ga vranglås i sted (ark 40)):

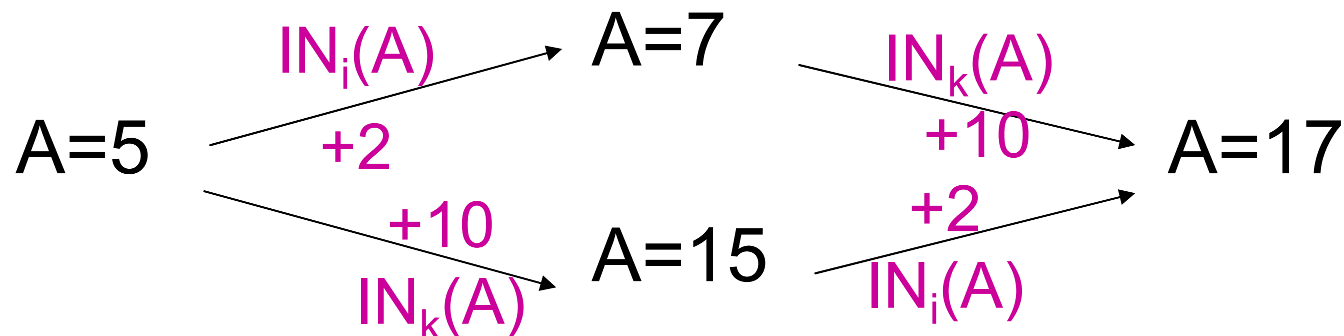
T_1	T_2
$ul_1(A)$; $r_1(A)$;	
	$ul_2(A)$; ***Avslått***
$xl_1(A)$; $w_1(A)$; $u_1(A)$;	
	$ul_2(A)$; $r_2(A)$;
	$xl_2(A)$; $w_2(A)$; $u_2(A)$;

Inkrementlåser

- Atomisk inkrementoperasjon: $IN_i(A)$

  $\{Read(A);\ A \leftarrow A+v;\ Write(A)\}$

  (**NB** Her gjøres Read og Write bare fra og til hukommelsen, ***ikke til disk*** !!)
- $IN_i(A)$ og $IN_k(A)$ er ikke i konflikt!



Inkrementlåser (forts.)

- Hensikten er å effektivisere bokholderitransaksjoner
- Inkrementlåser betegnes med I (Increment lock)
- Inkrementlåser er i konflikt med både lese- og skrivelåser, men ikke med andre inkrementlåser
- Her er kompatibilitetsmatrisen for S/X/I-låser:

Komp.matrise		S	X	I	(ønsket lås)
	S	Ja	Nei	Nei	
(holdt lås)	X	Nei	Nei	Nei	
	I	Nei	Nei	Ja	

Låshåndtering (Lock Scheduling)

- I praksis vil ikke noe DBMS la transaksjonene sette eller frigi noen låser selv
- Transaksjonene utfører bare operasjonene read, write, commit og abort, og eventuelt update og increment
- Låsene legges inn i transaksjonene og settes og frigis av en egen modul i DBMS kalt ***låshåndtereren*** (Lock Scheduler (i noen DBMS kalt Lock Manager))
- Låshåndtereren bruker en egen intern datastruktur, ***låstabellen***, til å administrere låsene
- Låstabellen er ikke en del av bufferområdet; den er utilgjengelig for transaksjonene

Låshåndtering (forts.)

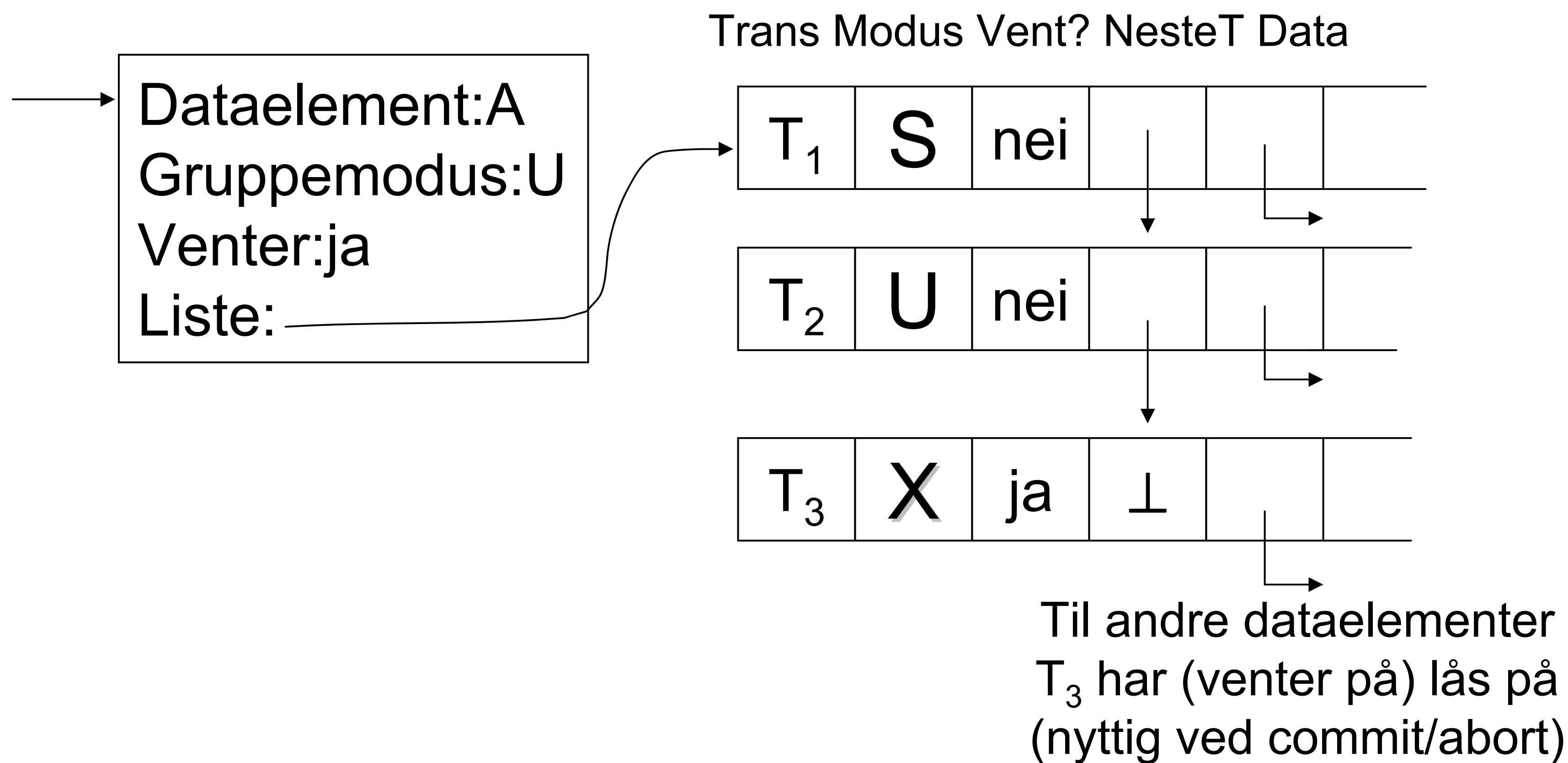
Låshåndtereren består av to deler:

