

# **Transaksjoner, seksjon 14-17**

Svein Erik Bratsberg, IDI/NTNU

Versjon 24.april 2023

# Innhold



- Seksjon 14 og 15 (pensumnotat)
  - Hvorfor transaksjoner?
  - Transaksjoner og SQL
  - Transaksjonsteori
- Seksjon 16
  - Flerbrukerkontroll (CC)
  - Korrekthet
  - Låsing
  - SNAPSHOT ISOLATION
- Seksjon 17
  - Logging og recovery
  - Abortering av transaksjoner
  - Krasjrecovery

# Hvorfor transaksjoner?



- Støtter deling og samtidig aksess av data
  - Flerbrukerkontroll. READ COMMITTED, SNAPSHOT ISOLATION, SERIALIZABLE
- Støtter sikker, pålitelig, atomisk aksess til store mengder data
  - Recovery: Rollback og krasjrecovery

# Databaseoperasjoner



- $X$  – databaseobjekt: post eller blokk
  - $\text{read}(X)$
  - $r(X)$
  - $\text{write}(X)$
  - $w(X)$
- Tilhørende transaksjon 1
  - $\text{read}_1(X)$
  - $r_1(X)$
- Commit1     $c_1$     suksess: avslutting av transaksjon 1
- Abort1         $a_1$     abortering av transaksjon 1

# Samtidighetsproblemer, eksempel

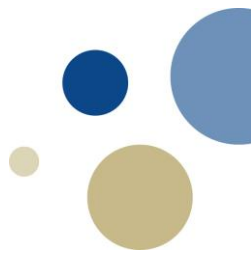
- To transaksjoner

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.06*A,	B=1.06*B	END

T1:	A=A+100,	B=B-100
T2:	A=1.06*A,	B=1.06*B

T1:	A=A+100,	B=B-100
T2:	A=1.06*A, B=1.06*B	

# Samtidighetsproblemer (2)



T1:	R(A), W(A),	R(B), W(B)
T2:	R(A), W(A), R(B), W(B)	

- $r_1(A); w_1(A); r_2(A); w_2(A); r_2(B); w_2(B); r_1(B); w_1(B);$

# Samtidighetsproblemer, klasser

- Dirty read

T1:	R(A), W(A),	R(B), W(B), Abort
T2:	R(A), W(A), C	

- Lost update / dirty write

T1:	W(A),	W(B), C
T2:	W(A), W(B), C	

# Samtidighetsproblem, klasser (2)

- Example: Lost update / dirty write
  - $w_1(\text{buyer}=\text{'Alice'}); w_2(\text{buyer}=\text{'Bob'}); w_2(\text{invoice}=\text{'Bob'}); w_1(\text{invoice}=\text{'Alice'});$
- Unrepeatable read

T1:	R(A),	R(A), W(A), C
T2:	R(A), W(A), C	

- Incorrect summery
  - En transaksjon beregner en aggregatfunksjon mens en annen gjør en oppdatering
  - Figur 20.3 (c)



# Incorrect summary

(c)

$T_1$	$T_3$
<pre>read_item(X); X := X - N; write_item(X);  read_item(Y); Y := Y + N; write_item(Y);</pre>	<pre>sum := 0; read_item(A); sum := sum + A; ⋮  read_item(X); sum := sum + X; read_item(Y); sum := sum + Y;</pre>

←  $T_3$  reads  $X$  after  $N$  is subtracted and reads  $Y$  before  $N$  is added; a wrong summary is the result (off by  $N$ ).

# Oppgave



- Se på følgende historier:
- **H1**:  $r_1(A)$ ;  $w_1(A)$ ;  $r_1(B)$ ;  $w_2(A)$ ;  $w_2(B)$ ;  $w_1(B)$ ;  $c_1$ ;  $c_2$ ;
- **H2**:  $r_1(A)$ ;  $w_1(A)$ ;  $r_2(A)$ ;  $w_2(A)$ ;  $c_2$ ;  $r_1(B)$ ;  $a_1$ ;
- Hva er problemene for H1 og H2?
  - Unrepeatable read / dirty write
  - Dirty read
  - Lost update

# Hvorfor trenger vi recovery?



- To typer recovery (gjenoppretting)
- En transaksjon ruller tilbake (aborteres)
  - Uventet situasjon
  - Manglende data
  - Brukeren bestemmer det
  - Samtidighetskontrollen bestemmer det (CC)
- Systemkrasjrecovery
  - Databasesystemet, OS'et eller datamaskinen krever en restart

# ACID – egenskaper ved en transaksjon



Transaksjon: en gruppering av operasjoner mot databasen som er

- **A** – atomiske: enten kjører de fullstendig, eller så kjører de ikke
- **C** – consistency: overholder konsistenskrav (primary key, references, check, osv)
- **I** – isolation: som er isolert fra hverandre. Merker ikke at noen kjører samtidig.
- **D** – durability: er permanente, dvs. mistes ikke etter commit.

En transaksjon er vanligvis en logisk operasjon eller oppgave

# Eksempler på transaksjoner



- En gruppering av operasjoner mot databasen
- Banktransaksjon
- Kjøp en flyreise
- Tegn en polylinje
- Fyll ut et skjema
- Lever en eksamen
- Setter inn poster som har indekselementer som også må oppdateres
- .....

# Commit/Abort



- En transaksjon slutter med
- **COMMIT**: Alt gikk bra og endringene fra transaksjonen finnes i databasen. `Connection.commit()`;
- **ROLLBACK** (abort): Transaksjonen rulles tilbake (aborteres) og ingen endringer fra transaksjonen finnes i databasen. `Connection.rollback()`;
- **Autocommit**: Hver SQL-setning er en egen transaksjon. Kan skrues på. Default av i Python/SQLite3-API. Settes via `isolation_level` i `connection`-objektet.

