UNIVERSITETET
I OSLO

INF1300
Introduksjon til databaser

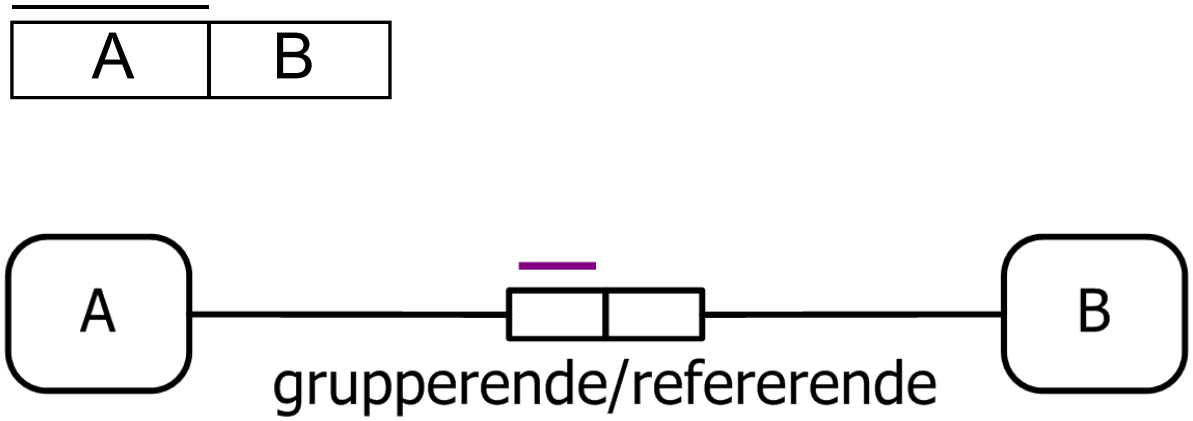
Dagens tema:

- **ORM og normalisering**
- **Denormalisering og splitting**
- **Triggere og databasefunksjoner**
- **Transaksjonshåndtering**

ORM og normalisering

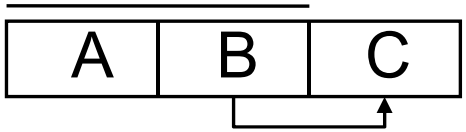
- Gruppering av ”korrekte” ORM-diagrammer gir **alltid** 3NF
- Gruppering av ”korrekte” ORM-diagrammer gir **som regel** BCNF. Unntakene skyldes alltid problemer knyttet til ekvivalente stier

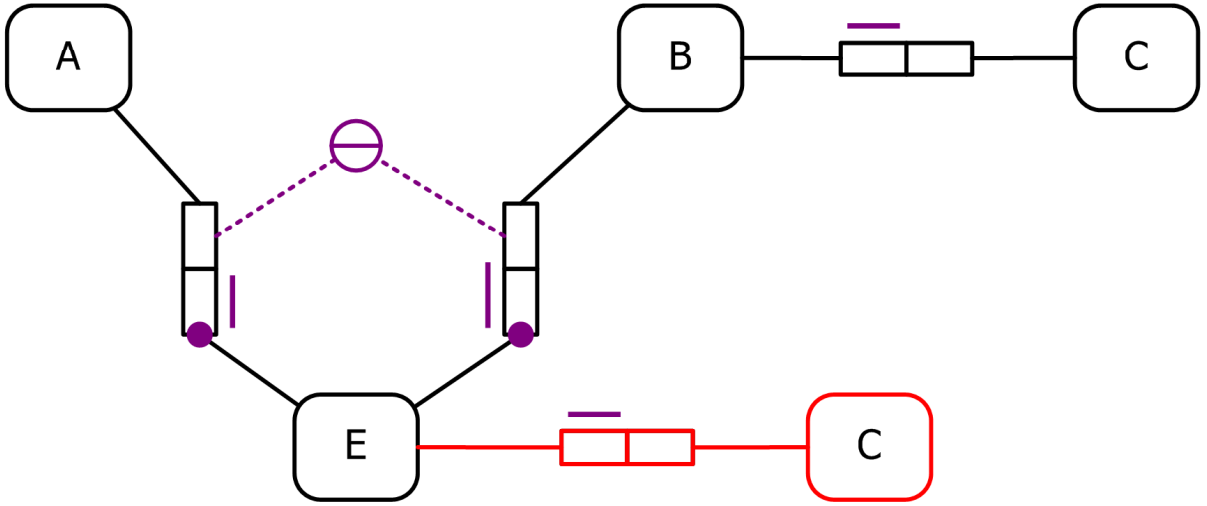
ORM og 1NF



Den grupperende rollen er alltid entydig.
Datamodellen blir alltid 1NF.

