

Leksjon 8: Transaksjoner

Jarle Håvik

DAT2000 – Database 2



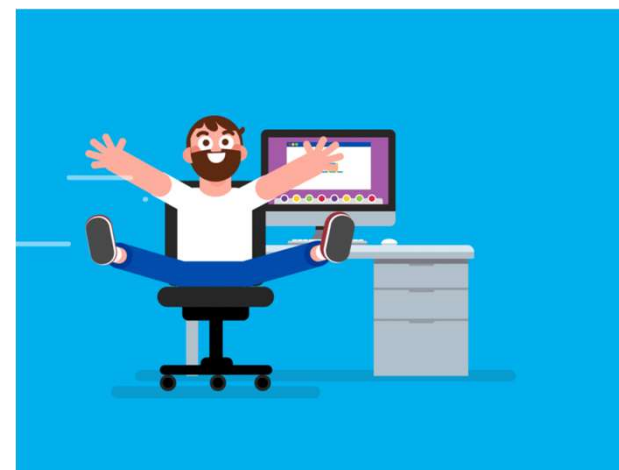
Agenda

❖ Transaksjonshåndtering

- Samtidighet
 - Samtidighetskontroll
 - Samtidighetsproblem – 3 situasjoner
 - Serialitet
- Teknikker for samtidighetskontroll
 - Låsing
 - Vranglås (Deadlock)

Kunnskaps- og ferdighetsmål

- Kunnskap:
 - *transaksjonshåndtering, samtidighet og låsing i en flerbrukerdatabase*
- Forstå hva transaksjoner er, og hvilke egenskaper de bør ha.
- Kunne bekrefte og avbryte transaksjoner med SQL.
- Forstå hvilke utfordringer feilsituasjoner og samtidighet medfører for systemet med hensyn til transaksjoner.
- Forstå hvordan loggføring brukes for å håndtere feilsituasjoner.
- Forstå hvordan låser brukes for å håndtere samtidighetsproblemer.
- Ferdigheter:
 - Kunne bekrefte og avbryte transaksjoner med COMMIT og ROLLBACK
 - Kunne bruke transaksjoner i lagrede rutiner
 - Kunne aktivere og deaktivere

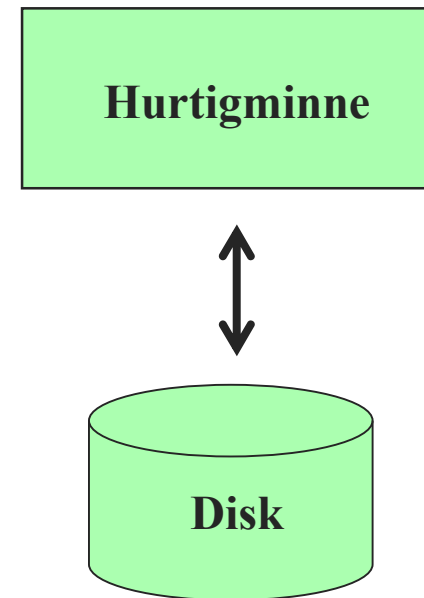


Litteratur

- Silberschatz, Abraham et.al: Database System Concepts
 - Kapittel 17 Transactions- Triggers s 799-828
 - Kapittel 18 Concurrency Control s. 835-880
- Kristoffersen, Bjørn: Databasesystemer
 - Kapittel 10 Transaksjoner (s. 285 – 309)
- Dubois, Paul: MySQL
 - kapittel 2.12 : [Performing Transactions](#)
- Korry Douglas og Susann Douglas: PostgreSQL, 2nd edition: [TRANSACTION PROCESSING](#)
- EDB: [How to work with postgresql transactions](#)

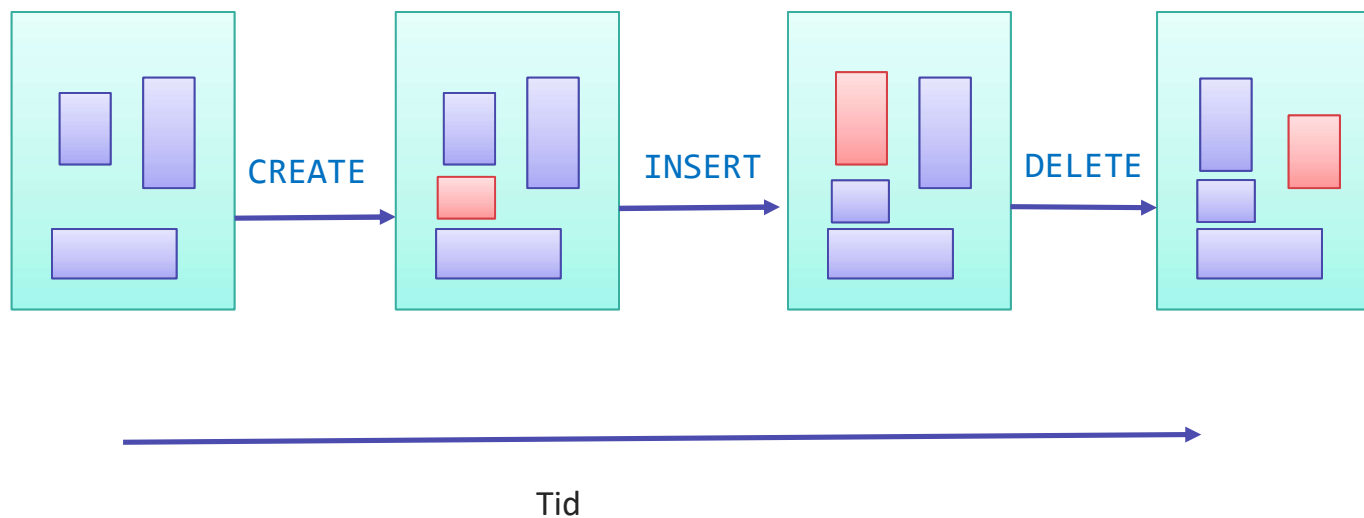
Øyeblikksbilde av en database

- Data **lagres** på disk, men må leses inn i hurtigminnet for **beregning**.
- På et **bestemt tidspunkt** vil en del av databasen eksistere **kun i minnet**.
- Databasesystemer er i produksjon over lang tid.
- Må håndtere **feilsituasjoner**.



Transaksjon

- Handling, eller serie med operasjoner, utført av bruker eller applikasjon, som leser eller oppdaterer innholdet i databasen.



Hva er en «operasjon» ?

- SQL-setninger kan behandle **mange rader**:

UPDATE Ansatt

SET Lønn = Lønn * 1.1

- Lønn til samtlige ansatte blir oppdatert.
- 1 SQL-setning kan altså utføre mange «operasjoner».

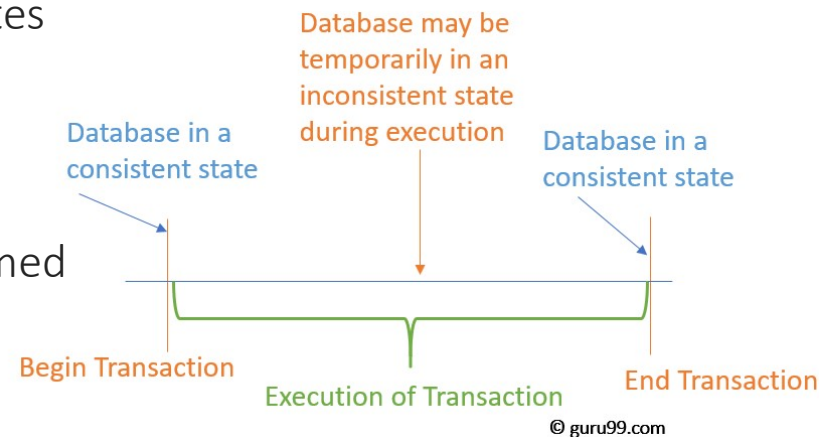
- Motsatt kan vi ønske å betrakte **flere SQL-setninger** som én «sammensatt» operasjon.

- Eksempel fra Hobbyhuset:

Ordrebestilling krever innsetting av rader i Ordre og Ordrelinje, samt oppdatering av Vare.

Transaksjonshåndtering

- Overfører (transformerer) databasen fra en konsistent tilstand til en annen, selv om konsistens kan brytes midlertidig under transaksjonen
- Logisk arbeidsenhet på databasen
- Applikasjonsprogram er en rekke transaksjoner med ikke-databasebehandling imellom.
- Transaksjoner er også enhetene for
 - gjenoppretting, konsistens og integritet!



