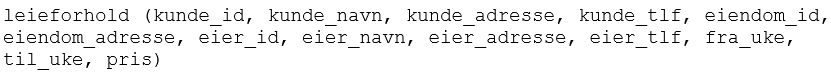
Øving 4: Normalisering og Transaksjoner

# Del 1: Normalisering



1. *Mulige kandidatnøkler fra tabell ovenfor:*

* kunde\_id
* eiendom\_id
* *eier\_id*
* fra\_uke
* til\_uke
* (kunde\_id, eiendom\_id)
* (kunde\_id, eiendom\_id, fra\_uke, til\_uke)
* *(kunde\_id, eier\_id, eiendom\_id, fra\_uke, til\_uke)*
* (kunde\_id, fra\_uke)

1. Grunnen til at tabellen ikke er problemfri mht til registrering og sletting av data er at hvis man prøver å slette ett spesifikt leieforhold, med f.eks. kunde\_id, kan man risikere å slette alle tidligere leieforhold denne kunden har gjort. Motsatt for registrering er det ingen ordentlig referanseintegritet for kunde, eiendom, eller eier, slik at man i utgangspunktet kan registrere to leieforhold med to forskjellige kunder med samme kunde\_id.

*Et bilde som inneholder skjermbilde, line, Plottdiagram, diagram

Automatisk generert beskrivelse*Funksjonelle avhengigheter mellom alle attributter:

*// Prisen burde være avhengig på eiendommen og tidsperioden*

1. Burde ha følgende funksjonelle avhengigheter:

* kunde\_id → (kunde\_navn, kunde\_adresse, kunde\_tlf)
* eiendom\_id → eiendom\_adresse
* eier\_id (eier\_navn, eier\_adresse, eier\_tlf)
* (eiendom\_id, fra\_uke, til\_uke) pris

Dermed har vi tabellen på 1NF. (Teknisk sett ikke siden denne tabellen ikke har en primærnøkkel). Vi kan få den på 2NF hvis vi splitter den i mindre tabeller og dermed også unngår partielle avhengigheter:

* KUNDE.kunde\_id\* → KUNDE.(kunde\_navn, kunde\_adresse, kunde\_tlf)
* EIENDOM.eiendom\_id\* → EIENDOM.eiendom\_adresse
* LEIEFORHOLD.(eiendom\_id, fra\_uke, til\_uke) → LEIEFORHOLD.pris
* EIER.eier\_id\* → EIER.(eier\_navn, eier\_adresse, eier\_tlf)  
    
  Et bilde som inneholder tekst, sirkel, skjermbilde, diagram

  Automatisk generert beskrivelse

For å få den til 3NF må vi unngå transitive avhengigheter. Fra figuren over ser vi at vi ikke har transitive avhengigheter, jo mindre man skal argumentere for at en adresse er avhengig av et navn. F.eks. Ola Normann → ola.normann@gmail.com. Man kan i teorien ha en epost adresse som er totalt annerledes enn sitt eget navn, så derfor ignorerer jeg dette. Dermed kan vi si at vi ikke har noen transitive avhengigheter og har derfor tabellene på 3NF.

For å få tabellen på BCNF må vi unngå funksjonelle determineringer mellom kandidatnøkler, og at enhver determinant er en kandidat nøkkel. Siden vi ikke har determineringer mellom kandidatnøkler, eller overlappende kandidatnøkler, og hver determinant er en kandidatnøkkel, kan vi også si at tabellen tilfredsstiller BCNF.

*Tabeller:*

KUNDE(kunde\_id, kunde\_navn, kunde\_adresse, kunde\_tlf)  
EIER(eier\_id, eier\_navn, eier\_adresse, eier\_tlf)  
EIENDOM(eiendom\_id, eiendom\_adresse)  
LEIEFORHOLD(kunde\_id, eier\_id, eiendom\_id, fra\_uke, til\_uke, pris)

*Relasjon:*  
  
KUNDE.kunde\_id\* → KUNDE.(kunde\_navn, kunde\_adresse, kunde\_tlf)  
EIENDOM.eiendom\_id\* → EIENDOM.eiendom\_adresse  
LEIEFORHOLD.(eiendom\_id, fra\_uke, til\_uke) → LEIEFORHOLD.pris  
EIER.eier\_id\* → EIER.(eier\_navn, eier\_adresse, eier\_tlf)

Kan vi løse denne oppgaven ved å gjennomføre prosessen 1NF --> 2NF --> 3NF? Ja det kan vi. Tilnærmingen min var faktisk å bruke denne prosessen utenom at jeg var klar over det. Siden 2NF forventer at tabellen er på 1NF og ikke har partielle avhengigheter, og 3NF forventer at tabellen er på 2NF og uten transitive avhengigheter, kan dette gjøres stegvis.