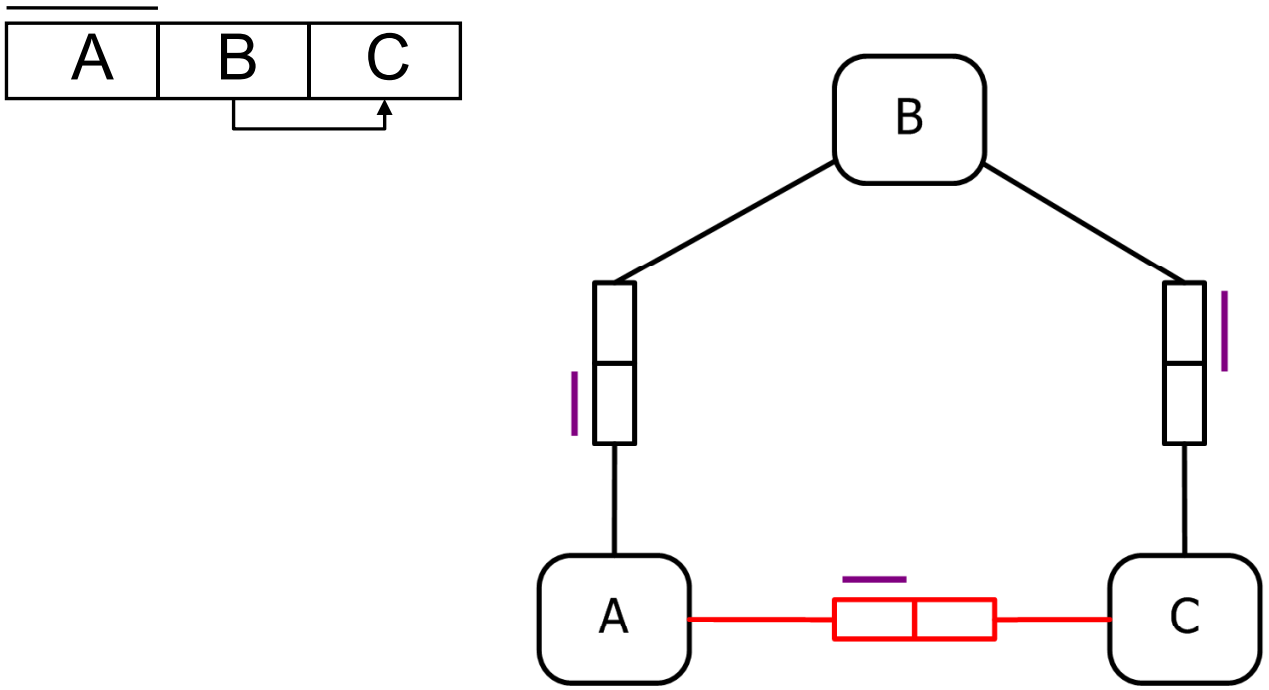
ORM og 2NF





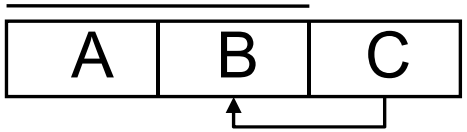
Hvis B→C plasseres på E også, vil 2NF bli brutt.