# Commit/abort (2)

- SET AUTOCOMMIT=0;  
UPDATE Account SET b = b - 1000 WHERE id=123123;  
UPDATE Account SET b = b + 1000 WHERE id=234234;  
COMMIT;
- Ekt-eksempel RegMålCtrl  
INSERT INTO Reg VALUES (1,123123,31,100);  
INSERT INTO Reg VALUES (2,123123,32,120);  
....  
INSERT INTO Reg VALUES (9,123123,175,245);  
UPDATE Loper SET status = 'ok'  
WHERE brikkenr=123123;  
COMMIT;

# SQLs isolasjonsnivå



- SET TRANSACTION ISOLATION LEVEL  
    READ UNCOMMITTED  
    READ COMMITTED  
    REPEATABLE READ  
    SERIALIZABLE (default)
- Mer isolasjon/ «korrekthet» nedover
- Mindre samtidighet nedover
- Egenskaper vi vill unngå:
  - Dirty read
  - Unrepeatable read
  - Unngå fantomer: Hvis T leser en mengde verdier basert på en søkebetingelse, så vil ikke denne mengden endres av andre før T er ferdig. Aktuelt ved reskaning (nested loop f.eks)



# SQLs isolasjonsnivå (2)



Isolation Level	Dirty Read	Unrepeatable Read	Phantom Problem
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Reads	No	No	Maybe
Serializable	No	No	No

# Quiz 1



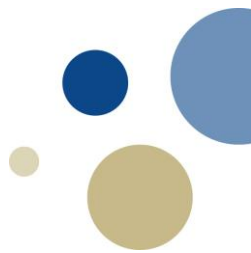
- Hvorfor må en transaksjon rulles tilbake?
  1. Tabellen er ikke på 1. normalform
  2. Samtidighetskontrollen bestemte det
  3. Pga. dirty read
  4. Brukeren angret seg
- ACID betyr
  1. Atomicity, Concurrency, Isolation, Durability
  2. Atomicity, Consistency, Isolation, Database
  3. Attribute, Consistency, Isolation, Durability
  4. Atomicity, Consistency, Isolation, Durability

# Quiz 2



- Hva er AUTOCOMMIT?
  1. Databasen har en konsistent tilstand
  2. SQL commiter når loggen er full
  3. Hver SQL-setning er en egen transaksjon
  4. Du har bestemt deg for bilkjøpet
- Hva er problemet med SERIALIZABLE?
  1. Du får mye dirty read
  2. Tillater lite samtidighet
  3. Svak isolasjon
  4. Vanskelig å få til DURABILITY

# READ COMMITTED



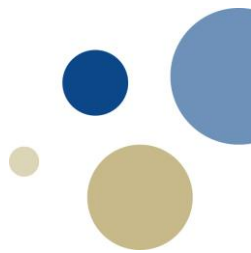
1. Når du leser fra databasen, vil du kun se data som er committed (ingen *dirty reads*).
2. Når du skriver til databasen, skriver du kun over data som er committed (ingen *dirty writes*).

READ COMMITTED er default i Oracle, MS SQL Server og PostgreSQL.

To metoder brukes for å støtte dette.

1. *Låsing*. Transaksjonene setter skriverlåser før de skriver data. Låsene slippes ved commit. Før de leser et element, så setter transaksjonen *leselås*, og slipper denne etter at dataelementet er lest.
2. *Snapshot isolation*. Mange databaser hindrer dirty reads ved å beholde gamle verdier for data ved skrijving inntil commit av transaksjonen. Read-transaksjoner kan lese den gamle verdien. Når den nye verdien er committed, kan andre transaksjoner ta den i bruk. Å ha leselåser vil holde alle writes unna, så det gjør man ikke. Dette kalles også for *multi-version concurrency control*. Mer om dette senere.

# SNAPSHOT ISOLATION



- Snapshot isolation brukes som løsning på “unrepeatable read” (“read skew”).
- Brukes av PostgreSQL, MySQL/InnoDB, Oracle, SQL Server, og andre
- Bruker write locks, men reads krever ikke låser
- Readers blokkerer aldri skrivere, skrivere blokkerer aldri lesere
- Multiversion concurrency control (MVCC). Lagrer flere versjoner av oppdaterte rader. En for hver snapshot.

# Transaksjonshistorie

- **Historie** (schedule)

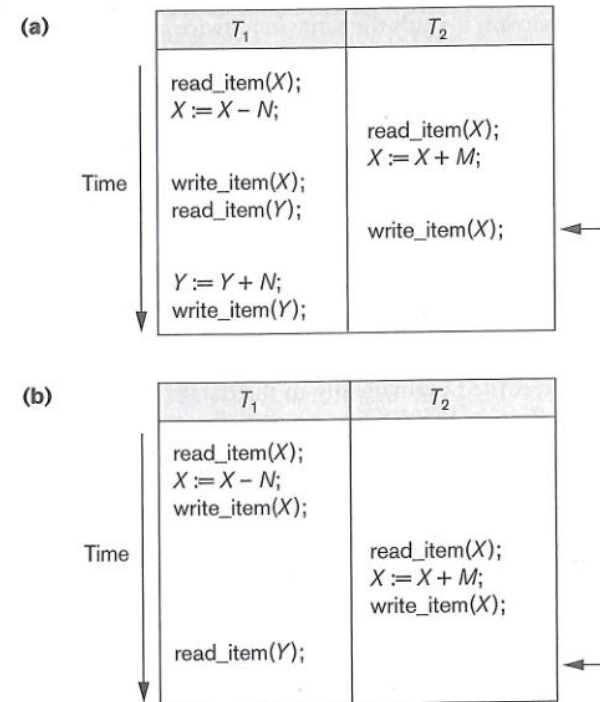
Liste av aksjoner

read, **w**rite, **a**bort, **c**ommit  
for en mengde transaksjoner

- Fra figur 20.3 a) og b):

$H_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$

$H_b: r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); a_1;$



# Transaksjonshistorie - konflikt



- **Konflikt:**  
To operasjoner fra en historie er i konflikt hvis
- (1) de tilhører forskjellige transaksjoner
- (2) de bruker samme dataelement
- (3) minst en av operasjonene er en write
  - $H_a: r_1(X)$  og  $w_2(X)$  er i *konflikt*
  - $H_a: w_1(X)$  og  $w_2(X)$  er i *konflikt*
  - $H_a: r_1(X)$  og  $r_2(X)$  er *ikke i konflikt*
- To operasjoner er i konflikt hvis endring av rekkefølgen endrer resultatet på databasen