Del 2: Transaksjoner

# Frivillig – Manuell bruk av skrive- og leselåser

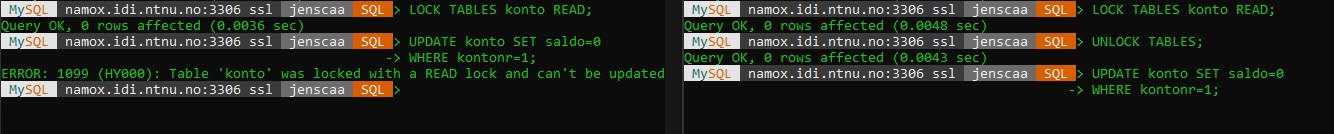
## Lese-lås

Bildet under viser at en person (konsollen til venstre) setter lese-lås på tabellen konto.

Et bilde som inneholder Multimedieprogramvare, programvare, Grafikkprogramvare, skjermbilde

Automatisk generert beskrivelse

* Etter å ha kjørt spørring «SELECT \* FROM konto» i begge konsollene får jeg opp en utskrift av innholdet i tabellen i begge konsollene.
* De er også mulig å sette en lese-lås på den siste klienten (konsollen til høyre).
* Fjerner lese-låsen i den siste klienten (konsollen til venstre). Etter å ha kjørt «UPDATE konto SET saldo=0 WHERE kontonr=1;» fikk jeg følgende resultat:



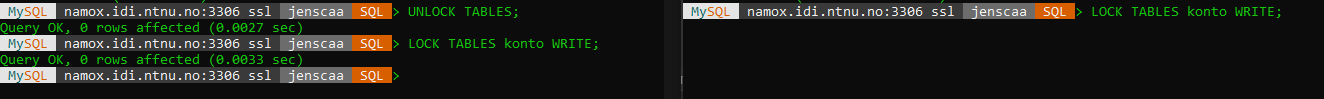
Her settes det en lese-lås på tabellen konto i konsollen til høyre og venstre. I konsollen til venstre settes skrur vi så av skrive-låsen. Deretter kjøres kommandoen «UPDATE konto SET saldo=0 WHERE kontonr=1;». Siden standarden til LOCK TABLES konto READ er LOCAL, kan man ikke endre på tabellen med lese-lås. Lese-lås tillater heller ikke andre klienter å skrive til tabellen, så derfor venter konsollen til høyre. Hvis man fjerner lese-låsen i konsollen til venstre vil UPDATE kommandoen i konsollen til venstre gå igjennom.

## Skrivelås

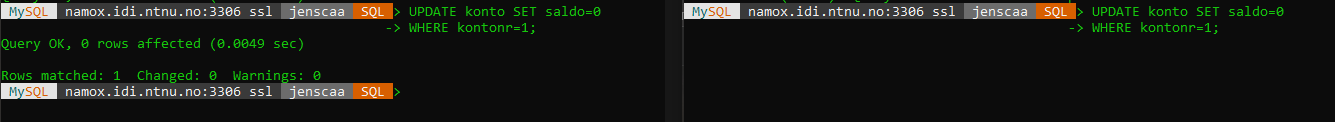
Bildet under viser at en klient (konsollen til venstre) setter en skrivelås på kontotabellen.  
Et bilde som inneholder skjermbilde, tekst, programvare, Multimedieprogramvare

Automatisk generert beskrivelse

* Etter å ha kjørt setningen «SELECT \* FROM konto;» i begge konsollene får jeg bare opp innholdet i konsollen til venstre med skrivelås, men ikke den uten skrivelås (Skrivelås kan bare være på en klient om gangen).
* Etter å ha låst opp tabellen i konsollen til venstre, også satt på skrivelås på nytt, er det ikke mulig å sette skrivelås på tabellen i konsollen til venstre. Konsollen henger seg sammen.



* Etter å ha kjørt kommandoen «UPDATE konto SET saldo=0 WHERE kontonr=1;» i begge klientene fikk jeg dette resultatet:



I konsollen til venstre, altså klienten med skrivelås til tabellen, fikk vi gjennom kommandoen UPDATE kommandoen, men siden vi tidligere hadde endret saldoen til kontonr 1, får vi beskjeden om at 0 endringer er blitt gjort til tabellen. I konsollen til høyre derimot går ikke UPDATE kommandoen i gjennom. Det er fordi en skrivelås ikke gir andre klienter tilgang til tabellen i det hele tatt, og derfor ender vi opp med at konsollen henger seg sammen. Klienten med skrivelås har rett til å lese og skrive til tabellen som en vil. Bare når skrivelåsen er opphevet går UPDATE kommandoen i konsollen til høyre igjennom.