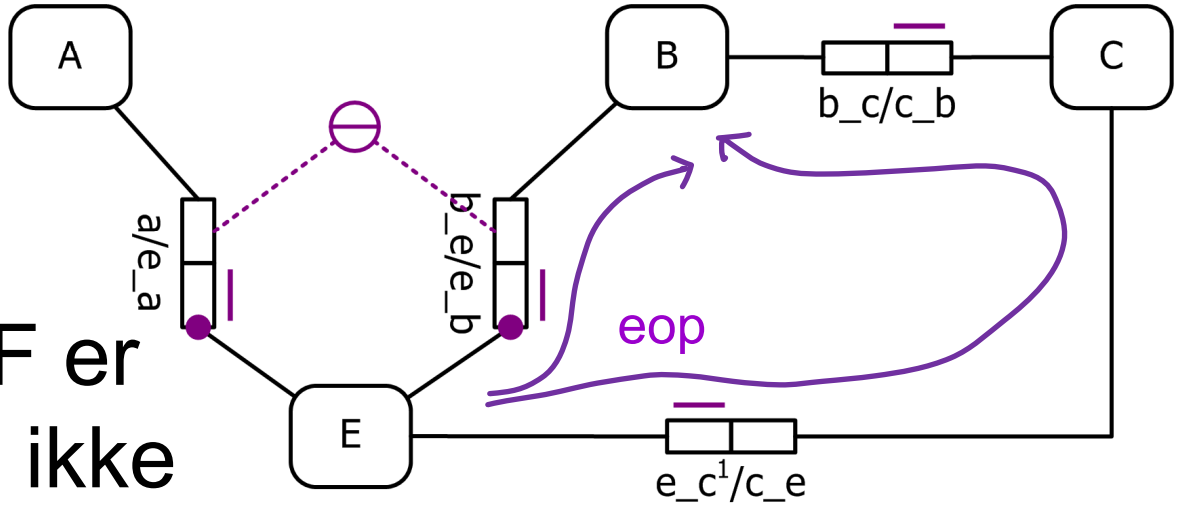
ORM og 3NF



Hvis A→C representeres i tillegg til A→B, B→C, vil 3NF bli brutt

ORM og BCNF





[1]**if an** E e_c C c_b **a** B
then that E e_b **that** B

Hvis BCNF er brutt, men ikke 3NF, skyldes det alltid ekviva-lente stier

Avvik fra optimal normalform

- Av hensyn til effektiviseringer eller forenklinger i applikasjonsprogrammene kan det enkelte ganger være hensiktsmessig å avvike fra den optimale normalformen som grupperingen av ORM-modellen gir
- Det er to måter å gjøre slike avvik på:
  - Denormalisering
  - Splitting av relasjoner

Denormalisering – 1

- Det å denormalisere det grupperte resultatet betyr å ta en join av to eller flere basisrelasjoner og la resultatet erstatte disse basisrelasjonene
- **Denormalisering gir alltid oppdaterings-anomalier**
- Kontroll av oppdateringsanomaliene fører til prosedyrerestriksjoner, enten i applikasjons-programmene eller i triggerprogrammer
- Hensikten er å spare tid ved at svært hyppige joiner er utført på forhånd

Denormalisering – 2

- Et meget vanlig eksempel er lagring av adresser
- En BCNF-normalisering av adresser vil gi to tabeller
  - En med gateadresse og postnummer
  - En med postnummer og poststed
- En denormalisering erstatter disse med én tabell med gateadresse, postnummer og poststed
- Dette gir mulighet for (feilaktig) å lagre samme postnummer med forskjellige poststeder
- Vi kan få DBMSet til å overholde regelen ved at vi benytter triggere og triggerprogrammer

Triggere og triggerprogrammer

- En trigger utløses av en hendelse som INSERT, DELETE eller UPDATE i databasen
- Triggere kan knyttes opp mot database-funksjoner (triggerprogrammer)
- Når triggeren utløses, utføres den tilhørende databasefunksjonen
- Hvordan man programmerer triggere og databasefunksjoner er svært DBMS-avhengig

Triggere og databasefunksjoner i Postgres

NB

Dette er kursorisk pensum

Dere skal vite hva triggere og databasefunksjoner er,
men dere vil ikke bli bedt om å programmere slike

PostgreSQL-funksjoner

- I Postgres 8.0 eller høyere lages funksjoner slik:

  CREATE FUNCTION *fnavn* ([*par1 type1* [, *par2 type2*] ...])
  RETURNS *returtype*
  AS $$ *programtekst* $$
  LANGUAGE '*programmeringsspråk*'

- Blant de lovlige programmeringsspråkene er
  - plpgsql (Postgres' eget PL/pgSQL)
  - sql (PostgreSQL)
  - perl (PL/Perl)
  - python (PL/Python)
- Med unntak av sql må språkene lastes inn f.eks. av superbruker, vanligvis som en del av installasjonen