# Transaksjoner og gjenopprettbarhet

- **Gjenopprettbar historie** (recoverable schedule):  
Hver transaksjon committer etter at transaksjoner de har lest fra har committet.  
 $H_1: w_2(A); w_1(B); w_1(A); r_2(B); c_1; c_2;$
- Historier som unngår galopperende abort (avoid cascading abort – **ACA**):  
Når transaksjoner kun kan lese verdier skrevet av committede transaksjoner.  
 $H_1$  er ikke ACA.  
 $H_2: w_1(A); w_1(B); w_2(A); c_1; r_2(B); c_2;$



# Historier og gjenopprettbarhet (2)

- **Strikt historie:**

Når transaksjonene verken kan lese eller skrive ikke-committede verdier

$H_3: w_1(A); r_1(B); w_2(B); c_1; w_2(A); c_2;$

- Kan gjøre *undo recovery* ved *before image* fra loggen

- Sammenheng:

*Strikt  $\subset$  ACA  $\subset$  Gjennopprettbar  $\subset$  Alle historier*

# Oppgaver

$H_1$  :  $r_1(A); w_1(A); r_2(A); w_2(A); C_2; C_1$

$H_2$  :  $r_1(A); w_1(A); C_1; r_2(A); w_2(A); C_2$

$H_3$  :  $r_1(A); w_1(A); r_2(A); w_2(A); C_1; C_2$

$H_4$  :  $r_2(A); w_2(B); w_1(B); C_2; r_1(A); C_1$

Ikke gjenopprettbar

Gjenopprettbar : Hver trans. committer etter trans. de har lest fra committer

ACA: En trans. kan kun lese committede verdier.

Strict: En trans kan verken lese eller skrive ikke-committede verdier.

# Historier og serialiserbarhet

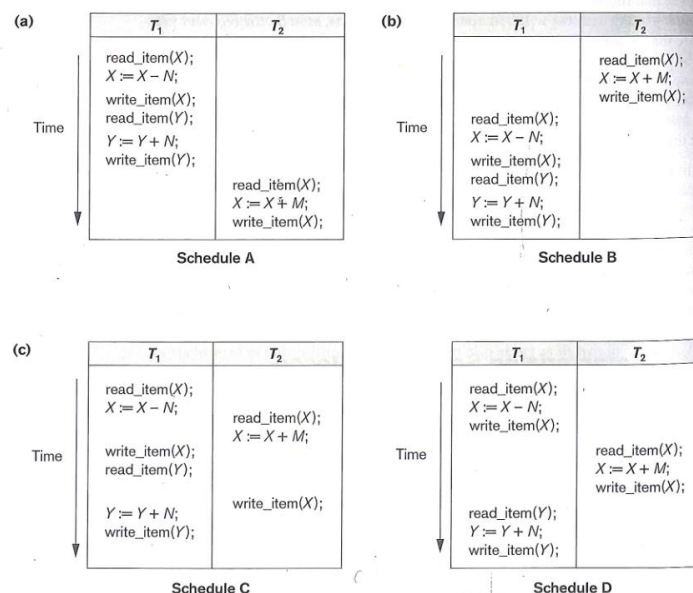
- **Seriell historie**

Historie som ikke fletter operasjoner fra forskjellige transaksjoner. Kjører etter hverandre

- **Serialiserbar historie**

Historie som har samme effekt på databasen som en seriell historie (resultatekvivalent)

- Figur 20.5



# Historier og serialiserbarhet (2)



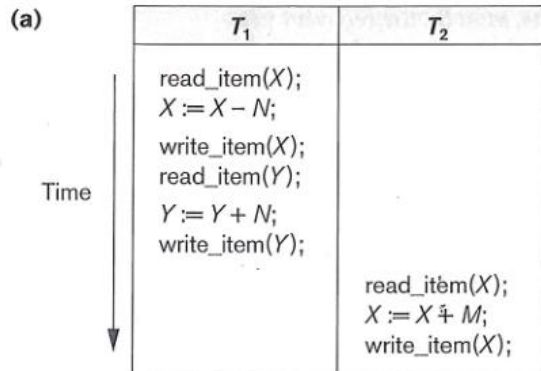
- Vi ønsker *serialiserbare* og ikke kreve serielle historier fordi vi ønsker samtidighet
  1. Parallele tråder
  2. Diskaksess – andre tråder kan jobbe så lenge

# Konfliktserialiserbarhet

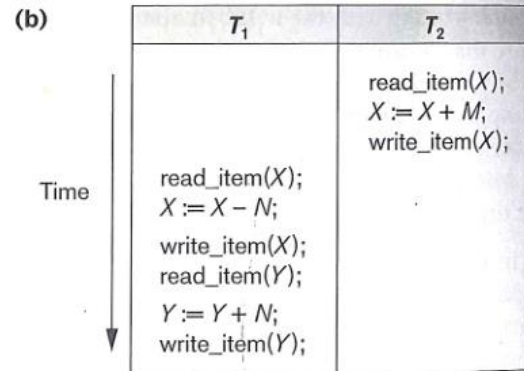


- Konflikt mellom to operasjoner
  - $r_1(A)$  og  $w_2(A)$
  - $w_1(A)$  og  $r_2(A)$
  - $w_1(A)$  og  $w_2(A)$
- To historier er *konfliktekvivalente* hvis de har samme rekkefølge for operasjoner med konflikt
- En historie er *konfliktserialiserbar* hvis den er konfliktekvivalent med en seriell historie
- Konfliktserialiserbarhet impliserer serialiserbarhet, men ikke nødvendigvis motsatt
- Figur 20.5 c) og d)