Transaksjonsbegrepet

- En **transaksjon** er en logisk operasjon mot databasen.
 - Kan involvere en eller flere tabeller.
 - Kan bestå av en eller flere SQL-setninger.
- Transaksjoner kan avsluttes på to måter:
 - **COMMIT**: bekreft, endringene gjøres permanente
 - **ROLLBACK**: angre, transaksjonen «rulles tilbake»
- Utfordringer:
 - **Feil** - for eksempel diskkrasj og strømbrudd
 - **Samtidige brukere** – som kan ødelegge for hverandre

Støtte for utfall av en transaksjon

1. Commit

- En transaksjon er fullført
- Databasen når en ny, konsistent tilstand
- Endringer gjort av en transaksjon lagres permanent i databasen
- Når databasen krasjer, vil eventuelle uforpliktete endringer gå tapt
- Forpliktet transaksjon kan ikke avbrytes.



Støtte for utfall av en transaksjon (2 av 2)

— 2. Aborterer

- Transaksjonen feiler, f.eks databasen kræsjer
- Databasen må tilbakeføres til den konsistent tilstand som var før den startet .
- Rulles tilbake (Rolled back) eller undone
- Avbrutte transaksjoner som er rullet tilbake kan bli startet på nytt på et senere tidspunkt.

```
BEGIN;  
COMMIT;  
SQL  
ROLLBACK;
```

Transaksjoner i SQL

- ❑ Ansatt 22 mottar bestilling av 5 enheter av produkt 4034 fra kunde 1003 – og oppretter ordre 2020:

START TRANSACTION;

INSERT INTO Ordre(OrdreNr,KNr,AnsNr)

VALUES (2020,1003,22);

INSERT INTO Ordrelinje(OrdreNr,VNr,Antall)

VALUES (2020,'4034',5);

UPDATE Vare

SET Antall = Antall-5

WHERE VareNr='4034';

COMMIT;

- ❑ Etter den første/de to første innsettingene (INSERT) er ikke databasen i en lovlig tilstand.

Transaksjoner i MySQL

- **Auto-commit** er standard. Kan skrus av:

```
SET autocommit = 0;
```

- Eller start **eksplisitt** med START TRANSACTION:

```
START TRANSACTION  
INSERT INTO Ordre(OrdreNr,KNr,AnsNr)  
VALUES (2020,1003,22);
```

```
-- Flere kommandoer ...  
COMMIT;
```

Transaksjoner i MySQL

- **Auto-commit** er standard. Kan skrus av:

```
SET autocommit = 0;
```

- Eller start **eksplisitt** med START TRANSACTION:

```
START TRANSACTION  
INSERT INTO Ordre(OrdreNr,KNr,AnsNr)  
VALUES (2020,1003,22);
```

```
-- Flere kommandoer ...  
COMMIT;
```

Commit & Rollback i Postgresql

--Commit & Rollback som bruker "test" fra tutorial 1

BEGIN;

INSERT INTO public.test VALUES (12, 'Test12');

COMMIT;

SELECT * FROM test;

BEGIN;

DELETE FROM public.test WHERE id=12;

ROLLBACK;

BEGIN;

DELETE FROM public.test WHERE id=12;

COMMIT;

De fire grunnleggende egenskapene ved en transaksjon(ACID):



- Atomicity – Atomisitet -
 - ‘Alt eller ingenting’ egenskapen.
Alle eller ingen av deloperasjonene i en transaksjon må fullføres.
- Consistency - Konsistens
 - En transaksjon fører databasen fra en lovlig tilstand til en annen lovlig tilstand.
- Isolation - Isolasjon
 - Delvise effekter av ufullstendige transaksjoner under utførelse skal ikke være synlige for andre transaksjoner før de er bekreftet (Committed).
- Durability - Varighet
 - Effekt av fullførte (committed) transaksjoner lagres i databasen og er permanent - de skal ikke gå tapt på grunn av senere svikt (feil).

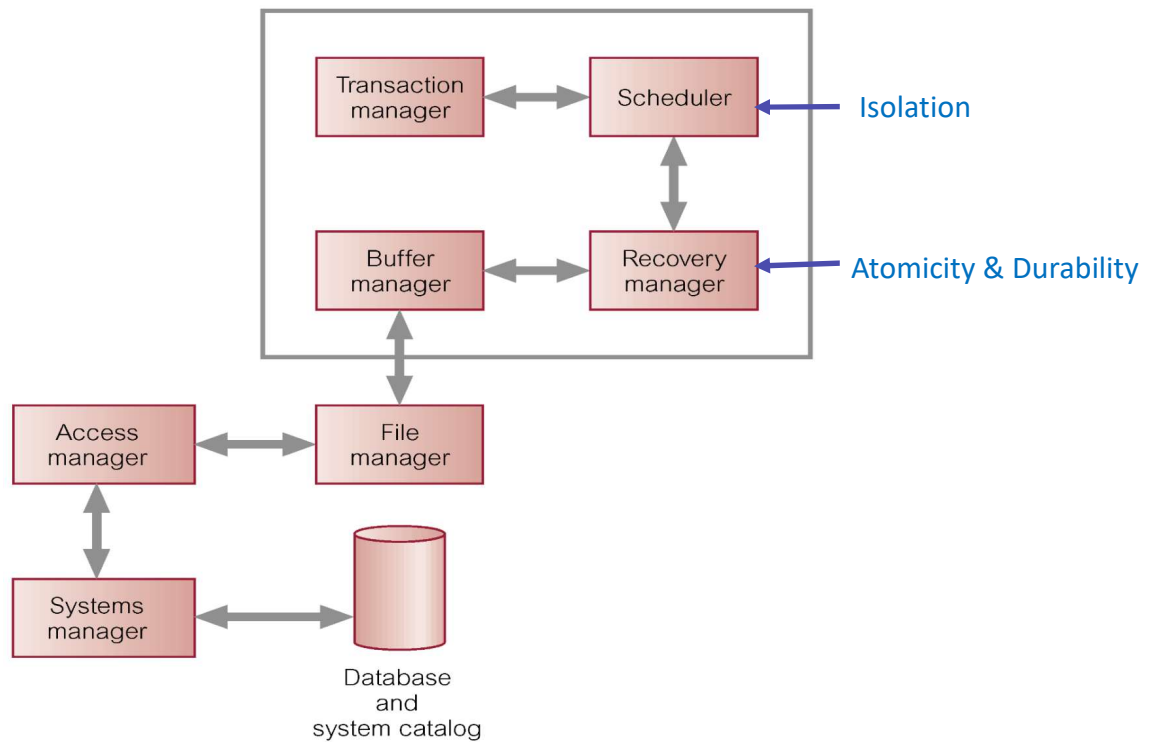
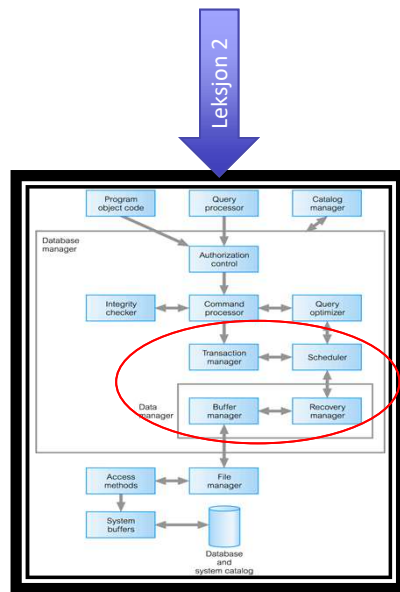
Ansvarlig
→ DBMS