# Transaksjoner

For å starte en transaksjon kan man benytte kommandoene START TRANSACTION eller bare BEGIN. Forskjellen med disse kommandoene er at START TRANSACTION tillater flere modifiseringer som WITH CONSISTENT SNAPSHOT, READ WRITE og READ ONLY, noe som ikke BEGIN tillater. For å avslutte en transaksjon kan man benytte kommandoene COMMIT eller ROLLBACK. Forskjellene mellom disse er en COMMIT utfører endringene som er gjort i transaksjonen, mens ROLLBACK nullstiller endringene som er gjort i transaksjonen til de originale verdiene.

Bildet på neste side viser en transaksjon som avsluttes med COMMIT. Her endres saldoen til konto 6 til 137.

Et bilde som inneholder tekst, skjermbilde, programvare, display

Automatisk generert beskrivelse

På bildet under vises en transaksjon som avsluttes med ROLLBACK.  
Et bilde som inneholder tekst, skjermbilde, programvare, display

Automatisk generert beskrivelse

Her prøver vi bevist å slette alle kontoene med saldo som er delelig på 5. Men siden vi avslutter med ROLLBACK blir endringene ikke lagret.

# TEORI

*Hvilke typer låser har databasesystemene?*

Database systemene har to type låser: *Shared lock* det vi kaller for lese-lås. Og *exclusive lock* det vi kaller for skrivelås.

* Lese-lås:
  + Brukes for lesing av tabeller og godtar lesning for transaksjoner.
  + Flere klienter og transaksjoner kan ha lese-lås på samme objekter samtidig.
  + Tillater kun andre klienter og transaksjoner å lese, ikke skrive.
  + Sesjonen som bruker lese-lås tillater kun å lese, ikke skrive til objekter.
* Skrive-lås:
  + Kan bare settes av en klient/transaksjon av gangen.
  + Tillater ikke andre klienter/transaksjoner å lese eller skrive til objekter.
  + Må oppheve skrive-låsen for at andre klienter/transaksjoner skal kunne lese og modifisere på objekter.

*Hva er grunnen til at man gjerne ønsker lavere isolasjonsnivå enn SERIALIZABLE?*

Grunnen til at man ønsker lavere isolasjonsnivå enn SERIALIZABLE er fordi man oftest anser dette isolasjonsnivået for å være for strengt. SERIALIZEABLE setter låser på alt (SELECT/INSERT/UPDATE/DELETE). Dette medfører til at transaksjoner kjøres i serie. Det er ikke noe problem i seg selv å bruke SERIALIZEABLE, men problemet er at transaksjoner kan utføres mye tregere, og kan henge seg sammen. Derfor ønsker man gjerne et lavere isolasjonsnivå.

*Hva skjer om to pågående transaksjoner med isolasjonsnivå serializable prøver select sum(saldo) from konto?*

Hvis to pågående transaksjoner med isolasjonsnivå SERIALIZEABLE prøver SELECT SUM(saldo) FROM konto, kommer transaksjonene til å gjøres i serie, altså etter hverandre, og ikke parallelt.

*Hva er to-fase-låsing?*

To-fase-låsing er en måte å benytte låsene på som garanterer riktig utførelse av transaksjonene. Man deler gjerne dett inn i to faser: Fase 1 og fase 2.

* Fase 1:
  + Utvidelsesfasen.
  + Låser settes på etter hvert som de trengs i transaksjonen.
* Fase 2:
  + Avslutningsfasen.
  + Låser opp alle låsene på en gang (med ROLLBACK/COMMIT).

*Hvilke typer samtidighetsproblemer (de har egne navn) kan man få ved ulike isolasjonsnivåer? Hva er optimistisk låsing/utførelse? Hva kan grunnen til å bruke dette være?*

Ved ulike isolasjonsnivåer kan vi få flere samtidighetsproblemer. Disse problemene er følgende: Tapt oppdatering, Ikke-bekreftede data og inkonsistent uthenting av data.

* Tapt oppdatering:
  + En oppdatering blir skrevet over en annen oppdatering.
  + Kan få dette ved isolasjonsnivåene: READ UNCOMMITTED
* Ikke-bekreftede data:
  + En transaksjon får se data fra en annen transaksjon før disse er bekreftet.
  + Kan få dette ved isolasjonsnivåene: READ UNCOMMITTED og READ COMMITED.
* Inkonsistent uthenting av data:
  + En transaksjon leser data som er delvis oppdatert fra en annen transaksjon.
  + Kan få dette ved isolasjonsnivåene: READ UNCOMMITTED, READ COMMITED og REPEATABLE READ.