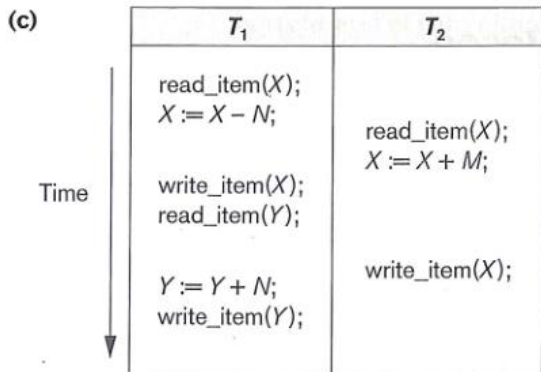
# Konfliktserialiserbarhet



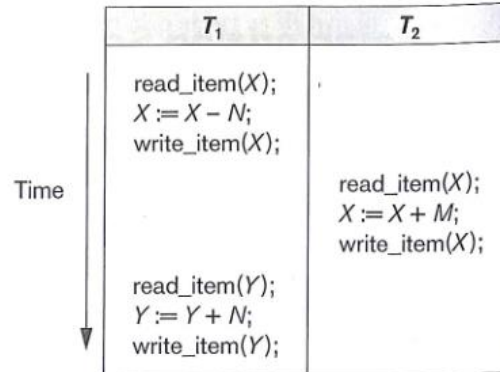
Schedule A



Schedule B



Schedule C



Schedule D

# Presedensgraf



- Rettet graf
- *Noder*: transaksjoner i historie H
- *Kanter*:  $T_1 \rightarrow T_2$  finnes når det finnes en operasjon i  $T_1$  som er i konflikt med en operasjon i  $T_2$ , og  $T_1$ s operasjon skjer før  $T_2$ s operasjon
- Hvis en presendensgraf ikke har *sykler*, er historien **konfliktserialiserbar**
- $H_1$ :  $r_2(A)$ ;  $r_1(B)$ ;  $w_2(A)$ ;  $r_3(A)$ ;  $w_1(B)$ ;  $w_3(A)$ ;  $r_2(B)$ ;  $w_2(B)$ ;
- $H_2$ :  $r_2(A)$ ;  $r_1(B)$ ;  $w_2(A)$ ;  $r_2(B)$ ;  $r_3(A)$ ;  $w_1(B)$ ;  $w_3(A)$ ;  $w_2(B)$ ;

# Serialiserbarhet ved låsing



- Bruker låser av dataelement (poster eller blokker) for å garantere konfliktserialiserbarhet
- Låsetyper
  - **Read\_lock (X)** (delt lås)
  - **Write\_lock (X)** (eksklusiv lås)
- Flere transaksjoner kan ha read\_lock (delt lås) på samme dataelement samtidig.
- Det er også mulig med oppgradering og nedgradering av låser.
  - Read\_lock -> Write\_lock
  - Write\_lock -> Read\_lock



# Implementasjon av låser

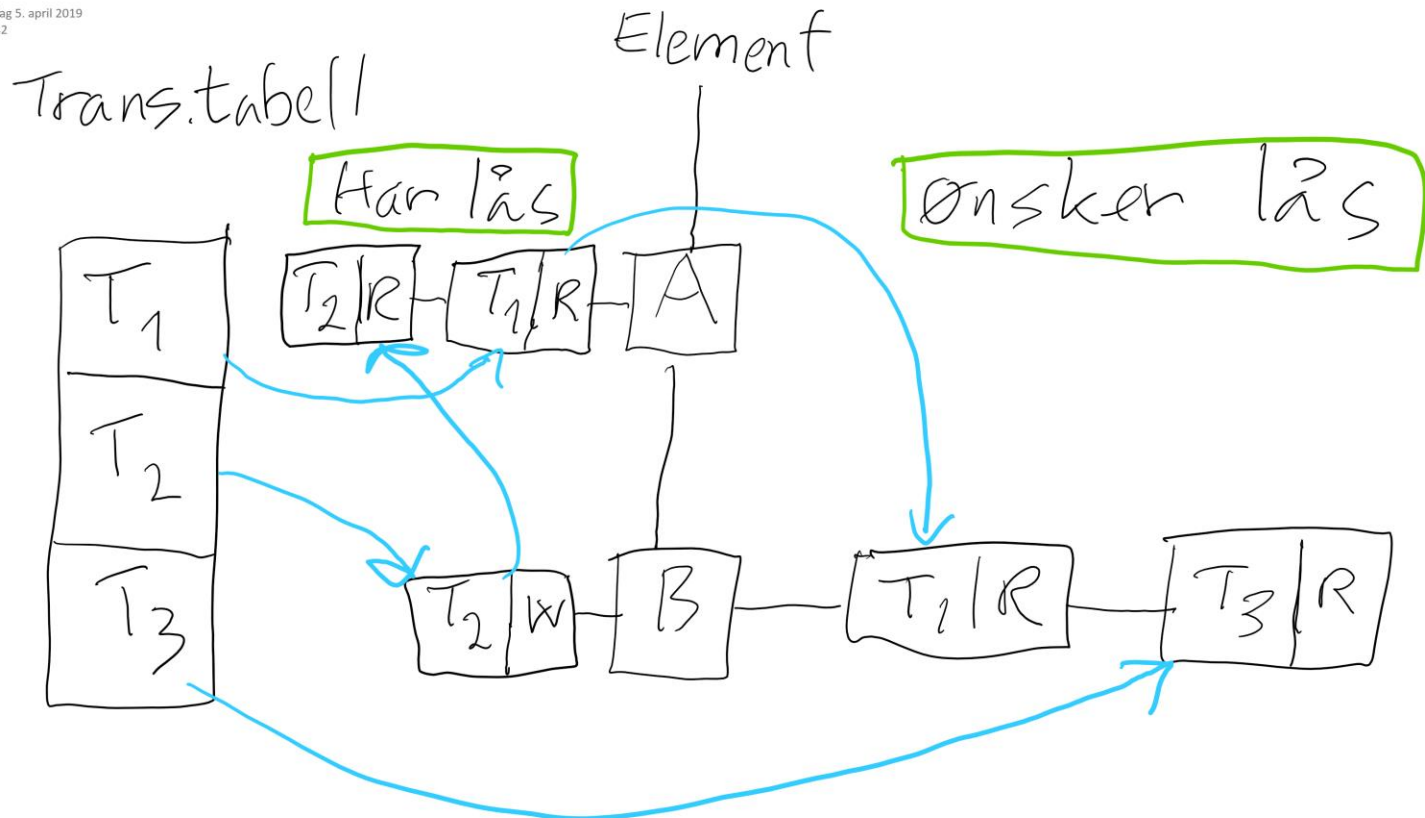


- Låsetabell i minne
  - Postlåser
  - Blokklåser
  - Tabelllåser
  - Verdiområdelåser (unngå fantomer)
  - Predikatlåser (unngå fantomer)
- Eksempel:
  - $w_2(B); r_1(A); r_2(A); r_1(B); r_3(B);$