→ Programmerer

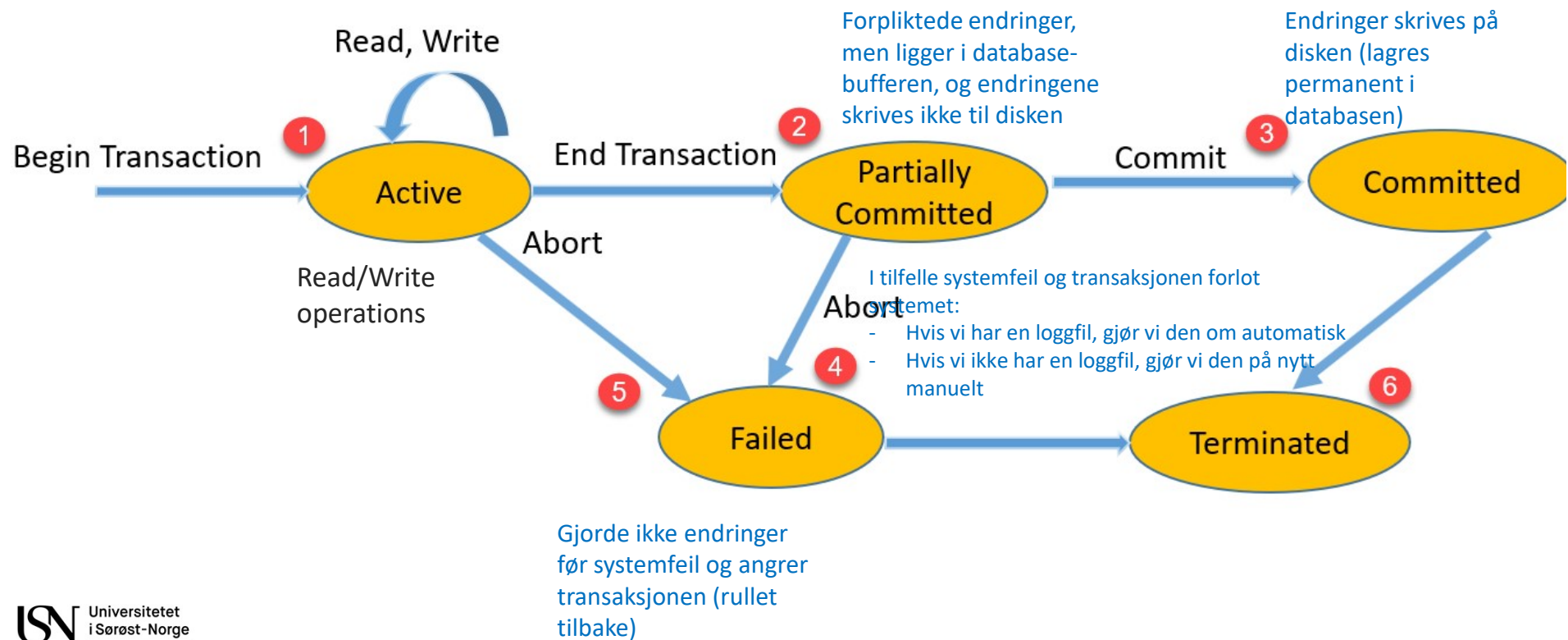
→ DBMS

→ DBMS

DBMS transaksjonsdelsystem



State transition diagram of a transaction:

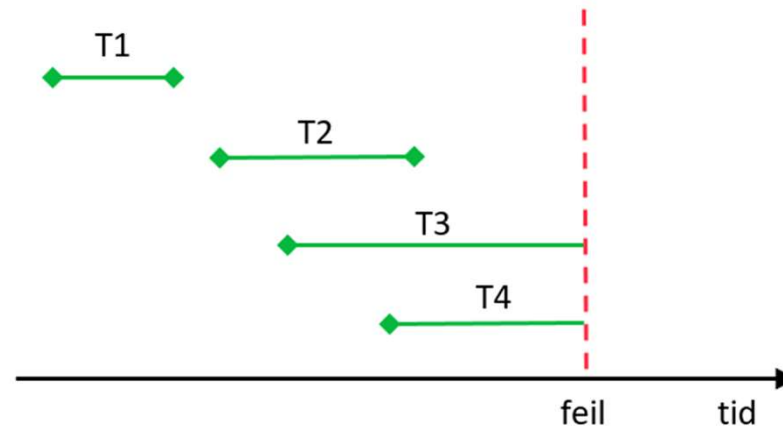


Tilstander under en transaksjon

- En transaksjon går inn i en aktiv tilstand umiddelbart etter at den begynner å utstede lese- og skriveoperasjoner.
- Når transaksjonen avsluttes, beveger den seg til den delvis forpliktete tilstanden. Her blir det foretatt:
 - Noen samtidighetskontrollteknikker som er nødvendige for å sikre at transaksjonen ikke forstyrret andre utførende transaksjoner (isolasjonskontroll).
 - Noen gjenopprettingsprotokoller er nødvendige for å sikre at en systemfeil ikke fører til manglende evne til å registrere endringene i transaksjonen permanent (holdbarhetskontroll).
- Når begge kontrollene er vellykket, sies det at transaksjonen har nådd sitt forpliktelsespunkt og kommer inn i Committed state.
- Når en transaksjon kommer inn i COMMITTED STATE, har den avsluttet utførelsen med suksess.
- En transaksjon kan gå til **failed state** hvis en av kontrollene mislykkes, eller hvis den ble avbrutt i sin aktive tilstand, og det kan da hende at transaksjonen må rulles tilbake for å angre effekten av skriveoperasjonene på databasen.
- Avbrutt tilstand tilsvarer at transaksjonen forlater systemet.
- Mislykkede eller avbrutte transaksjoner kan startes på nytt senere, enten automatisk eller etter at de er sendt inn som nye transaksjoner.

Avbrutte transaksjoner

- Fire transaksjoner i et hendelsesforløp der det oppstår en instansfeil.
- T1 og T2 har skrevet COMMIT til loggen og ble bekreftet før feilen inntraff.
- T3 og T4 var under utførelse. Kanskje hadde T3 rukket å gjennomføre flere skrive-operasjoner, mens T4 kun hadde gjennomført en leseoperasjon.



Eksempel 1 : Bankkonto

Bankkontoer

Kontonamn	Saldo
bal _x	1500
bal _y	750

Overføring av 50\$ bankkonto bal_x til bankkonto bal_y

```
UPDATE Bankkontoer
SET Saldo = Saldo - 50
WHERE Kontonamn = balx;
```

Trekk 50\$ fra bal_x

```
UPDATE Bankkontoer
SET Saldo = Saldo + 50
WHERE Kontonamn = baly;
```

Legg 50\$ til bal_y

Problemer?

- Manglende dekning
- Kontonamn feil
- Syntaksfeil

A: enten overfører vi pengene, eller vi gjør det ikke

C: penger er ikke tapt eller tent

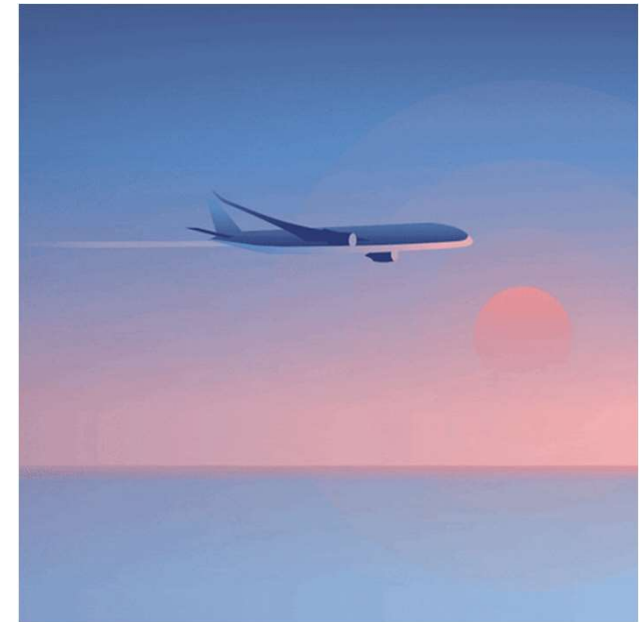
I: T_B skal ikke se endringer hos bal_x og bal_y før T_A committer

D: pengene går ikke tilbake til X permanent!

Eksempel 2 : Reservasjon av flybillett

Flere reisende som leter etter informasjon om ledige seter

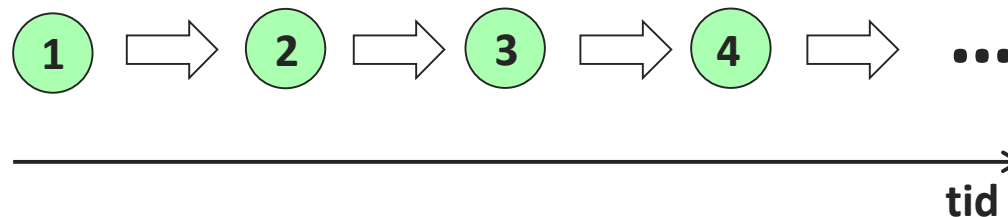
- En reisende reserverer et sete i flight 100.
- Men en annen reisende reserverer samtidig det samme setet.
- Dermed blir informasjonen sett av den første reisende foreldet



Samtidighetskontroll

Transaksjoner og tilstander

- En **tilstand** er innholdet i databasen på et bestemt tidspunkt.
- En **transaksjon** bringer databasen fra én tilstand til en annen.
- Hvis vi tenker oss at kun én transaksjon blir utført av gangen, kan **livsløpet** til en database visualiseres slik:



Samtidighetskontroll

- Ønsker flere samtidige brukere / transaksjoner:
 - Utnytte parallellitet (CPU og I/O)
 - Redusere gjennomsnittlig ventetid (lange transaksjoner)

_____ 1 av gangen

_____ Flere samtidig

- Når er det trygt å tillate samtidig aksess?
 - Mange brukere kan lese samme data samtidig
 - To brukere kan skrive samtidig kun hvis de jobber med forskjellige deler av databasen.