Eksempel på PL/pgSQL

```
CREATE FUNCTION fakturasum ( fakturanr integer )
    RETURNS trigger
    AS $$
    DECLARE
            faktsum  NUMERIC(9,2);
    BEGIN
            SELECT SUM(belop) INTO faktsum
            FROM Fakturalinje fl
            WHERE fl.faktura = fakturanr ;
            UPDATE Faktura SET belop = faktsum
                    WHERE Faktura.faktnr = fakturanr ;
            RETURN NULL;
    END;
    $$ LANGUAGE 'plpgsql' ;
```

Triggere i PostgreSQL

- Syntaks for triggerdefinisjoner:

```
CREATE TRIGGER navn { BEFORE | AFTER }
    hendelse1 [ OR hendelse2 [ OR hendelse3 ] ]
    ON tabell FOR EACH { ROW | STATEMENT }
    EXECUTE PROCEDURE funksjon ( parameterliste )
```

- Hendelsene kan være INSERT, DELETE og UPDATE
- FOR EACH ROW betyr at *funksjon* kalles en gang for hver forekomst
- FOR EACH STATEMENT betyr at *funksjon* kalles bare en gang

Eksempel på trigger

```
CREATE TRIGGER sett_fakturabelop
AFTER INSERT OR UPDATE OR DELETE ON
        Fakturalinje
FOR EACH ROW EXECUTE PROCEDURE
        fakturasum (Fakturalinje.faktura) ;
```

Slutt på Postgres-spesifikt kursorisk stoff

Splitting – 1

- Splitting vil si at vi etter grupperingen velger å fordele attributtene i en basisrelasjon på flere basisrelasjoner (nedenfor kalt delrelasjoner)
- Det stilles vanligvis tre krav til en splitting, hvorav de to første er absolutte:
  - Alle attributter er med i minst en delrelasjon
  - Primærnøkkelen er med i alle delrelasjonene
  - Ingen attributter som ikke er med i primær-nøkkelen, er med i mer enn én delrelasjon

Splitting – 2

- Hvis det tredje kravet på forrige lysark er oppfylt, gir ikke splitting noen oppdateringsanomalier
- Splitting medfører alltid en økning i antall lese- og skriveoperasjoner
- Det er to hovedgrunner til splitting:
  - Sikkerhetshensyn
  - Økt effektivitet som et resultat av mindre behov for bufferplass
    (Splitting sparer ikke nevneverdig diskplass)

Splitting pga sikkerhetshensyn

- Splitting kan være nyttig i behandlingen av sensitive data
- Ta som eksempel pasientdata i helsevesenet:
  - Navn, fødselsdato o.l. kan ligge i én relasjon som er allment tilgjengelig
  - Diagnose og prøveresultater kan ligge i en annen relasjon med begrenset tilgjengelighet
- DBMSer har vanligvis adgangskontroll på tabeller, men ikke på enkeltattributter

Splitting pga effektivitet

- Splitting kan være formålstjenlig hvis den opprinnelige relasjonen har noen (store) attributter som er NULL i mange forekomster
- Dette kan bety mye for behovet for bufferplass, og dermed for mengden av disk-I/O
- Databaseadministratoren kan på en måte innføre «falske» underbegreper ved hjelp av slik splitting.

Ekte underbegreper oppstår når noen forekomster ***ikke kan*** ha verdi i et attributt; underbegrepene inneholder de forekomstene som ***har*** verdier i dette attributtet. I motsetning til disse skiller splitting ut de forekomstene som ***akkurat nå "tilfeldigvis"*** har en verdi.