# Låseimplementasjon

- $w_2(B); r_1(A); r_2(A); r_1(B); r_3(B);$

fredag 5. april 2019  
12:42



# 2PL – tofaselåsing (two-phase locking)

- En transaksjon har tofaselåsing hvis alle låseoperasjoner skjer før alle opplåsingsoperasjoner

T1	T2
	Write_lock(X)
Write_lock(X)	Read(X)
wait	$X = X + 1000$
wait	Write(X)
wait	Commit / Unlock(X)
Read(X)	
$X = X - 100$	
Write(X)	
Commit / Unlock(X)	

# 2PL og «incorrect summary»

T1	T2
Write_lock(X)	Sum = 0
Read(X)	Read_lock(X)
X = X - 100	Wait
Write(X)	Wait
Write_lock(Y)	Wait
Read(Y)	Wait
Y = Y + 100	Wait
Write(Y)	Wait
Commit / Unlock (X, Y)	Wait
	Read(X)
	Sum = Sum + X
	Read_lock(Y)
	Read(Y)
	Sum = Sum + Y
	Commit / Unlock (X,Y)

~~761~~ 787

(a)

$T_1$	$T_2$
read_lock(Y); read_item(Y); unlock(Y); write_lock(X); read_item(X); $X := X + Y$ ; write_item(X); unlock(X);	read_lock(X); read_item(X); unlock(X); write_lock(Y); read_item(Y); $Y := X + Y$ ; write_item(Y); unlock(Y);

(b) Initial values:  $X=20$ ,  $Y=30$

Result serial schedule  $T_1$   
followed by  $T_2$ :  $X=50, Y=80$

Result of serial schedule  $T_2$   
followed by  $T_1$ :  $X=70, Y=50$

(c)

	$T_1$	$T_2$
Time ↓	read_lock(Y); read_item(Y); unlock(Y);	read_lock(X); read_item(X); unlock(X); write_lock(Y); read_item(Y); $Y := X + Y$ ; write_item(Y); unlock(Y);
	write_lock(X); read_item(X); $X := X + Y$ ; write_item(X); unlock(X);	

Result of schedule  $S$ :  
 $X=50, Y=50$   
 (nonserializable)

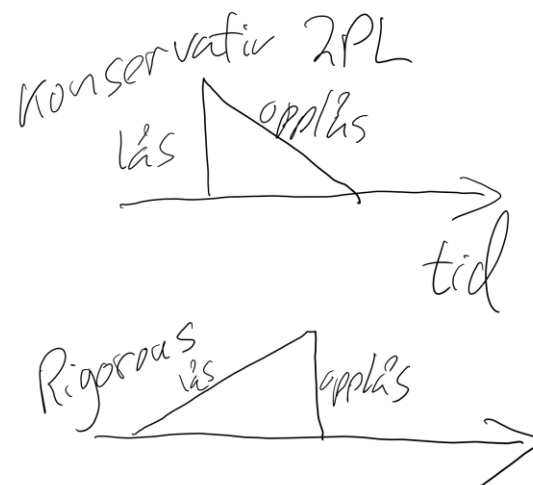
**Figure 21.3**

Transactions that do not obey two-phase locking. (a) Two transactions  $T_1$  and  $T_2$ . (b) Results of possible serial schedules of  $T_1$  and  $T_2$ . (c) A nonserializable schedule  $S$  that uses locks.

# 2PL-modeller

- Tofaselåsing impliserer serialiserbarhet
- **Basic 2PL:** «Symmetrisk fjell»
- **Konservativ 2PL:** Låser alt man trenger aller først
- **Strict 2PL:** Opplåsing av skrivelåser etter commit/abort
- **Rigorous 2PL:** Opplåsing etter commit/abort

fredag 5. april 2019  
14:31



# Vranglås



- To eller flere transaksjoner venter gjensidig på hverandres låser

T1	T2
Read_lock(X)	Read_lock(Y)
Write_lock(Y)	Write_lock(X)

- Kan løses ved forskjellige metoder
  - Unngåelse
  - Oppdagelse
  - Timeout

# Vranglåsoppdagelse



- Den vanligste løsningen
- Konstruer wait-for-grafen:
  - Hver transaksjon er en node
  - Hvis  $T_i$  venter på en lås holdt av  $T_j$ , får vi en rettet kant  $T_i \rightarrow T_j$
- Vi har vranglås hvis grafen har sykler
- Prøv å abortere en transaksjon og se om sykkelen forsvinner



# Timeout



- Den enkleste løsningen
- La hver transaksjon ha en timeout.
- Hvis timeouten går, aborter transaksjonen
- Vanskelig å sette timeouten riktig

# Rigorous 2PL eksempel



- $H_1: r_1(A); w_2(A); w_2(B); w_3(B); w_1(B); C_1; C_2; C_3;$
- $H_2: r_1(A); w_2(B); w_2(A); w_3(B); w_1(B); C_1; C_2; C_3;$
- For *låsing*: Hvis en transaksjon blir blokkert, blir alle operasjoner i transaksjonen satt på vent, mens de neste operasjonene i historien blir utført i sekvens.

