Øving 5 - Normalisering og transaksjoner

Av: John I. Eriksen

Del 1 - Normalisering

Kandidatnøkler

Tabell:

```
leieforhold (kunde_id,
             kunde_navn,
             kunde_adresse,
             kunde_tlf,
             eiendom_id,
             eiendom_adresse,
             eier_id,
             eier_navn,
             eier_adresse,
             eier_tlf,
             fra_uke,
             til_uke,
             pris
             )
```

Kandidatnøkler for tabellen, gitt at en person kun kan leie én eiendom av gangen og at en eiendom kan leies ut til kun én person av gangen:

```
(kunde_id, fra_uke, til_uke)
(eiendom_id, fra_uke, til_uke)
```

Ved å kombinere attributtene i én nøkkel, lages kompositttnøkler. Dette gjør at de kan unikt identifisere et spesifikt leieforhold, siden de spesifiserer (her) en spesifikk kunde og/eller eiendom, i tillegg til tidsintervallet vi ønsker å se på. Brukes de sammen (både kunde++ og eiendom++) kan vi finne overlappen til dem og dermed identifisere hvem som leide hva i et gitt tidsrom.

Problemer med registrering og sletting av data

Problemene med tabellen er

Redundans

For hvert nytt `leieforhold`, legges det inn kundedata og eiendomsdata på nytt. Dette gjør at hvis f.eks en person leier flere ganger kan det legges inn ulik data for hver gang på samme person. Selv om en kunde har én ID, har man anledning til å legge inn f.eks ulikt telefonnummer eller adresse på de ulike oppføringene. Dette kan føre til at at samme informasjon må fylles inn mange steder (redundans), i tillegg til at databasen risikerer bli fylt opp med ubrukelig informasjon om den ikke oppdateres på hvert eneste sted den er lagt

inn.

Innsetting

Det er ikke mulig å legge inn om *bare* en ny kunde eller en eiendom. Tabellen er konstruert for å representere spesifikke leieforhold, med kunde, eiendom og eier. Man kan altspå ikke isolere informasjon om kunder eller eiendommer på en måte som ikke går ut over tabellenstrukturen. Man kunne lagt inn f.eks NULL eller placeholder-verider, men dette ville ført til dobbeltarbeid og ubrukelig data i tabellen.

Uregelmessighet ved sletting

Samme som for innsetting. Om man vil slette et spesifikt leieforhold, vil alle data vedrørende dette kunde-, eiendom- og eierforholdet slettes, fordi all data om en eier, eiendom ogkunde er knyttet sammen på én oppføring. All data som de involverte partene (eier, eiendom og kunde) vil da slettes fra databasen, med mindre disse er lagret i andre leieforhold. Dvs. man kan ikke slette et leieforhold uten å miste all relatert data om kunden, eier og eiendommen.

Uregelmessighet ved oppdatering

Siden all data om kunde, eier og eiendom må registreres på nytt for hvert leieforhold, vil det være vanskelig og tidkrevende å holde informasjon om en spesifikk gjentagende part oppdatert. F.eks hvis en kunde endrer adresse, eller en eier endrer telefonnummer, vil dette måtte oppdateres manuelt for hver oppføring hvor disse er involvert. Dette er, som nevnt, tidkrevende, og ikke minst er det overhengende fare for at det blir gjort skrivefeil e.l., hvilket vil føre til feil informasjon som kan være vanskelig å finne tilbake til i ettertid. Hvis informasjonen ikke blir oppdatert, eller blir oppdatert forskjellig, vil det føre til inkonsistent informasjon i tabellen. F.eks flere forskjellig telefonnummer på samme eier/kunde.

Kompleksitet ved spørringer

Fordi samme informasjon om f.eks en kunde gjentas for hver eneste gang den akutelle kunden forekommer i et leieforhold, vil en spørring på f.eks telefonnummer måtte sjekke alle absolutt alle rader for å finne ut om det aktuelle telefonnummeret allerede er hentet ut. Dette vil gjøre at spørringene blir "tyngre" å prosessere, dvs. tar lengre tid og krever mer ressurser. Med kompleksitet menes altså til hvor tung beregningene som må gjøres i spørringene er, hvor mye datakraft og tid som kreves.