Håndtering av BLOBer

- Den vanligste bruken av Binary large objects (**BLOB**er) er lagring av multimediadata (lyd, bilder og video)
- Selv om SQL tillater å ha BLOB som domene for et attributt, er det dumt å lagre et BLOB som del av en forekomst (det krever for mye bufferplass)
- I stedet lar vi attributtverdien være en peker inn i et separat diskområde hvor BLOBene lagres
- Dette kan minne om splitting, men regnes ikke som det

En-/flerbruker DBMSer
En-/multiprosessorer

- Et DBMS er et **enbrukersystem** hvis det bare tillater én bruker av gangen
- Et DBMS er et **flerbrukersystem** hvis flere brukere kan anvende systemet på samme tid
- I våre dager er det nærmest utenkelig at et DBMS er et enbrukersystem
- En **enprosessors** datamaskin kan understøtte et flerbrukersystems DBMS ved at parallelle eksekveringer **flettes** (engelsk: *interleaving*)
- En **multiprosessor** tillater ekte parallell prosessering av eksekveringer

Transaksjoner

- En **transaksjon** (databasetransaksjon) er et logisk stykke arbeid utført mot en database
- Transaksjoner må overholde integritetsreglene:

  Hvis databasen er i en lovlig tilstand før en transaksjon starter, så skal databasen være i en lovlig tilstand når transaksjonen avslutter
- En transaksjon skal oppfattes av omverdenen som en udelelig enhet

Lese- og skrivetransaksjoner

- En **lesetransaksjon** er en transaksjon som leser fra databasen, men ikke foretar endringer i den
- En **skrivetransaksjon** er en transaksjon som foretar endringer i databasen
- Lesetransaksjoner kan altså bare lese data fra databasen, mens skrive-transaksjoner både kan lese og modifisere dataene

Abort og commit

- Når en transaksjon skal avslutte, er det to måter det kan skje på:
  - Transaksjonen klarte ikke (eller fikk ikke lov til) å utføre alle sine lese- og skriveoperasjoner. Den må da avbrytes, og vi sier at den gjør **abort** (eller at den aborterer eller aborteres)
  - Transaksjonen fikk gjort alt den skulle og vil gjøre sine endringer tilgjengelig for andre. Den bekrefter da at den avslutter vellykket. Vi bruker det engelske ordet **commit** for dette

ACID-egenskapene

- **ACID**-egenskapene er fire krav til håndtering av transaksjoner som alle DBMSer må følge
- Ordet ACID består av forbokstavene på de engelske betegnelsen på disse kravene:
  - Atomicity
  - Consistency preservation
  - Isolation
  - Durability/permanency

Atomicity

- En transaksjon er en atomær prosesseringsenhet; den utføres enten i sin helhet (**commit**) eller ikke i det hele tatt (**abort**)

Consistency preservation

- En korrekt eksekvering av en transaksjon må ta databasen fra én konsistent tilstand til en annen. (Dette er en viktig del av definisjonen av en transaksjon og er strengt tatt et unødvendig krav, men all databaselitteratur har det med)

Isolation

- Eventuelle oppdateringer en transaksjon gjør, skal ikke gjøres synlige for andre transaksjoner før transaksjonen er committed (samtidige transaksjoner skal ikke vite om hverandre)