# Multiversjons-CC



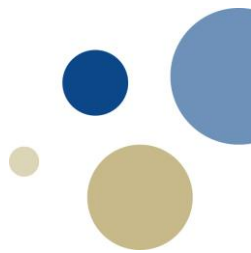
- CC = Concurrency Control
- Brukes mye i dagens SQL-databaser
- La en leseoperasjon som er i konflikt, lese en gammel versjon.
- Basert på tidsstempelordning (timestamp ordering):
  - Hver transaksjon har et tidsstempel  $TS(T)$
  - Hvert dataelement kan finnes i flere versjoner  $(X_1, X_2, \dots, X_k)$
  - For hver versjon  $X_i$  lagres
    - $read\_TS(X_i)$ : Største tidsstempel for en trans som har lest den
    - $write\_TS(X_i)$ : Tidsstempel for transaksjon som skrev  $X_i$ .
  - Når en transaksjon skriver  $X_i$ , settes begge til  $TS(T)$

# Multiversjons-CC (2)



- Regler for å sikre serialiserbarhet
  1. T ønsker å utføre **write(X)**
    - Hvis  $\text{write\_TS}(X_i) \leq \text{TS}(T)$  AND  $\text{read\_TS}(X_i) > \text{TS}(T)$ 
      - Rull tilbake T
    - Ellers
      - Lag ny versjon  $X_i$  med  $\text{read\_TS}(X_i) = \text{write\_TS}(X_i) = \text{TS}(T)$
  2. T ønsker å utføre **read(X)**
    - Finn i som har høyest  $\text{write\_TS}(X_i) \leq \text{TS}(T)$
    - $\text{read\_TS}(X_i) = \text{Max}(\text{TS}(T), \text{read\_TS}(X_i))$
    - Returner  $X_i$
- Sikrer alltid suksess ved lesing
- Ulempe: Administrasjon av mange versjoner, mer plass, ikke update-in-place?

# Multiversjons-CC (3)



- To måter i praksis
  1. Lagrer flere versjoner av poster i databasen og kjører GC (søppeltømming) når de gamle versjonene ikke trenges lengre: Microsoft SQL, PostgreSQL, MySQL InnoDB (consistent reads).  
<https://dev.mysql.com/doc/refman/8.0/en/innodb-consistent-read.html>
  2. Lagrer kun siste versjon av posten, men kan konstruere den forrige versjonen vha. undo: Oracle
- Noen systemer bruker både låsing og multiversjons-CC, dvs. SNAPSHOT ISOLATION
  1. Write-operasjoner bruker låser og 2PL
  2. Read-operasjoner bruker multiversjons-CC

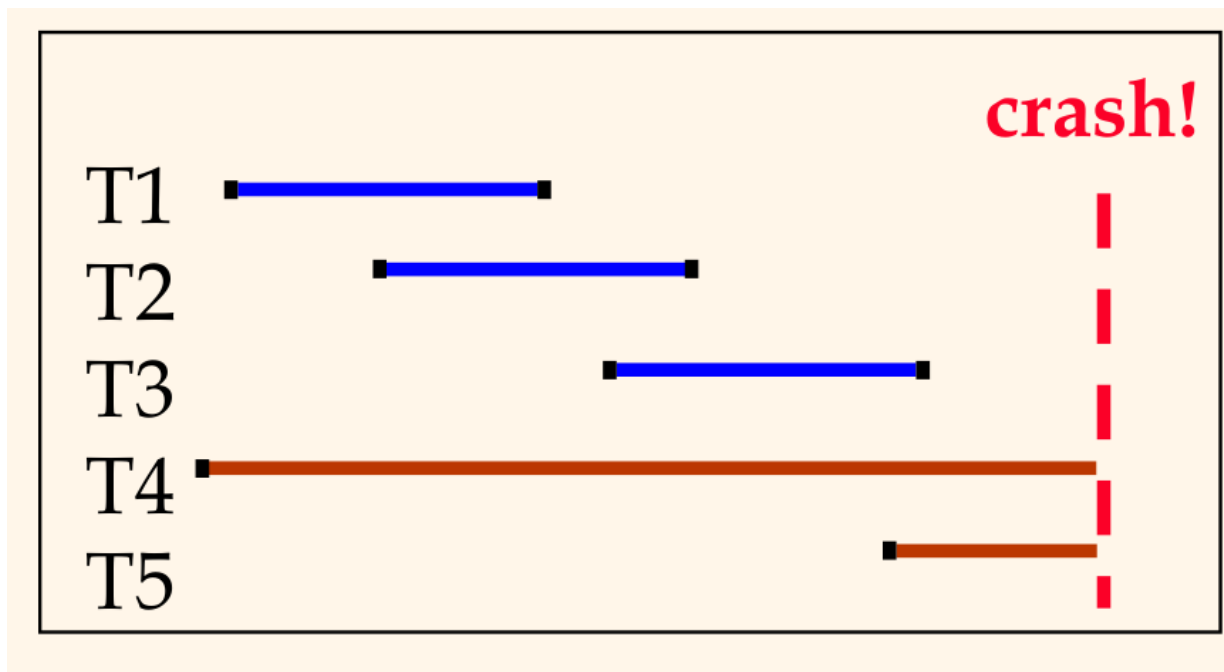
# Recovery



- Databasesystemet støtter sikker, atomisk aksess til store mengder data
- Transaksjonene er
  - A – atomiske: Enten har de kjørt helt, eller overhodet ikke
  - C
  - I
  - D – durability: Er permanente. Etter commit mistes ikke data.

# Transaksjoner etter krasjrecovery

- Vinnere: T1, T2 og T3 skal være permanente.
- Tapere: T4 og T5. Må aborteres. Hvorfor?



# Force/steal-klassifisering av Logging & Recovery-algoritmer

- Utgangspunkt: Hvor fleksibel (uavhengig) er *buffer manager* til logging/recovery
  - Når kan skitne (dirty) blokker skrives?
  - Når må skitne blokker skrives?
- **Force:** Må en skitten (oppdatert) blokk tvinges til disk ved commit.
  - Tregt: datablokkene kan være spredd over hele disken
- **Steal:** Kan en transaksjon stjele plassen i bufferet til en skitten blokk?
  - Hvis ikke, må en aktiv transaksjon ha alle skitne blokker i buffer inntil commit.