- Del I analyserer hver transaksjon T og legger inn «riktige» låsekrav foran operasjonene i T og legger kravene i låstabellen.
  Hvilke krav den velger, er avhengig av hvilke låstyper som er tilgjengelige
- Del II kontrollerer om operasjonene og låskravene den mottar fra del I kan utføres.
  De som ikke kan det, legges i en kø for å vente på at den låsen som hindrer utførelsen blir fjernet
  (det er en kø for hver lås)
- Når T gjør commit (eller abort), sletter del I alle låser satt av T og varsler del II som sjekker ventekøene for disse låsene og lar de transaksjonene som kan, fortsette

Låstabellen

- Låstabellen er logisk sett en tabell som for hvert dataelement i databasen inneholder all låsinformasjon for dette elementet
- I praksis organiseres låstabellen som en hashtabell med dataelementets adresse som nøkkel
- Ulåste dataelementer er ikke med i låstabellen
- Låstabellen er derfor proporsjonal med antall ønskede og innvilgede låser, og ikke med antall dataelementer
- For hver A i låstabellen lagres følgende informasjon:
  - Gruppemodus (strengeste lås som holdes på A)
  - Et venteflagg som sier om noen venter på å få låse A
  - En liste over de T som har, eller venter på, lås på A

Eksempel på låsinfo for et dataelement A



Granularitet og varsellåser

- Begrepet dataelement er med vilje udefinert
  Tre naturlige granulariter på dataelementer er:
  - en relasjon – et naturlig største (låsbare) dataelement
  - en blokk – en relasjon består av en eller flere blokker
  - et tuppel – en blokk kan inneholde ett eller flere tupler
- Forskjellige transaksjoner kan samtidig ha behov for låser på alle disse nivåene
- For å få til det innfører vi varsellåser, IS og IX, som sier at vi har til hensikt å sette henholdsvis en lese- eller skrivelås lenger ned i hierarkiet

Varsellåser (forts.)

Komp.matrise		IS	IX	S	X	(ønsket lås)
	IS	Ja	Ja	Ja	Nei	
(holdt lås)	IX	Ja	Ja	Nei	Nei	
	S	Ja	Nei	Ja	Nei	
	X	Nei	Nei	Nei	Nei	

- Eksempel: T vil skrive tuppel A i blokk B i relasjon R
  1. Hvis R hverken har S-lås eller X-lås, setter T IX-lås på R
  2. Fikk T satt IX-lås på R, sjekker den om B har S- eller X-lås
     Har ikke B det, setter T IX-lås på B
  3. Fikk T satt IX-lås på B, sjekker den om A har noen lås
     Har ikke A det, setter T X-lås på A og kan skrive A
- Merk at hvis en transaksjon T har en skrivelås på R,
  kan ingen andre skrive noe tuppel i R før T sletter låsen

Håndtering av fantomtupler

- Eksempel:
- Vi skal summere et felt for alle tuplene i en relasjon R
- Før summeringen setter vi leselås på alle tuplene i R for å sikre oss et konsistent svar
- Under summeringen legger en annen transaksjon inn et nytt tuppel i R, noe som gjør at summen kan bli gal
- Dette er mulig fordi tuppelet ikke eksisterte da vi satte våre leselåser (et slikt tuppel kalles et ***fantomtuppel***)
- Løsningen er å sette en IS-lås på relasjonen
  Da får ingen legge inn nye tupler før låsen er slettet