Les-Beregn-Skriv

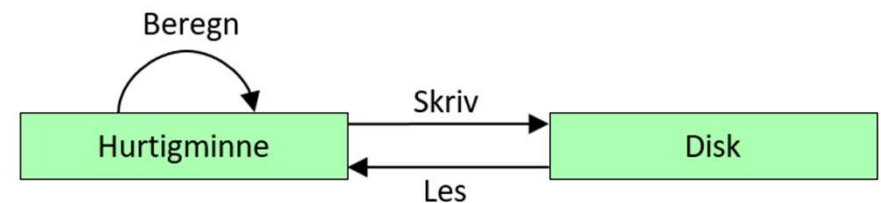
- Hva skjer når en «celle» på ytre lager skal oppdateres?

UPDATE Ansatt

SET Lønn = Lønn * 1.1

WHERE AnsNr = 14

- Transaksjonen består av følgende deloperasjoner:
 1. **Les** post med AnsNr = 14 fra ytre lager
 2. **Beregn** ny lønn
 3. **Skriv** resultat til ytre lager
- Les-Beregn-Skriv er ikke en atomær operasjon.
- Det betyr at to oppdateringer kan «flettes» i tid !



Samtidighetskontroll

Samtidighetskontroll er prosessen med å administrere samtidige operasjoner på databasen uten å få dem til å forstyrre hverandre.

- Selv om to transaksjoner kan være korrekte i seg selv, kan sammenblanding av operasjoner gi et feil resultat.
- Samtidighetskontroll er nødvendig for å sikre isolasjon av transaksjoner, ellers kan det oppstå mange samtidige problemer!

Behovet for samtidskontroll

Tre eksempler på potensielle problemer forårsaket av samtidighet:


1. Tapt oppdatering - Lost update problem (WW konflikt).
2. Uforpliktet avhengighet - Uncommitted dependency problem (WR konflikt).
3. Inkonsekvent analyse - Inconsistent analysis problem (RW Conflict).

Tapt oppdatering (lost update)

Samtidighetsproblemer kan oppstå når to transaksjoner jobber med **samme data til samme tid**.

Vi studerer hva som skjer når to transaksjoner skal oppdatere en verdi A på disken (en verdi i en bestemt kolonne i en bestemt rad i en tabell).

Lokal kopi T1	T1	Verdi A på disk	T2	Lokal kopi T2
120	Les inn	120		
110	Tell ned med 10	120	Les inn	120
110	Skriv til disk	110	Øk med 20	140
		140	Skriv til disk	140



Oppdateringen til transaksjon T1 blir skrevet over av transaksjon T2 og går tapt.

Hvis $A=120$ ved start, så får vi her $A=140$ ved slutt.

Korrekt resultat er $A=130$ ($120+20-10$).

Tapt oppdatering 2 (Write-Write)

Fullført oppdatering overskrives av en annen transaksjon.


- T_1 and T_2 utføres samtidig på den samme kontoen med startverdien 100 £
- T_1 tar ut 10\$
- T_2 setter inn 100\$
- T_1 er ikke isolert fra T_2
- T_1 skriver over data skrevet av T_2

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	$bal_x = bal_x + 100$	100
t_4	$bal_x = bal_x - 10$	write(bal_x)	200
t_5	write(bal_x)	commit	90
t_6	commit		90

Angret oppdatering (dirty read)

Transaksjon T2 bygger på et resultat fra transaksjon T1 som transaksjon T1 seinere «angrer» på at den produserte.

T1	A	T2
	120	
Tell ned A med 10	110	
	110	Les inn A og bruk denne verdien
ROLLBACK	120	



❑ Transaksjoner må ikke få lov til å avlese andres «mellomresultater».

Angret oppdatering 2 - Dirty Read (Write Read)

Oppstår når en transaksjon kan se mellomresultater av en annen transaksjon før den er committed.


- T₃ tar ut 10\$
- T₄ setter inn 100\$
- T₄ avbryter etter oppdatering av saldoen og dens endringer ($bal_x = 200$) har blitt angret (som betyr at saldoen er gjenopprettet til 100)
- T₃ er ikke isolert fra T₄

Time	T ₃	T ₄	bal _x
t ₁		begin_transaction	100
t ₂		read(bal _x)	100
t ₃		bal _x = bal _x + 100	100
t ₄	begin_transaction	write(bal _x)	200
t ₅	read(bal _x)	:	200
t ₆	bal _x = bal _x - 10	rollback	100
t ₇	write(bal _x)		190
t ₈	commit		190

Inkonsistent analyse (incorrect summary)

Transaksjon T2 produserer en rapport basert på noen «gamle» og noen «nye» resultater.

T1	A	B	T2	Lokal sum
	10	50	Nullstill sum	0
	10	50	Legg til A i sum	10
Øk A med 10	20	50		10
Øk B med 20	20	70		10
			Legg til B i sum	80



❑ Selv 1 «leser» og 1 «skriver» kan altså gi samtidighetsproblemer !

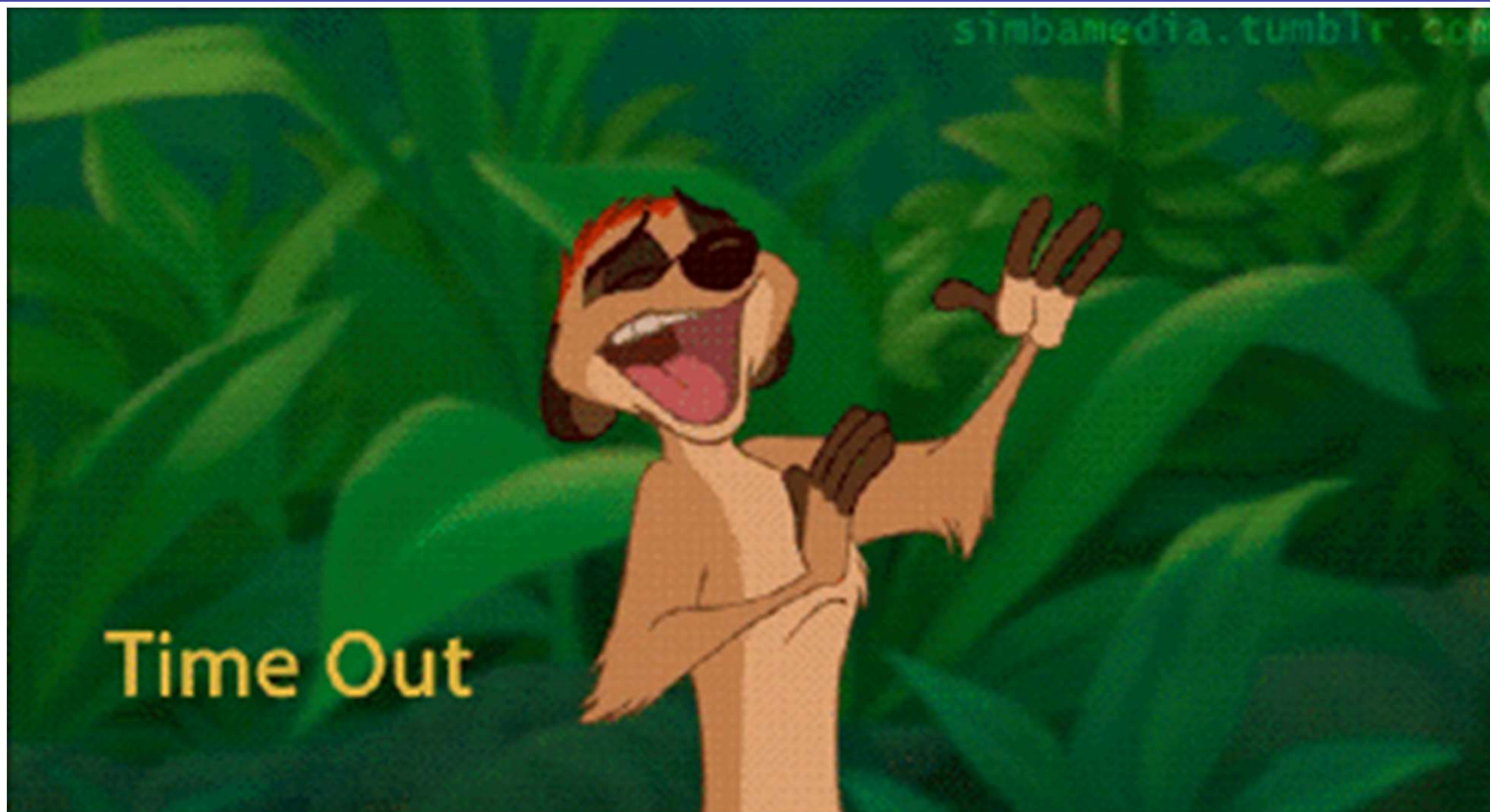
Inkonsistent analyse 2 (RW)

Oppstår når transaksjonen leser flere verdier, men andre transaksjoner oppdaterer noen av dem under utførelsen av den første.