# Force/Steal (2)

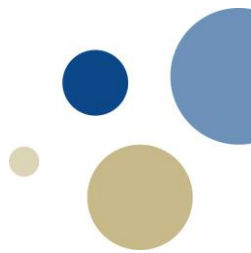
	No steal	Steal
Force	Shadowing (ikke logging)	Undo-logging No-redo
No-force	Redo-logging No-undo	Undo/re-do-logging Aries

# Write-ahead logging (WAL)



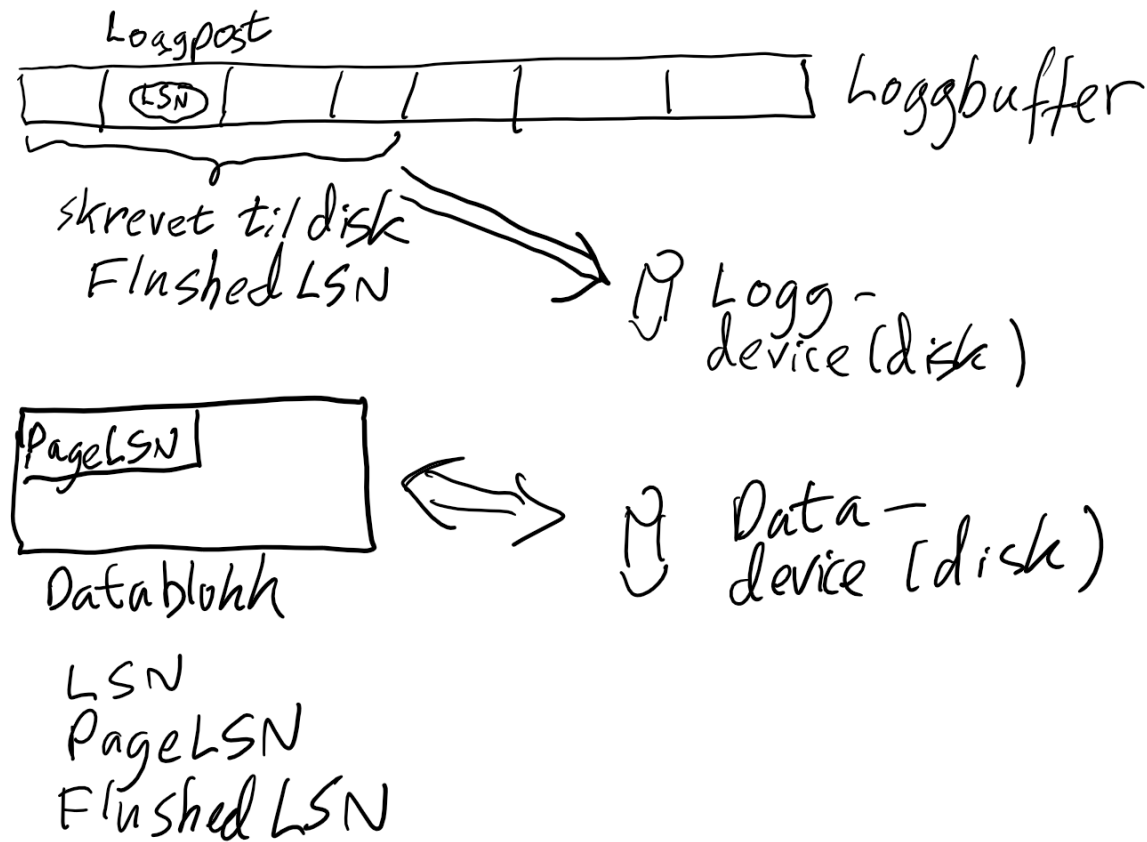
- Basis for undo/redo-logging
- Hver endring (insert/delete/update) har en loggpost i loggen.
- Regler:
  - Skriv en loggpost som endret en datablokk til disk før du skriver datablokken (for undoformål)
  - Skriv loggen til disk før en transaksjon committer (for redoformål)  
«Force log at commit»

# WAL-konsepter i ARIES

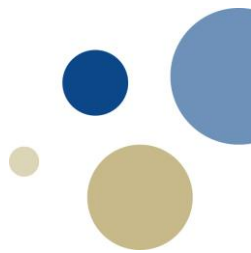


- **LSN** – loggsekvensnummer. ID for loggpost. Stigende nr.
- **PageLSN** – LSN til loggpost som sist endret en blokk  
Lagret i hver blokk.
- **FlushedLSN** – LSN til nyeste skrevne loggpost til disk
- Ved skriving av datablokk til disk, sjekk  
 $\text{PageLSN} < \text{FlushedLSN}$
- Hvis ikke, skriv (flush) logg først.

# LSN-begreper (log sequence number)



# Loggpost i ARIES



LSN	TransID	PrevLSN	OpType	PageId	Offset	BeforeImage	AfterImage
-----	---------	---------	--------	--------	--------	-------------	------------

- *PrevLSN*: Peker til forrige loggpost i samme transaksjon. For abortering av transaksjon.
- *OpType*: Update/insert/delete
- *PageId*: Hvilken blokk ble endret (BlokkId)
- *Offset*: Hvor i blokken ble det endret?
- *BeforeImage*: Verdi før endring
- *AfterImage*: Verdi etter endring

# Datastruktur for recovery (ARIES)



- **Transaksjonstabell**
  - Et element per aktiv transaksjon
    - TransId
    - Tilstand: aktiv, committed, aborting, aborted
    - LastLSN: Peker til nyeste loggpost i transaksjonen
  - Transaksjonen blir borte når transaksjonen committer
- **Dirty page table (DPT)**
  - Et element per skitten (dirty) blokk i buffer
    - PageID
    - RecLSN: Peker til eldste loggpost som gjorde blokken skitten
  - Elementet blir borte når «disk callback» fra write blir kjørt

# Sjekkpunkting



- Periodisk lager DBMSet et sjekkpunkt i loggen som skal minimalisere tiden det tar å gjøre recovery
- Du slipper å skanne hele loggen ved recovery
  - Begin checkpoint
    - Lag start sjekkpunkt-loggpost
  - End checkpoint
    - Lag slutte sjekkpunkt-loggpost som inneholder
      - Transtabell
      - DPT – dirty page table
  - Lagre LSN til sjekkpunktloggpost på sikkert sted. Logganker
- I noen systemer er sjekkpunkting koblet til det å skrive skitne blokker til disk (ikke ARIES)
- Penultimate checkpointing: Skriv data ved sjekkpunkting. Bruk nest siste sjekkpunkt som startpunkt.
- Fuzzy checkpointing: Tillat samtidige transaksjoner

# Abortering av transaksjon



- Finn LastLSN fra transaksjonstabellen
- For hver loggpost i transaksjonen (bakover)
  - Lag CLR – kompenserende loggpost, som gjør det motsatte av loggposten (non-CLR)
  - Gjør REDO av CLRen
- Fjern transaksjonen fra transaksjonstabellen
- CLRen er grunnlag for låser på radnivå (mer presise enn låser på blokker)