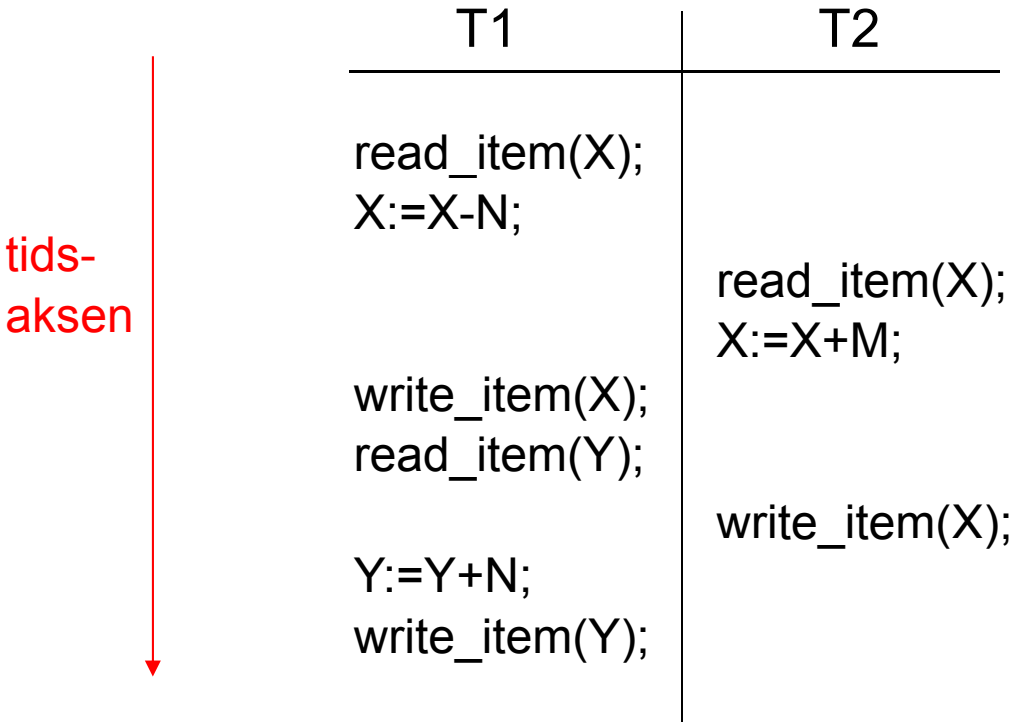
Durability/permanency

- Hvis en transaksjon oppdaterer databasen, og oppdateringene er committed, må ikke senere oppståtte feil kunne gjøre at oppdateringene går tapt (oppdateringene skal være varige)

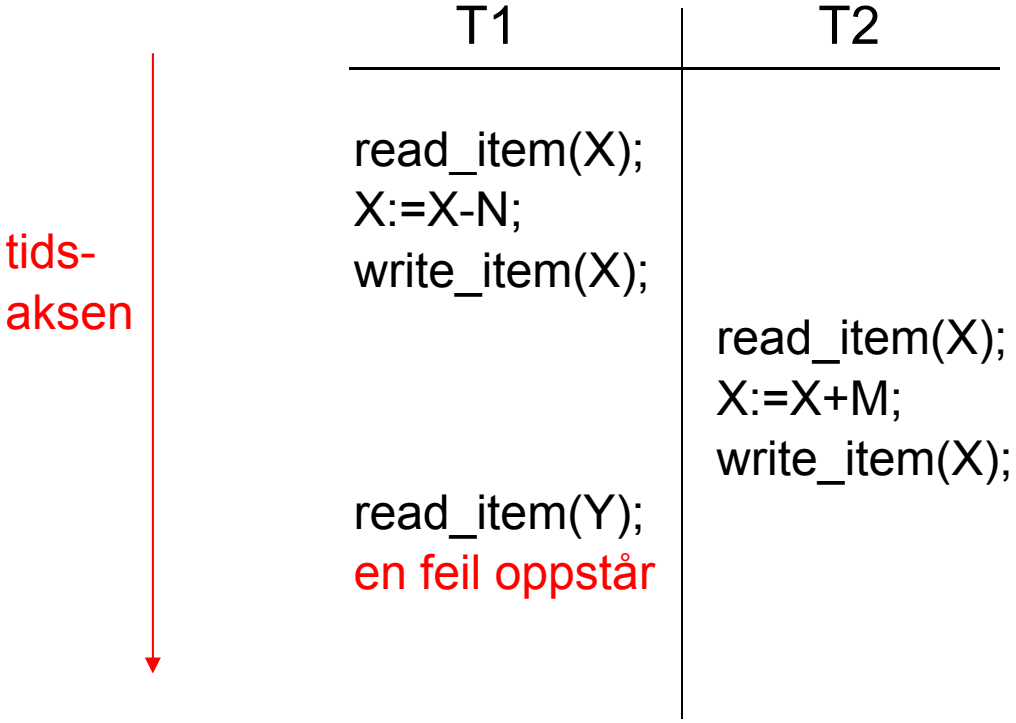
Hva kan gå galt?

- Vi skal nå se på tre eksempler på oppdateringsanomalier, det vil si eksekveringer som bryter mot ACID-reglene og derfor ikke kan tillates
- De engelske betegnelsene på eksemplene er:
  1. Lost update
  2. Dirty read
  3. Incorrect summary