- T₅ overfører 10\$ fra bal_x to bal_z
- T₆ summerer opp toralen for bal_x, bal_y, bal_z
- T₅ endrer på data som T₆ aggregerer

Time	T ₅	T ₆	bal _x	bal _y	bal _z	sum
t ₁		begin_transaction	100	50	25	
t ₂	begin_transaction	sum = 0	100	50	25	0
t ₃	read(bal _x)	read(bal _x)	100	50	25	0
t ₄	bal _x = bal _x - 10	sum = sum + bal _x	100	50	25	100
t ₅	write(bal _x)	read(bal _y)	90	50	25	100
t ₆	read(bal _z)	sum = sum + bal _y	90	50	25	150
t ₇	bal _z = bal _z + 10		90	50	25	150
t ₈	write(bal _z)		90	50	35	150
t ₉	commit	read(bal _z)	90	50	35	150
t ₁₀		sum = sum + bal _z	90	50	35	185
t ₁₁		commit	90	50	35	185

Problemet unngås ved å forhindre at T₆ fra å lese bal_x og bal_z før T₅ har fullført oppdateringene.



Serialiserbarhet

Serialiserbarhet

- Målet med en samtidighetskontrollprotokoll er å planlegge transaksjoner på en slik måte at du unngår forstyrrelser.
- Kan kjøre transaksjoner serielt, men dette begrenser graden av samtidighet eller parallellitet i systemet
- Serialiserbarhet identifiserer de utførelsene av transaksjoner som garanteres for å sikre konsistens.

Serialiserbarhet - Serializability

R: Read, *W*: Write *O*: Object

R_T(O): Transaksjon *T* leser et objekt *O*

W_T(O): Transaksjon *T* skriver et objekt *O*

Schedule - Forløp

- Sekvens av lese/skrive operasjoner utført av et sett med samtidige transaksjoner

Serielt forløp.

- Forløp der operasjoner for hver transaksjon utføres fortløpende uten noen sammenflettede operasjoner fra andre transaksjoner.
- Ingen garanti for at resultatene av alle seriekjøringer av et gitt sett med transaksjoner vil være identiske.

— Ex: T_1 beregner renter, T_2 setter inn 100 000\$.

T1	T2
R(A)	
W(A)	
R(B)	
W(B)	
Commit	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit

Ikke-seriell tidsplan

R: Read, *W*: Write *O*: Object

R_T(O): Transaksjon *T* leser et object *O*

W_T(O): Transaksjon *T* skriver et objekt *O*

- Forløp hvor operasjoner fra ett sett med samtidige transaksjoner er sammenflettet.
- Målet med serialiserbarhet er å finne ikke-serielle tidsplaner som gjør at transaksjoner kan utføres samtidig uten å forstyrre hverandre.
- Med andre ord, vi ønsker å finne tidsplaner som ikke tilsvarer en serieplan. En slik tidsplan sier vi er **serialiserbar (serializable)**.

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)
	Commit
Commit	

Serialiserbarhet

- Et **forløp** (schedule) er som sagt en «fletting» i tid av deloperasjonene til en samling transaksjoner.
 - Opplagt riktig: Kun 1 transaksjon av gangen = **sekvensiell forløp**.
 - **Men**: Det er mer effektivt med samtidige transaksjoner.
- Et **serialiserbart forløp** tillater samtidighet («fletting»), men har samme effekt som et sekvensielt forløp.

T1	A	B	T2
Skrivelås A	10	10	Skrivelås B
Tell ned A med 10	0	20	Øk B med 10 hvis større enn 0
Lås opp A	0	20	Lås opp B
Skrivelås B	0	20	Skrivelås A
Tell ned B med 10	0	10	Øk A med 10 hvis større enn 0
Lås opp B	0	10	Lås opp A

NB! Ikke alle forløp er serialiserbare selv om de bruker låser.

Serialiserbarhet (Serializability)

- Når en ser på seriabilitet er rekkefølgen på Read/Write viktig:
 - a) Hvis to transaksjoner bare leser (RR) et dataelement, kommer de ikke i konflikt og rekkefølgen har ingen betydning.
 - b) Hvis to transaksjoner enten leser eller skriver *separate* dataelementer, kommer de ikke i konflikt og rekkefølgen er ikke viktig.
 - c) Hvis en transaksjon skriver et dataelement og en annen leser eller skriver det samme dataelementet, er rekkefølgen for utførelse viktig.

	T1	T2	Result
Samme dataelement	Read	Read	Ingen konflikt
	Read	Write	Konflikt
	Write	Read	Konflikt
	Write	Write	Konflikt
Forskjellige dataelement	Read/Write	Read/Write	Ingen konflikt

Eksempel: Serialiserbart forløp?

-Start:

A=100 \$,

B=100 \$,

A+B=200 \$

- Ekvivalente ikke-serielle forløp.

Seriell schedule

T ₁	T ₂
read(A) A := A - 50 write(A) read(B) B := B + 50 write(B) commit	read(A) temp := A * 0.1 A := A - temp write(A) read(B) B := B + temp write(B) commit

(Slutt: A=45 \$, B=155 \$, A+B=200 \$)

Ikkje-seriell schedule (serialiserbar)

T ₁	T ₂
read(A) A := A - 50 write(A) commit	read(A) temp := A * 0.1 A := A - temp write(A) commit
read(B) B := B + 50 write(B) commit	read(B) B := B + temp write(B) commit

(Slutt: A=45 \$, B=155 \$, A+B=200 \$)

Begge forløpene produserer samme resultat

Eksempel: Serialiserbart forløp?

-Start:

A=100 \$,

B=100 \$,

A+B=200 \$

-IKKE ekvivalente ikke
serielle forløp

Serial schedule

T_1	T_2
read(A) $A := A - 50$ write(A) read(B) $B := B + 50$ write(B) commit	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B) $B := B + temp$ write(B) commit

(Slutt: A=45 \$, B=155 \$, A+B=200 \$)

Nonserial schedule

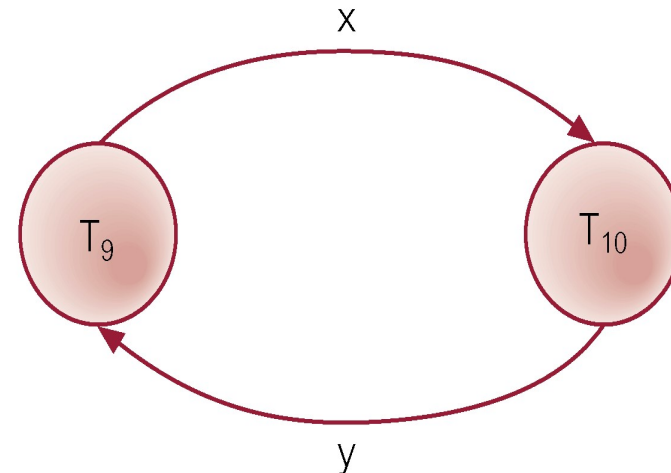
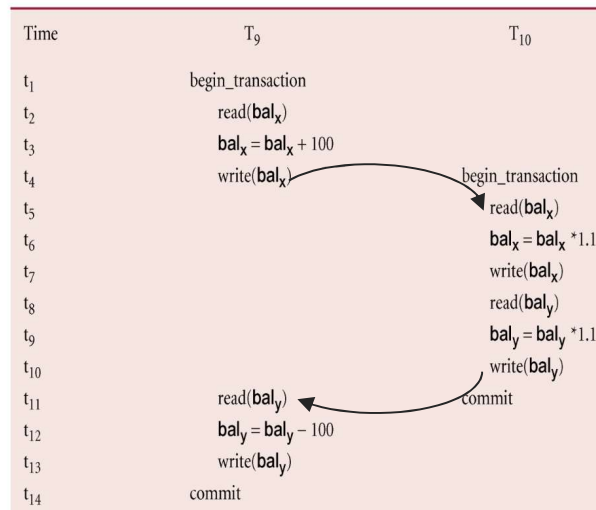
T_1	T_2
read(A) $A := A - 50$ write(A) read(B) $B := B + 50$ write(B) commit	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B) commit $B := B + temp$ write(B) commit

(Slutt: A=50 \$, B=110 \$, A+B=160 \$)

Forløpene produserer ikke samme sluttresultat

Ikke konflikt serielle tidsplaner:

- Kant fra T₉ til T₁₀, når T₁₀ leser verdien til et element skrevet av T₉
- Kant fra T₉ til T₁₀, når T₁₀ skriver en verdi til et element etter at det er lest av T₉
- Kant fra T₉ til T₁₀, når T₁₀ skriver en verdi til et element etter at det er skrevet til av T₉
- Presedens (eller serialiserings) graf inneholder en **cycle**, så planen er derfor ikke konflikt serialiserbar (*not conflict serialiable*)



Øvinger – sjekk med Silberschatz

$A \rightarrow B$, dersom B leser A

$A \rightarrow B$, dersom B skriver etter at A har lest det

$A \rightarrow B$, dersom B skriver etter at A har skrevet det

For hvert av de følgende forløpene, oppgi om tidsplanen er konflikt serialiserbar eller ikke konflikt serialiserbar. Tegn en presedensgraf for hver av forløpene.