Konklusjon

Tabellen burde deles opp i flere mindre tabeller, for å isolere de forskjellige delene, og gjøre det enklere å bruke med den på en fornuftig måte.

Diagram: Funksjonelle avhengigheter

Fra læreboka:

> "I tabellen R determinerer kolonne A en annen kolonne B dersom det for hver A-verdi kun knyttes én B-verdi.
> Det kan uttrykkes slik: `r.a --> r.b`

Setter opp diagrammer bestående av piler som peker fra kandidatnøkler til de avhengige attributtene
`kandidatnøkkel --> avhengig_attributt1, avhengig_attributt2`

`kunde_id` determinerer alle kolonner relatert til kunde:
`kunde_id --> kunde_navn, kunde_adresse, kunde_tlf`

Fordi: Siden `kunde_id` er unik for hver kunde, vil denne være identifisere den unike kunden og dens tilhørende navn, adresse of tlf. Vi antar da at det ikke er lagt inn oppføringer på `kunde_id` som inneholder avvikende data i disse attributtene.

`eiendom_id` determinerer alle kolonner relatert til eiendom:
`eiendom_id --> eiendom_adresse`

Fordi: Siden `eiendom_id` er unik for hver eiendom, vil denne identifisere en unik eiendom, slik som `kunde_id` gjør for kunder, med samme antagelse om ingen duplikater og avvikende attributtdata.

`eier_id` determinerer alle kolonner relatert til eier:
`eier_id --> eier_navn, eier_adresse, eier_tlf`

Fordi: Samme som `kunde_id` og `eiendom_id`.

Kombinasjonen av `kunde_id`, `fra_uke`, `til_uke` determinerer `pris`, `eiendom_id` og `eier_id`.
`(kunde_id, fra_uke, til_uke) --> asd`

Fordi: Siden en kunde kun kan leie én eiendom om gangen, og en eiendom kun kan leies ut til én kunde om gangen, vil en spesifikk kunde i et spesifikt tidsintervall determinere prisen, eiendommen og eiendommens eier.

Kombinasjonen av `eiendom_id`, `fra_uke`, `til_uke` determinerer `pris`, `kunde_id` og `eier_id`.
`(eiendom_id, fra_uke, til_uke) --> asd`

Fordi: På samme måte som forrige, vil en spesifikk eiendom i et gitt tidsintervall ha en spesifikk kunde og eier.

Funksjonelle avhengigheter og BCNF

> BCNF = Boyd-Codd Normal Form

For å oppnå BCNF kan tabellen deles opp i tabellene Kunde, Eiendom, Eier og Utleie:

Kunde:

```
(kunde_id, kunde_navn, kunde_adresse, kunde_tlf)
```

- Kandidatnøkkel: `kunde_id`
- Funksjonelle avhengigheter: `kunde_id --> (kunde_navn, kunde_adresse, kunde_tlf)`
- Hvorfor: Isolerer alle kundedata i en egen tabell, hvor `kunde_id` unikt identifiserer hver enkelt kunde. Dette gjør det enklere å administrere databasen, og fjerner uregelmessighetene ved innsetting, sletting

og oppdatering nevnt tidligere.

Eiendom:

```
(eiendom_id, eiendom_adresse, eier_id)
```

- Kandidatnøkkel: `eiendom_id`
- Funksjonelle avhengigheter: `eiendom_id --> (eiendom_adresse, eier_id)`
- Hvorfor: Samler info om eiendommer i en egen tabell. Inkluderer `eier_id` slik at eiendommene har en tilhørende eier.

Eier:

```
(eier_id, eier_navn, eier_adresse, eier_tlf)
```

- Kandidatnøkkel: `eier_id`
- Funksjonelle avhengigheter: `eier_id --> (eier_navn, eier_adresse, eier_tlf)`
- Hvorfor: Som hos `kunde`, samles her alle informasjon om eier i en egen tabell. Uregelmessigheter i forbindelse med oppdatering, sletting og innsetting elimineres.