Optisk låsing/utførelse er et begrep som i praksis ikke er låsing i det hele tatt. Det går ut på at hvis en klient f.eks. ønsker å oppdatere en tabell, så har man et ekstra felt i tabellen som registrer om andre har gjort endringer mot denne tabellen. Hvis det har blitt gjort en annen oppdatering i mellomtiden fra klienten leser fra og oppdaterer på tabellen, så må dataen lese ut dataen på nytt.

Grunnen til å bruke dette er for å sikre at de endringene som skal bli utført, må bli utført riktig i forhold til andre endringer som andre transaksjoner har gjennomført.

*Hvorfor kan det være dumt med lange transaksjoner (som tar lang tid)? Vil det være lurt å ha en transaksjon hvor det kreves input fra bruker?*

Grunnen til at det kan være dumt med lange transaksjoner er at jo lengre en transaksjon er, jo større er sannsynligheten for at konflikter og feil oppstår med andre transaksjoner. Og dermed må transaksjonen ved behov utføre en ROLLBACK. I tillegg kan transaksjoner blokkere andre transaksjoner ved å låse tabeller. Hvis transaksjonen i tillegg er lang, kan dette føre til at det vil ta lengre tid før neste transaksjon kan brukes. Til sist kan lange transaksjoner gi dårlig ytelse i form treg respons grunnet potensielle køer, konflikter og annet unødvendig arbeid som lett kunne vært unngått.

I enkelte tilfeller kan det være lurt å ha transaksjoner hvor det kreves input fra bruker. Dette vil jeg anta at gjelder oppdateringer i databaser eller innsetting eller noe liknende. Men et mulig problem som kan oppstå er hvis en transaksjon setter låser og venter på brukerinput, slik at andre transaksjoner settes på vent. Hvis en bruker ikke svarer, kommer alt til å settes i stop. Man burde derfor vurdere å implementere håndteringer for slike scenarioer.

Oppgaver om Transaksjoner

**Oppgave 1**

|  |  |  |
| --- | --- | --- |
| **Tid** | **Klient 1** | **Klient 2** |
| 1 | Sett isolasjonsnivå til **read uncommitted**. | Sett isolasjonsnivå til **serializable**. |
| 2 | Start transaksjon. | Start transaksjon. |
| 3 |  | select \* from konto where kontonr=1; |
| 4 | select \* from konto where kontonr=1; |  |
| 5 | update konto set saldo=1 where kontonr=1; |  |
| 6 |  | commit; |
| 7 | commit; |  |

*Hva skjer og hvorfor? Hva hadde skjedd om Klient 2 hadde brukt et annet isolasjonsnivå?*

Svar:

Siden klient 1 er i isolasjonsnivå READ UNCOMMITTED og ikke bryr seg om låser, kommer ikke klient 2 sin lese-lås på konto nummer 1 til å spille noen rolle.

Hvis klient 2 hadde brukt et annet isolasjonsnivå, hadde vi ikke merket noe forskjell.

Siden klient 2 setter en lese-lås på SELECT setningen så opprinnelig ikke klient 1’s spørringer på tid 4 og 5 til å gjennomføres helt til klient 2 utfører commit på tid 6.

Hvis klient 2 hadde brukt et annet isolasjonsnivå, så hadde det ikke hatt noe å si for responstiden til klient 1. Altså spørringene fra klient 1 og 2 vil utføres i rekkefølge.

**Oppgave 2**

**a)**

|  |  |  |
| --- | --- | --- |
| **Tid** | **Klient 1** | **Klient 2** |
| 1 | Sett isolasjonsnivå til **read uncommitted**. | Sett isolasjonsnivå til **read uncommitted**. |
| 2 | Start transaksjon | Start transaksjon |
| 3 | update konto set saldo=1 where kontonr=1; |  |
| 4 |  | update konto set saldo=2 where kontonr=1; |
| 5 | update konto set saldo=1 where kontonr=2; |  |
| 6 | **commit;** | update konto set saldo=2 where kontonr=2; |
| 7 |  | **commit;** |

*Hva blir resultatet? Hvorfor må det være slik?*

Svar:

Siden begge klientene bruker et isolasjonsnivå med READ UNCOMMITTED så har rekkefølgen noe å si dersom oppdateringer og innsettinger blir gjort. I transaksjonen til klient 1 oppdateres saldoen til konto 1 til 1. I klient 2 oppdateres konto 1 til 2. Altså vi får en tapt oppdatering når klient 2 overskriver oppdateringen til klient 1. Akkurat det samme skjer igjen når klient 1 og 2 oppdaterer konto 2. Sluttresultatet av dette er at konto 1 og 2 får henholdsvis saldoen 2 og 2. Vi ser bare at endringene fra klient 2 har gått igjennom.