- a) read(T1, balx), read(T2, balx), write(T1, balx), write(T2, balx), commit(T1), commit(T2)
- b) read(T1, balx), read(T2, baly), write(T3, balx), read(T2, balx), read(T1, baly), commit(T1), commit(T2)
- c) read(T1, balx), write(T2, balx), write(T1, balx), abort(T2), commit(T1)
- d) write(T1, balx), read(T2, balx), write(T1, balx), commit(T2), abort(T1)
- e) read(T1, balx), write(T2, balx), write(T1, balx), read(T3, balx), commit(T1), commit(T2), commit(T3)

Teknikker for samtidighetskontroll

Teknikker for samtidskontroll

- To grunnleggende samtidighetskontrollteknikker:
 - Låsing,
 - Tidsstempling.
- Begge er konservative tilnærminger: forsink transaksjoner i tilfelle de kommer i konflikt med andre transaksjoner.
- Optimistiske metoder antar at konflikt er sjeldne og bare sjekker for konflikter ved begå.

Låsing

- Transaksjoner bruker låser for å hindre tilgang for andre transaksjoner og dermed forhindre feilaktige oppdateringer.
- Mest brukte tilnærming for å sikre serialiserbarhet.
- Generelt må en transaksjon kreve en **shared (Read)** eller **exclusive (Write)** lås på et dataelement før den leser eller skriver.
- Låser forhindrer at en annen transaksjon endrer eller til og med leser det, i tilfelle en skrivelås.
- En mekanisme som brukes til å kontrollere tilgangen til databaseobjekter



Låsing – Grunnleggende regler

- Dersom en transaksjon har leselås (shared-S lock) på et element så kan den lese det , MEN ikke oppdatere det.
- Dersom transaksjonen har skrivelås (exclusive-X lock) på et element så kan den både lese og oppdatere det .
- Leseoperasjoner kan ikke komme i konflikt → flere transaksjoner kan ha leselås på et og samme element samtidig.
- Skrive lås, eller eksklusiv lås gir transaksjonen eksklusiv tilgang til det elementet

T _B forespørsel:			
		X lock	S lock
T _A har:	X lock	N	N
	S lock	N	Y
	No lock	Y	Y

Y: lås kan bli innvilget
N: lås blir ikke innvilget

Låsing – Grunnleggende regler

- Noen systemer tillater at en transaksjon får oppgradere fra leselås til en eksklusiv lås, eller nedgradere fra eksklusiv lås til en delt lås.
- Dataelementer (eller objekter) av forskjellige størrelser (fra hele databasen, tupel (er) eller kolonne (r) i en relasjon, flere relasjoner, til et felt eller en enkelt dataverdi) kan være låst.
- Jo større dataobjektet som blir låst, jo lavere grad av samtidighet i systemet.

Eksempel: Feil låseplan (1 av 2)

Bruk av låser
garanterer ikke
serialiserbare forløp!

Resultat ved serielt forløp:

$bal_x=220$, $bal_y=330$ dersom T_9
utføres før T_{10}

Resultat av et ikke serielt forløp:

$bal_x=220$, $bal_y=340$ som er
forskjellig fra resultat av det
serielle forløpet.

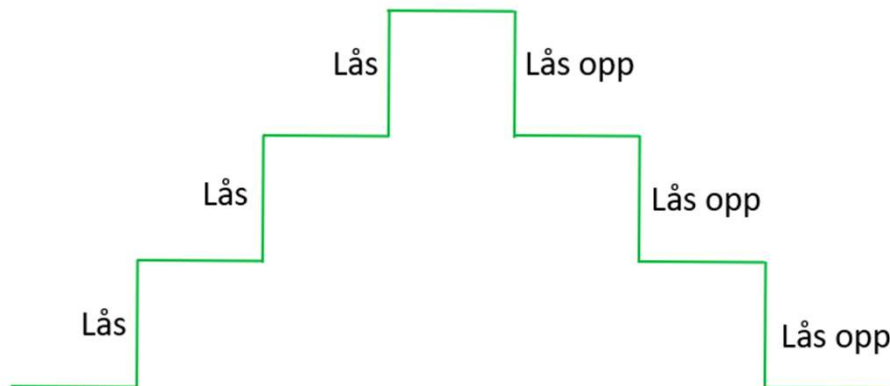
Time	T_9	T_{10}		bal_x	bal_y
t_1	write_lock(bal_x)			100	400
t_2	read(bal_x)			"	"
t_3	$bal_x = bal_x + 100$			"	"
t_4	write(bal_x)			200	"
t_5	unlock(bal_x)			"	"
t_6		write_lock(bal_x)		"	"
t_7		read(bal_x)		"	"
t_8		$bal_x = bal_x * 1.1$		"	"
t_9		write(bal_x)		220	"
t_{10}		unlock(bal_x)		"	"
t_{11}		write_lock(bal_y)		"	"
t_{12}		read(bal_y)		"	"
t_{13}		$bal_y = bal_y * 1.1$		"	"
t_{14}		write(bal_y)		"	440
t_{15}		unlock(bal_y)		"	"
t_{16}		commit		"	"
t_{17}	write_lock(bal_y)			"	"
t_{18}	read(bal_y)			"	"
t_{19}	$bal_y = bal_y - 100$			"	"
t_{20}	write(bal_y)			"	340
t_{21}	unlock(bal_y)			"	"
t_{22}	commit			220	340

Eksempel: Feil låseforløp (2 av 2)

- Problemet er at transaksjoner frigjør låser for tidlig, noe som resulterer i tap av total isolasjon og atomicity.
- For å garantere serialiserbarhet, trenger vi en tilleggsprotokoll angående posisjonering av låse- og låseopphevingsoperasjoner i hver transaksjon.

Løysing - Tofaselåsing

- En transaksjon følger reglene for **tofaselåsing** hvis alle låseoperasjoner gjøres før første frigivelse (opplåsing).



For one object		Lock requested	
		Read	Write
Lock already set	none	OK	OK
	read	OK	wait
	write	Wait	wait

- To faser for transaksjonen:
 - Vekstfase - anskaffer alle låser, men kan ikke frigjøre noen låser.
 - Krympfase - frigjør låser, men kan ikke anskaffe nye låser.
- Følgende gjelder: Hvis alle transaksjoner følger tofaselåsing vil ethvert forløp bli serialiserbart.