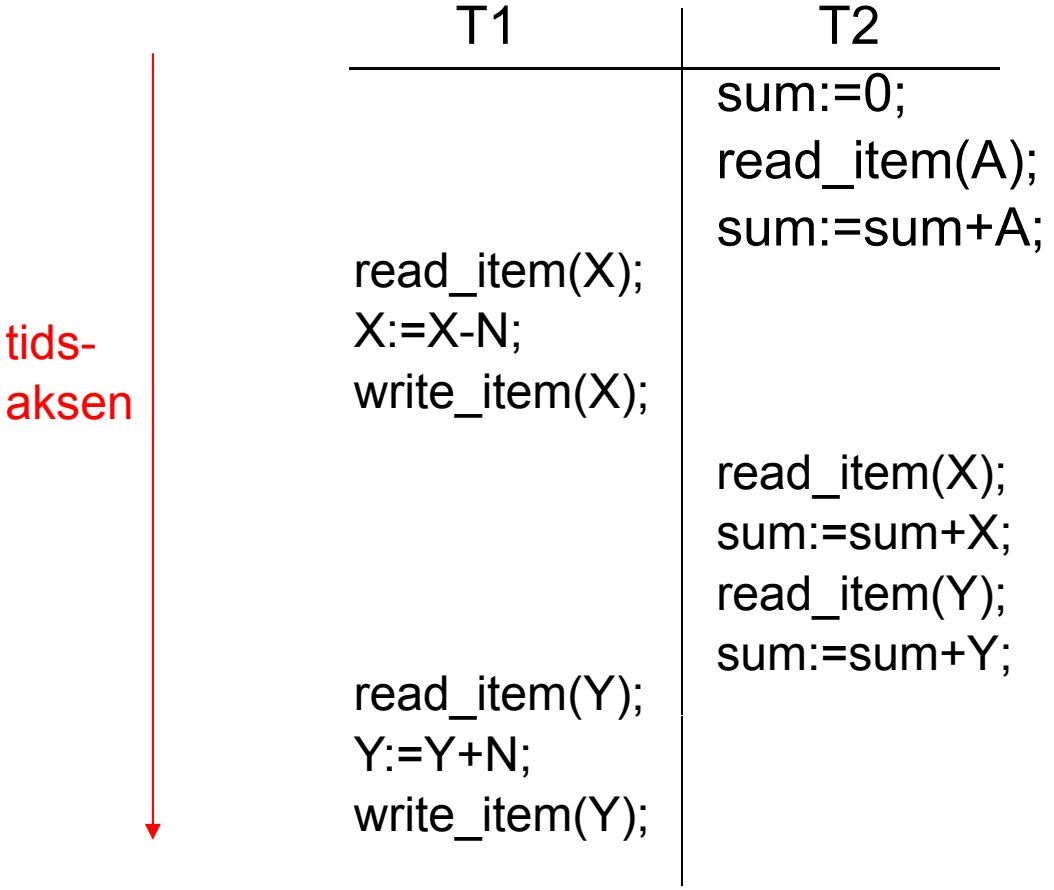
Eks. 1: Tap av oppdatering



Eks. 2: Midlertidig oppdatering



Eks. 3: Feil summasjon



Databaselogger

- En databaselogg er en fil hvor alle oppdateringer av databasen lagres
- Når en transaksjon skriver en variabel x, skrives en post i loggen som består av gammel og ny verdi av x, og hvilken transaksjon som gjorde oppdateringen
- Når en transaksjon gjør commit eller abort, skrives dette i loggen
- Når man tar backup av databasen, begynner man på en ny logg og kaster den gamle

Sikring av **A**CID

- **A**tomicity: En transaksjon er en atomær prosesseringsenhet

- Databasesystemets gjenopprettelsesmetode (engelsk: recovery method) har ansvaret for å sikre **A** ved at den omgjør eventuelle endringer en mislykket transaksjon T har rukket å påføre databasen
- Det gjøres ved å lese loggen og skrive tilbake de gamle verdiene av data som T har endret

Sikring av ACID

- **C**onsistency preservation: Transaksjoner bringer databasen fra én konsistent tilstand til en annen

- **C** sikres delvis av databasehåndteringssystemet ved at dette garanterer at visse typer integritets-regler ikke blir brutt
- Dersom DBMSet ikke kan håndtere en regel, må databaseprogrammereren (transaksjons-programmereren) ta ansvaret for at konsistens bevares

Sikring av ACID

- Isolation: Oppdateringer skal ikke være synlige for andre før transaksjonen er committed