Utleie:

```
(kunde_id, eiendom_id, fra_uke, til_uke, pris, eier_id,)
```

- Kandidatnøkkel: `(kunde_id, eiendom_id, fra_uke, til_uke)`
- Funksjonelle avhengigheter: `(kunde_id, eiendom_id, fra_uke, til_uke) --> pris`
- Hvorfor: Tabellen kobler sammen kunde og eiendom via et leieforhold i et tidsintervall. Det spesifikke leieforholdet peker på prisen. Vi trenger ikke ha med eier her, siden eierskap spores via tabellen "Eiendom"

1NF, 2NF, 3NF

Det er mulig å løse oppgaven via prosessen 1NF --> 2NF --> 3NF. Hvis jeg har skjønt dette riktig vil en tabell som oppfyller kravene til BCNF, per definisjon, inneholde kravene til 3NF, som inneholder kravene til 2NF, som igjen inneholder kravene til 1NF. Altså hvis en tabell oppfyller BCNF vil den måtte oppfylle alle "underkravene" 1NF til 3NF, og kan da gjennomgå alle trinnene.

Del 2 - Transaksjoner

Teorispørsmål

> Hvilke typer låser har databasesystemene?

- **Exclusive (X):** Låser av en side eller rad, slik at den er reservert for den som holder låsen frem til transaksjonen er fullført og slipper låsen. Som Serializable.
- **Shared (S):** Låser av en side eller rad slik at den kun er tilgjengelig for lesing. Så lenge låsen er aktiv er skriving ikke tillatt. Flere klienter/transaksjoner kan legge inn lås, slik at flere kan holde en read-only lås.
- **Update (U):**

- **Intent (I):**
- **Schema (Sch):**
- **Bulk Update (BU):**

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

Fordi det låser oppføringer fra endringer før de blir committed, hvilket gjør at man ikke kan jobbe/lese flere på samme tabell. Dette kan være ønskelig å gjøre om man vil ha raskere oppdteringer, OG vet hva man gjør. Deadlocks kan også være et problem med Serializable, hvis flere jobber samtidig. Mer effektivitet og parallele operasjoner.

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

Den første (si Klinet 1) som bruker `select sum(saldo)` vil låse av ressursen for den andre (Klient 2), som blir nødt til å vente til K1 enten committer eller på annen måte slipper låsen.

> Hva er to-fase-låsing?

En metode for å garantere Serializable, ved å kjøre låsing og opplåsing i to faser.
Grow phase: Kan legge inn låser på dataelementer, men kan ikke slippe noen låser.
Shrink phase: Kan bare slippe eksisterende låser, men ikke legge inn nye.

Altså må transaksjonene først låse, og så låse opp, i sekvens. Dermed blir transaksjonene mer forutsigbare, selv om dette ikke garanterer at det ikke oppstår Deadlocks.

> Hvilke typer samtidighetsproblemer (de har egne navn) kan man få ved ulike isolasjonsnivåer?

- Lost update and dirty write
- Dirty read
- Non-repeatable read or fuzzy read
- Non-Repeatable Read
- Phantom Reads

https://blog.idera.com/database-tools/isolation-levels-101-concurrency-issues
https://www.sqlshack.com/concurrency-problems-theory-and-experimentation-in-sql-server/
https://www.w3computing.com/sqlserver2012/isolation-levels-concurrency-problems/

> Hva er optimistisk låsing/utførelse? Hva kan grunnen til å bruke dette være?

Databasen blir låst opp for endringer

> 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?

Man risikerer å låse ut andre transaksjoner, hvilket gjør at det blir ineffektivt for flere å jobbe mot samme database/tabell/datasett. I tillegg

Oppgave 1

1. isolasjon

- Klient 1 setter isolasjonsnivået til Read Uncommitted. Dette er laveste nivå, og betyr at K1 kan lese data som er midt i en tranaksjon, men ikke committed.
- Klient 2 setter isolasjonsnivået til Serializable, som er det høyeste nivået. Det gjør at den låser tabellen for endringer.

2. Begge klienter starter tranaksjonen,
3. Klient 2 gjør en `select * from konto where kontonr=1;`, hvilket låser oppføringen.
4. Klient 1 utfører samme operasjon, `select * from konto where kontonr=1;`.
5. Klient 2 utfører en oppdatering av saldo, `update konto set saldo=1 where kontonr=1;`. Klient 2 ser denne operasjonen, siden den ikke har blitt committet.
6. Klient 2 committer sin transaksjon. Klient 1 står i kø.
7. Klient 1 committer sin transaksjon, som blir lagret i databsen.