Forebygge tapt oppdateringsproblem ved bruk av 2PL

Time	T ₁	T ₂	bal _x
t ₁		begin_transaction	100
t ₂	begin_transaction	write_lock(bal _x)	100
t ₃	write_lock(bal _x)	read(bal _x)	100
t ₄	WAIT	bal _x = bal _x + 100	100
t ₅	WAIT	write(bal _x)	200
t ₆	WAIT	commit/unlock(bal _x)	200
t ₇	read(bal _x)		200
t ₈	bal _x = bal _x - 10		200
t ₉	write(bal _x)		190
t ₁₀	commit/unlock(bal _x)		190

Forebygge uforpliktet avhengighetsproblem ved bruk av 2PL

Time	T ₃	T ₄	bal _x
t ₁		begin_transaction	100
t ₂		write_lock(bal _x)	100
t ₃		read(bal _x)	100
t ₄	begin_transaction	bal _x = bal _x + 100	100
t ₅	write_lock(bal _x)	write(bal _x)	200
t ₆	WAIT	rollback/unlock(bal _x)	100
t ₇	read(bal _x)		100
t ₈	bal _x = bal _x - 10		100
t ₉	write(bal _x)		90
t ₁₀	commit/unlock(bal _x)		90

Forebygge inkonsistent analyse problemet ved bruk av 2PL

Time	T ₅	T ₆	bal _x	bal _y	bal _z	sum
t ₁		begin_transaction	100	50	25	
t ₂	begin_transaction	sum = 0	100	50	25	0
t ₃	write_lock(bal_x)		100	50	25	0
t ₄	read(bal_x)	read_lock(bal_x)	100	50	25	0
t ₅	bal_x = bal_x - 10	WAIT	100	50	25	0
t ₆	write(bal_x)	WAIT	90	50	25	0
t ₇	write_lock(bal_z)	WAIT	90	50	25	0
t ₈	read(bal_z)	WAIT	90	50	25	0
t ₉	bal_z = bal_z + 10	WAIT	90	50	25	0
t ₁₀	write(bal_z)	WAIT	90	50	35	0
t ₁₁	commit/unlock(bal_x , bal_z)	WAIT	90	50	35	0
t ₁₂		read(bal_x)	90	50	35	0
t ₁₃		sum = sum + bal_x	90	50	35	90
t ₁₄		read_lock(bal_y)	90	50	35	90
t ₁₅		read(bal_y)	90	50	35	90
t ₁₆		sum = sum + bal_y	90	50	35	140
t ₁₇		read_lock(bal_z)	90	50	35	140
t ₁₈		read(bal_z)	90	50	35	140
t ₁₉		sum = sum + bal_z	90	50	35	175
t ₂₀		commit/unlock(bal_x , bal_y , bal_z)	90	50	35	175

Øvelser på låser

Vurder databaseskjemaet for universitetsregistrering:

```
Student(snum: integer, sname: string, major: string, level: string, age: integer)
```

```
Class(name: string, meets at: time, room: string, fid: integer)
```

```
Enrolled(snum: integer, cname: string)
```

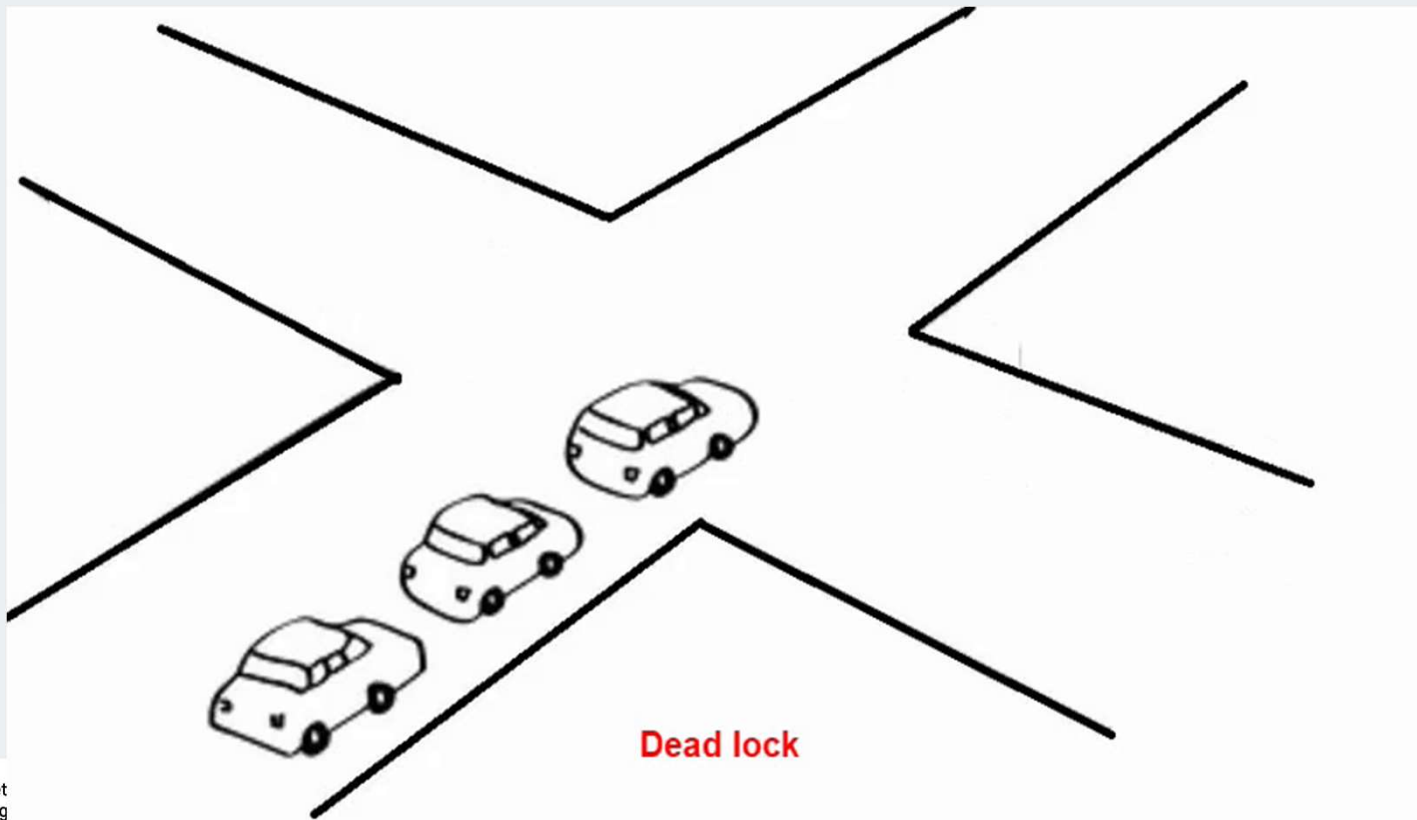
```
Faculty(fid: integer, fname: string, deptid: integer)
```

Betydningen av disse relasjonene er grei; For eksempel har Enrolled en post per student-klasse par slik at studenten er registrert i klassen.

For hver av de følgende transaksjonene, oppgi hvilken type lås du vil bruke og forklar hvorfor du valgte den

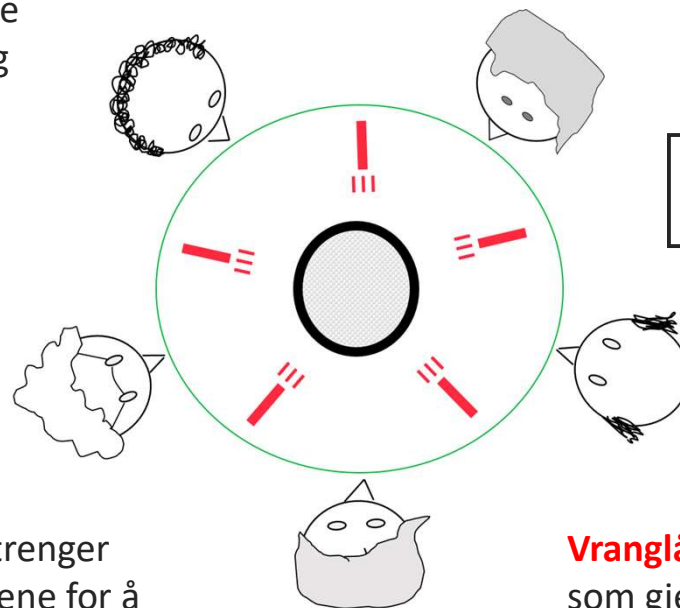
1. Registrer en student identifisert med hennes *snum* i klassen som heter 'Database System 2'.
2. Endre påmelding for en student som er identifisert med hennes *snum* fra en klasse til en annen klasse.
3. Tilordne et nytt fakultetsmedlem identifisert med hennes *fid* til klassen med minst antall studenter.
4. For hver klasse - Vis antall studenter som er registrert i klassen.

Vrangelås - Deadlock



Vranglås: The Dining Philosophers

Filosofene
tenker og
spiser.



Hva skjer hvis alle
vil spise samtidig?

En filosof trenger
begge gaflene for å
spise.

Vranglås: Transaksjoner
som gjensidig venter på at
de andre skal frigi låser.

Deadlock eksempel

En stans som kan oppstå når to (eller flere) transaksjoner hver venter på at låser som den andre blir frigitt.

Time	T ₁₇	T ₁₈
t ₁	begin_transaction	
t ₂	write_lock(bal_x)	begin_transaction
t ₃	read(bal_x)	write_lock(bal_y)
t ₄	bal_x = bal_x - 10	read(bal_y)
t ₅	write(bal_x)	bal_y = bal_y + 100
t ₆	write_lock(bal_y)	write(bal_y)
t ₇	WAIT	write_lock(bal_x)
t ₈	WAIT	WAIT
t ₉	WAIT	WAIT
t ₁₀	⋮	WAIT
t ₁₁	⋮	⋮

Vi kan bestemme oss for å avbryte T₁₈, slik at alle låser som holdes av T₁₈ frigjøres og T₁₇ er i stand til å fortsette driften igjen.