Dette skjer fordi vi bruker READ UNCOMMITTED, et isolasjons nivå som tillater flerbruksproblemet: Tapt oppdatering

**b)**

|  |  |  |
| --- | --- | --- |
| **Tid** | **Klient 1** | **Klient 2** |
| 1 | Sett isolasjonsnivå til **read uncommitted**. | Sett isolasjonsnivå til **read uncommitted**. |
| 2 | Start transaksjon | Start transaksjon |
| 3 | update konto set saldo=1 where kontonr=1; |  |
| 4 |  | update konto set saldo=2 where kontonr=2; |
| 5 | update konto set saldo=1 where kontonr=2; |  |
| 6 |  | update konto set saldo=2 where kontonr=1; |
| 7 |  |  |

*Hva blir resultatet? Forklar forskjellen fra oppgave a).*

*Vil det ha noe å si om man endrer isolasjonsnivå på klientene?*

Svar:

I likhet med oppgave a) vil vi få tapt oppdatering, men ikke på nøyaktig samme måtte. Istedenfor for å bare få tapt oppdatering i klient 1, får vi tapt oppdatering i både klient 1 og klient 2. Altså oppdateringene på Tid 3 og 4, kommer til å bli overskrevet av oppdateringene på Tid 5 og 6.

Det kommer til å ha litt å si dersom man endrer isolasjonsnivåene på klientene. Hvis man f.eks. endrer klientenes isolasjonsnivå til READ COMMITTED, REPEATABLE READ eller SERIALIZEABLE (uvesentlig hva man endrer det til i dette tilfellet), kommer klientene til å låse sine data slik at de kommer i vranglås neste gang de prøver å oppdatere hverandres data. Med andre ord får vi en DEADLOCK.

**Oppgave 3**

|  |  |  |
| --- | --- | --- |
| **Tid** | **Klient 1** | **Klient 2** |
| 1 | Sett isolasjonsnivå til **read uncommitted**. | Sett isolasjonsnivå til **serializable**. |
| 2 | Start transaksjon. | Start transaksjon. |
| 3 | select sum(saldo) from konto; |  |
| 4 |  | update konto set saldo=saldo + 10 where kontonr=1; |
| 5 | select sum(saldo) from konto; |  |
| 6 |  | commit; |
| 7 | select sum(saldo) from konto; |  |
| 8 | commit; |  |

*Hva skjer?*

Svar:

I dette tilfellet bruker klient 1 isolasjonsnivået READ UNCOMMITTED, mens klient 2 bruker SERIALIZABLE. Altså først skriver klient 1 ut summen fra kontoen, også inkrementerer klient 2 saldoen med 10 på konto 1. Deretter prøver klient 1 å skrive ut summen igjen. Summen kommer til å bli oppdatert til 1 siden klient 1 er i isolasjonsnivået READ UNCOMMITTED. Etter at klient 2 har utført COMMIT, prøver klient 1 å skrive ut summen igjen og får samme resultat som forrige forsøk.

*Hva vil skje om Klient 1 bruker read committed, repeatable read eller serializable?*

Svar:

Hvis klient 1 bruker READ COMMITED, REPEATABLE READ eller SERIALIZABLE kommer ikke vi til å få samme resultat som over. Forskjellen er klient 1 ikke kommer til å merke endringene i klient 2 i det hele tatt. Forskjellen blir bare lagt merke til når klient 2 har utført COMMIT.

**Oppgave 4**

*Lag en kjøring med to klienter som tester phantom reads. Her kan det være lurt å tenke igjennom isolasjonsnivå. Om resultatet ikke er som forventet så kan det være lurt å sjekke dokumentasjonen.*

Svar:

Et bilde som inneholder tekst, skjermbilde, programvare, Multimedieprogramvare

Automatisk generert beskrivelse

Dette resultatet med REPEATABLE READ fikk ikke forventet resultat. Grunnen til dette skal være at REPEATABLE READ trolig låser SELECT spørringene selv om de i teorien ikke burde. Ifølge dokumentasjonen låser REPEATABLE READ bare SELECT med FOR UPDATE eller FOR SHARE.

Med READ COMMITTED får vi ønskede resultat:  
Et bilde som inneholder skjermbilde, tekst, programvare, Multimedieprogramvare

Automatisk generert beskrivelse