Hvis Klient 2 hadde brukt et annet isolasjonsnivå:

- **Read Uncommitted:** Begge klientene ville kunne lese og skrive data, fordi dette nivået ikke setter noen form for lås på skrivetilgang. Alle kan endre, selv om andre har pågående tranaksjoner.
- **Read Committed:** Klient 2 vil bare kunne se committed data. K2 vil låse oppføringen under `select`-operasjonen, men bare for den tiden det tar å lese. Etter at lesingen er utført, er den frigjort, og da kan Klient 1 kjøre `select` og `update` på den.
- **Repeatable Read:** Klient 2 vil låse oppføringen med `select`, og ikke tillate noen endringer på oppføringen før den har blitt committed. Det låser slik at om man kjører en read på denne flere ganger, vil det være samme data der, frem til en eventuell endring har blitt committed.

> **Read committed** is an isolation level that guarantees that any data read was committed at the moment is read. It simply restricts the reader from seeing any intermediate, uncommitted, 'dirty' read. It makes no promise whatsoever that if the transaction re-issues the read, will find the Same data, data is free to change after it was read.
>
> **Repeatable read** is a higher isolation level, that in addition to the guarantees of the read committed level, it also guarantees that any data read cannot change, if the transaction reads the same data again, it will find the previously read data in place, unchanged, and available to read.
>
> The next isolation level, **serializable**, makes an even stronger guarantee: in addition to everything repeatable read guarantees, it also guarantees that no new data can be seen by a subsequent read.
>
> https://stackoverflow.com/questions/4034976/difference-between-read-commited-and-repeatable-read-in-sql-server

Oppgave 2

2a)

- Klient 1 setter isolasjonsnivået til Read Uncommitted. Dette er laveste nivå, og betyr at K1 kan lese data som er midt i en tranaksjon, men ikke committed.
- Klient 2 setter isolasjonsnivået til Serializable, som er det høyeste nivået. Det gjør at den låser tabellen for endringer.
- K1 bruker `update` og setter saldoen på konto 1 til 1.
- K2 vil ikke kunne gjennomføre sin `update` på konto 1 før K1 har committed sin tranaksjon. Dette fordi K2 har nivå Serializable, som forhindrer den fra å lese uregistrerte endringer i dataene.
- K1 oppdaterer saldoen på konto 2 til 1.
- K1 comitter sine endringer. Da kan K2 gjennomføre oppdateringen på konto 1, som har stått på vent.
- K2 oppdaterer saldoen på konto 2 til 2, og committer sine endringer.

Resultatet:

- Saldo på konto 1 = 2
- Saldo på konto 2 = 2
- Altså er det Klient 2 sine endringer som blir stående.

Det blir slik fordi Klient 1 ikke låser oppføringene den jobber på (Read Uncommitted), og viser ikke de uncommittede endringene til en klient med nivå Serializable. Videre vil Serializable forhindre K2 fra å utføre endringer som kommer i konflikt med K1 før K1 har committed. Men etterpå er det fritt å skrive over det K1 la inn.

2b)

- Begge klientene er på nivå Read Uncommitted, hvilket betyr at de kan skrive og lese fritt, selv om den andre klienten ikke har committed endringene sine.
- K1 setter saldo på konto 1 til 1.
- K2 setter saldo på konto 2 til 2.
- K1 setter saldo på konto 2 til 1.
- K2 setter saldo på konto 1 til 2.

Resultat:

- Saldo på konto 1 = 2.
- Saldo på konto 2 = 1.

Forskjellen fra oppgave 2a er at K2 var satt til Serializable, og vi har ingen Commit-kommando. Dette forhindret den fra å utføre uppdateringer som ville påvirke K1. K2 kunne da ikke lese endringer fra K1 som ikke var committet.

Hvis man endret isolasjonsnivået vil man eliminere at de leses i "sanntid", men at man må vente på en committ. Hvis Klien 1 endres til serializable ville den måtte vente på Klient 2 før den kan utføre sin transaksjon. Omvent har vi eksempel på i oppgave 2a.
Hvis begge ble satt til Serializable, ville de bli stående i kø og vente på en commit som ikke kommer.