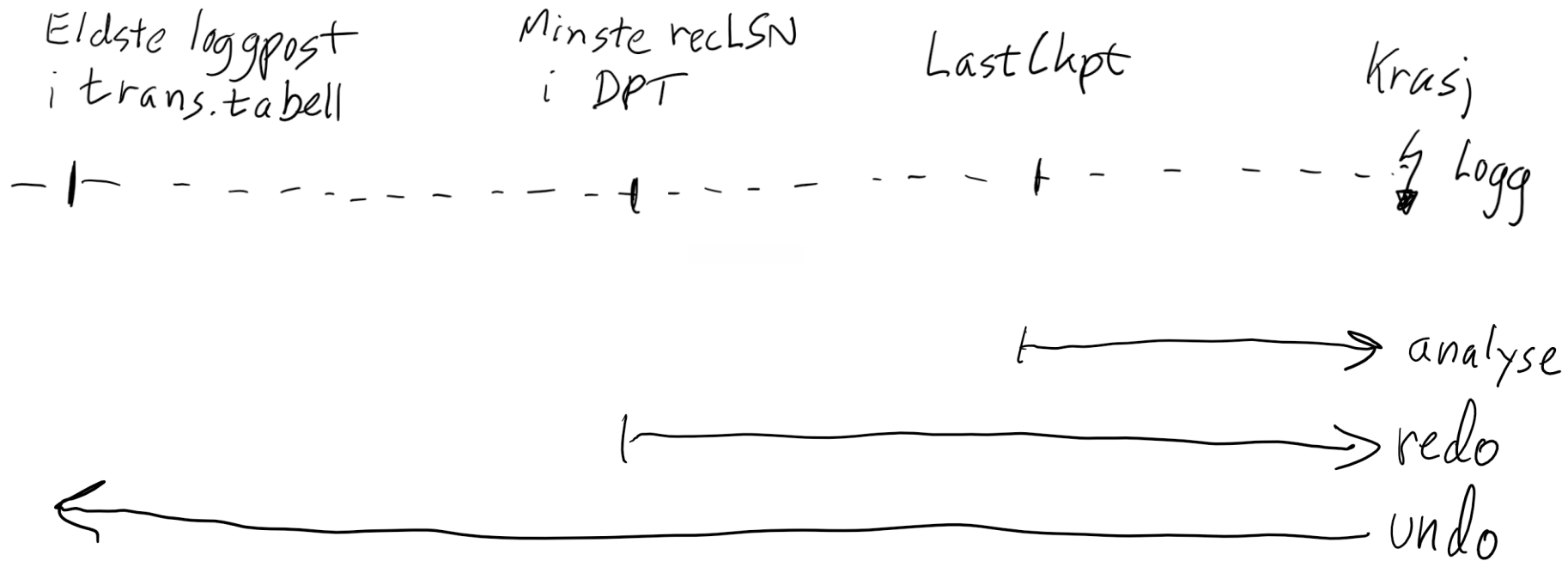


# Recovery etter krasj



- Mål:
  - Sørge for at vinnertransaksjoner er permanente.  
De som har committed før krasj.
  - Sørge for at tapertransaksjoner blir borte (aborted).  
De som ikke committed før krasj.
- Faser:
  1. **Analyse**: Finn vinnere og tapere. Rekonstruer DPT/TransTab
  2. **REDO**: Redo alle loggposter
  3. **UNDO**: Undo effekten av alle tapertransaksjoner

# 3 faser i Recovery



# Recovery – eksempel, analyse

LSN	Last_lsn	Transaction	OpType	Page_id	Other_info
101	0	T1	Update	C	...
102	0	T2	Update	B	...
103	101	T1	Commit		...
104			Begin_ckpt		
105			End_ckpt		
106	0	T3	Update	A	...
107	102	T2	Update	C	...
108	107	T2	Commit		...

Assume T1, T2 and T3 to be the transactions that exist and that the transaction table in log record 105 is this:

Transaction	Last_lsn	Status
T1	103	Commit
T2	102	In progress

⇒  
become

Transaction	Last_lsn	Status
T1	103	Commit
T2	108	Commit
T3	106	In progress

The Dirty Page Table that exists in log record 105 is this one:

Page_id	Rec_lsn
C	101
B	102

⇒  
become

Page_id	Rec_lsn
C	101
B	102
A	106

# REDO av loggpost (ARIES)



- Loggposten trenger ikke REDO hvis
  1. Den tilhørende blokken ikke er i dirty page table (DPT)
  2. Blokken er i DPT, men recLSN for blokken er større enn loggpostens LSN
  3. Blokkens pageLSN er større eller lik loggpostens LSN. Her må blokken leses inn.
- Ellers redo loggpost:
  1. Sett inn / skriv after image inn i blokken.
  2. Oppdater blokkens pageLSN til loggpostens LSN

# Andre recoveryteknikker



- *Undo/no-redo*: Som ARIES, men kun undo-logging
- *No-undo/redo*: Som ARIES, men kun redo-logging
- *Shadowing*: bruker ikke logging, men lager kopier av data ved oppdatering. Committer transaksjonen ved å kopiere inn pekere til nye data. Må ha katalog med pekere til data.
- Skiller mellom update-in-place og shadowing.

# Quiz 1



- Hva er en teknikk for å sikre Durability?
  - Undo logrecord
  - PageLSN
  - Force
  - NO-Force
- Hva er write-ahead logging?
  - Loggankeret sikres på disken
  - Loggen skrives før data
  - Data kopieres til disken for å få loggen liten
  - Grupper mange loggposter for stor båndbredde

# Quiz 2



- Hvorfor har vi Dirty Page Table?
  - Vi må vite hvilke blokker som må vaskes
  - Spare UNDO
  - Spare REDO
  - Gjenbruk av pages
- Why shouldn't we/Luke/Baby-Yoda use the FORCE?
  - Loggen blir et hot-spot
  - Billigere å skrive logg enn data
  - Spar det til Mandalorian season 4
  - Cachen blir invalidert