Deadlock løsning

- Bare en måte å bryte vranglåsen: avbryte en eller flere av transaksjonene.
 - Dette innebærer vanligvis å angre alle endringene som er gjort av den eller de avbrutte transaksjonene.
- Deadlock skal være transparent for brukeren, så DBMS bør starte transaksjonen (e) på nytt.
- DBMS kan imidlertid i ikke praksis starte avbrutte transaksjoner på nytt siden det ikke er klar over transaksjonslogikken til tross for at den har kjennskap til transaksjonshistorikken (med mindre det ikke er noen brukerinngang i transaksjonen eller inngangen ikke er en funksjon av databasetilstanden).

Vranglås

- Tre generelle teknikkar for å håndtere vranglås:
 - Tidsavbrudd (Timeouts)
 - Forhindre at vranglås oppstår
 - Oppdaging og gjenoppsettning av vranglåser.

Håndtere vranglås

- **Oppdage** og «løse opp» vranglås
 - Bygge opp en **ventegraf**: Hvis transaksjon T1 må vente på transaksjon T2, så legg til en kant (pil) fra T1 til T2.
 - Av og til: Gå gjennom ventegrafen og sjekk om det har oppstått en **sykel**, for eksempel at T1 venter på T2 som venter på T3 som igjen venter på T1.
 - Velg en av transaksjonene i cyklen. **Avbryt** transaksjonen («rull den tilbake»). Gjennomfør de andre, og start avbrutt transaksjon etterpå.
- **Forhindre** vranglås
 - Alle transaksjoner gis et unikt **tidsstempel**.
 - Hvis en transaksjon vil måtte vente på en **eldre** transaksjon blir den **avbrutt**. Det betyr at yngre transaksjoner aldri vil vente på eldre og dermed kan det ikke oppstå sykler.

Tidsavbrudd

- Transaksjon som ber om låsing vil bare vente i en systemdefinert tidsperiode.
- Hvis lås ikke er tildelt i løpet av denne perioden, vil låseforespørsel få et tidsavbrudd.
- I dette tilfellet antar DBMS at transaksjonen kan være låst, selv om den ikke trenger å vere det, og den avbryter og starter transaksjonen automatisk på nytt.

Forhindre vranglås

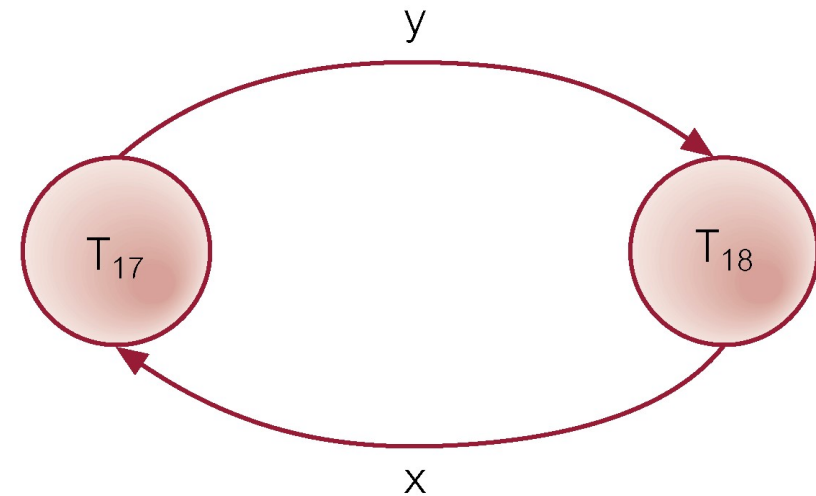
- DBMS ser fremover for å se om transaksjonen vil kunne føre til låsing og aldri tillate at vranglås oppstår.
- Tilordne prioriteringer basert på tidsstempler for transaksjoner.
- Jo lavere tidsstempel jo høyere prioritet
- Hvis en transaksjon vil måtte vente på en eldre transaksjon blir den avbrutt. Det betyr at yngre transaksjoner aldri vil vente på eldre og dermed kan det ikke oppstå sykler.
- Anta at Ti vil ha en lås som Tj holder. To strategier er mulige:
 - **Wait-Die:** Hvis Ti har høyere prioritet, venter Ti på Tj; ellers avbryter Ti (**dies**)
 - **Wound-wait:** Hvis Ti har høyere prioritet, avbryter Tj (**wounded**).; ellers venter Ti
- Hvis en transaksjon starter igjen, må du sørge for at den har sin opprinnelige tidsstempel

Oppdage og «løse opp» vranglås

- DBMS tillater at vranglås oppstår men gjenkjenner dei og bryter dei opp.
- Vanligvis håndtert ved bygging av en **ventegraf** - wait-for graph (WFG) som viser transaksjonsavhengigheter:
 - Opprett en node for hver transaksjon.
 - Hvis transaksjon T1 må vente på transaksjon T2, så legg til en kant (pil) fra T1 til T2.
 - Av og til: Gå gjennom ventegrafen og sjekk om det har oppstått en **sykel**, for eksempel at T1 venter på T2 som venter på T3 som igjen venter på T1. Vranglås oppstår når og berre når vi har en sykkel.
 - Velg en av transaksjonene i cyklen. **Avbryt** transaksjonen («rull den tilbake»). Gjennomfør de andre, og start avbrutt transaksjon etterpå.

Eksempel: Ventegraf (WFG)

Time	T ₁₇	T ₁₈
t ₁	begin_transaction	
t ₂	write_lock(bal_x)	begin_transaction
t ₃	read(bal_x)	write_lock(bal_y)
t ₄	bal_x = bal_x - 10	read(bal_y)
t ₅	write(bal_x)	bal_y = bal_y + 100
t ₆	write_lock(bal_y)	write(bal_y)
t ₇	WAIT	write_lock(bal_x)
t ₈	WAIT	WAIT
t ₉	WAIT	WAIT
t ₁₀	⋮	WAIT
t ₁₁	⋮	⋮



Ekstra oppgave

Teikn ventegrafen for disse transaksjonene

T1	T2	T3	T4
S(A)			
R(A)			
	X(B)		
	W(B)		
S(B)			
		S(C)	
		R(C)	
	X(C)		
			X(B)
		X(A)	

Gjenoppretting fra Deadlock Detection

- Flere omsyn:
 - Valg av “offer”;
 - hvor langt en skal rulle en transaksjon tilbake
 - unngå utsulting (starvation).

Isolasjonsnivåer

- Godta litt «innblanding» av effektivitetshensyn?
- Usikker lesing: T1 kan lese data skrevet av T2 før T2 har skrevet COMMIT.
- Ikke-repeterbar lesing: T1 kan få to forskjellige svar fordi T2 endrer og bekrefter.
- Fantomer: T1 oppdager nye rader satt inn av T2.

Isolasjonsnivå	Fantomer	Ikke-repeterbar lesing	Usikker lesing
SERIALIZABLE	Nei	Nei	Nei
REPEATABLE READ	Ja	Nei	Nei
READ COMMITTED	Ja	Ja	Nei
READ UNCOMMITTED	Ja	Ja	Ja

Pessimistisk og optimistisk låsing

- Pessimistisk låsing = standard låsing.
- **Optimistisk** låsing:
 - Hensiktsmessig i systemer med få konflikter, for eksempel hvis de fleste kun leser, og i «interaktive» databaseapplikasjoner.
 - Transaksjoner blir utført uten restriksjoner fram til COMMIT, men skriver til lokal kopi.
 - Validering før lokal kopi blir skrevet til databasen.
- Kan velge mellom optimistisk/pessimistisk låsing i Access.

Oppsummering

- En transaksjon er en logisk operasjon.
 - Blir definert ved COMMIT og ROLLBACK.
 - Skal tilfredsstille ACID-egenskapene.
- DBHS må håndtere feil og samtidighet.
- Feil:
 - Hvilke transaksjoner ble avbrutt når?
 - Strømbrydd: Transaksjonslogg.
 - Diskkrasj: Sikkerhetskopi + transaksjonslogg.
- Samtidighet:
 - Transaksjoner må ikke få lov til å «forstyrre» hverandre.
 - DBHS bruker leselåser og skrivelåser.
 - Låser alene sikrer ikke et korrekt resultat.
 - Tofaselåsing sikrer serialiserbare forløp.
 - Kan fremdeles få vranglås.
 - Vranglås kan oppdages eller forhindres.