Oppgave 3

1. K1 = Read uncommitted. K2 = Serializable.

- Dvs. K1 kan her se alle endringer som blir gjort, selv om de ikke har blitt committed.

3. Klient 1 leser data `sum(saldo)` fra konto.
4. Klient 2 oppdaterer saldo med `saldo + 10` på konto 1.
5. Klient 1 leser uncommitted data fra `saldo`.
6. Klient 2 committer endringene.
7. Klient 1 leser summen på saldo.
8. Klient 1 committer lesingen.

Klient 1 ser her hele tiden de endringene som gjøres av Klient 2.

Om Klient 1 endret isolasjonsnivå:

Read Committed:

- Klient 1 ser kun committede endringer.
- Dvs. at ved steg 3 og 7 vil den se "uoppdatert" verdi, mens i steg 7 vil den se den nye verdien committed av Klient 2.

Repeatable Read: Leser kun committede endringer. I praksis samme resultat som med Read Committed.

Serializable: Det blir låsekonflikter mellom K1 og K2. En av tranaksjonene må gjøre en ROLLBACK slik at den andre kan fullføre.

Oppgave 4

Kode for å generere en enkel database/tabell:

```sql
CREATE TABLE konto (
    kontonr INT PRIMARY KEY,
    saldo INT
);
```

```sql
INSERT INTO konto (kontonr, saldo) VALUES (1, 100);
INSERT INTO konto (kontonr, saldo) VALUES (2, 200);
```

Scenario 1: Ingen phantom reads

Tid	Klient 1 (Repeatable Read)	Klient 2 (Read Committed)
1	Sett isolasjonsnivå til **repeatable read**	Sett isolasjonsnivå til **read committed**
2	Start transaksjon	Start transaksjon
3	SELECT * FROM konto WHERE saldo >= 100;	
4		INSERT INTO konto (kontonr, saldo) VALUES (3, 300);
5		COMMIT;
6	SELECT * FROM konto WHERE saldo >= 100;	
7	commit;	

I dette scenariet vil Klient 1 (Repeatable Read) ikke oppleve phantom reads. Det betyr at den andre `SELECT`-spørringen vil returnere de samme radene som den første, selv om Klient 2 har lagt til en ny rad med `saldo >= 100`.

Dersom isolasjonsnivået for Klient 1 endres til "Read Committed" og kjører samme scenarie, vil de resultere i en phantom read. Den andre `SELECT`-spørringen vil da inkludere den nye raden som Klient 2 har lagt til.

Scenario 2: Phantom reads

I dette tilfellet, vil Klient 1 oppleve en phantom read. Under den første `SELECT`-spørringen vil Klient 1 se de to opprinnelige kontoene (kontonr 1 og 2). Etter at Klient 2 har lagt til en ny konto (kontonr 3) og utført et COMMIT, vil Klient 1 sin andre SELECT-spørring også inkludere denne nye kontoen. Dette er et eksempel på en phantom read, hvor antallet rader som oppfyller en gitt betingelse endrer seg innenfor en enkelt transaksjon.

Tid	Klient 1 (Read Committed)	Klient 2 (Read Committed)
1	Sett isolasjonsnivå til Read Committed	Sett isolasjonsnivå til Read Committed
2	Start transaksjon	Start transaksjon
3	SELECT * FROM konto WHERE saldo >= 100;	
4		INSERT INTO konto (kontonr, saldo) VALUES (3, 300);
5		COMMIT;
6	SELECT * FROM konto WHERE saldo >= 100;	
7	COMMIT;	

Alternativ 2: Emil

```
CREATE TABLE ordre (
    id INT PRIMARY KEY,
    produkt VARCHAR(255),
    antall INT);
```

Tid	Klient 1 (Read Committed)	Klient 2 (Read Committed)
1	SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;	Sett isolasjonsnivå til Read Committed
2	BEGIN;	BEGIN;
3	SELECT * FROM ordre;	
4		INSERT INTO ordre (id, produkt, antall) VALUES (1001, 'NyttProdukt', 10);
5		COMMIT;
6	SELECT * FROM ordre;	
7	COMMIT;	

//end