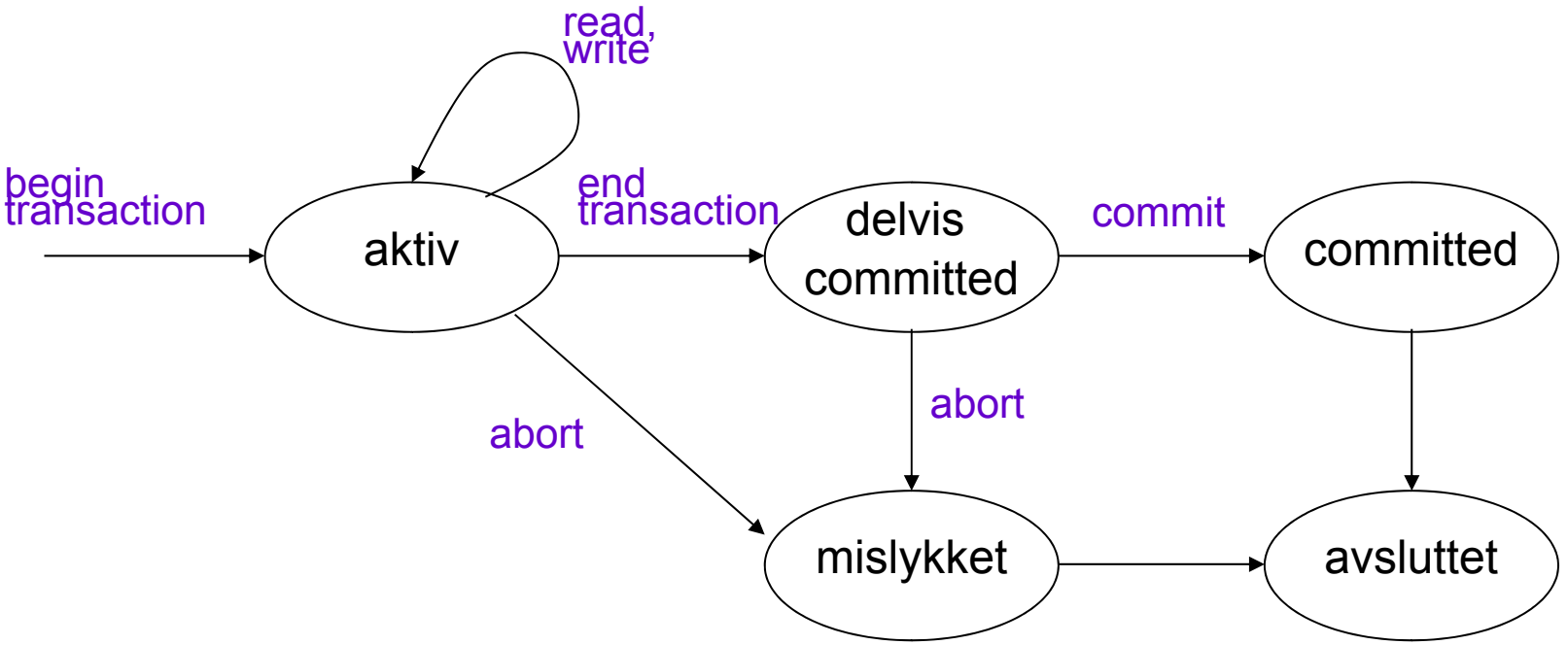
- Samtidighetskontrollen (engelsk: concurrency control method) har ansvaret for å sikre I
- Det finnes mange metoder for å gjøre dette, og I er den av ACID-egenskapene som er vanskeligst (og dermed mest interessant) å håndheve

Sikring av ACID

- **D**urability/permanency: Oppdateringer som er committed, er varige

- Databasesystemets gjenopprettelses-metode har ansvaret for å sikre **D**
- Etter et systemkræsj leses loggen, og data som er skrevet av committede transaksjoner, gjenopprettes i databasen

Transisjonsdiagram for eksekveringen av en transaksjon



Oversikt over DBMS-oppgaver

- Tolking av SQL-kode og oversettelse til relasjonsalgebra
- Optimalisering av relasjonsalgebrauttrykk til best mulige eksekveringsplaner
- Utføring av eksekveringsplaner
- Buffer- og diskhåndtering
- Transaksjonshåndtering og samtidighetskontroll
- Logging og gjenoppretting etter feil og aborter