Database programmerings tips

Denne artikel vil introducere nogle problem stillinger med flere samtidige brugere, som man skal tænke på, når man udvikler en applikatiom som bruger en database.

Den forudsætter noget kendskab til SQL og programmering.

Skrevet den **14. Feb 2010** af **arne_v** I kategorien **Databaser / Generelt** | ★★★☆☆

Historie:
V1.0 - 08/07/2006 - original
V1.1 - 12/02/2010 - smårettelser

Forudsætning

Du kan nok med fordel lige skimme denne artikel først:
http://www.eksperten.dk/guide/991

Indledning

Når der er flere samtidige brugere af en database, så er der visse ting man skal tænke over i sit applikations design.

Og selvom der ikke umiddelbart er planer om at bruge applikationen i flerbruger sammenhæng, så er det stadig et sundt princip lave sin kode, så den er sikker i en fler bruger sammenhæng. Fordi kode har det med at blive videreudviklet og genbrugt.

De efterfølgende eksempler vil primært bruge pseudo kode med SQL til at vise pointerne.

Det burde være nemt at implementere denne i enhver database og ethvert applikations sprog.

Men der er små forskelle i SQL dialekter som gør at nogle ting skal laves lidt forskelligt i forskellige databaser.

Nogen gange vil jeg eksplicit nævne nogle af de mere kendte databaser.

Den klassiske fejl som man *aldrig* må lave

For at illustrere problem stillingen med flere samtidige brugere, så lad os tage den helt klassiske fejl.

Man indsætter i en tabel som selv genererer en primary key og den skal man så bruge til at indsætte i en tabel som foreign key.

```
execute update "INSERT INTO brugere (brugernavn, rigtigtnavn) VALUES (brugernavnsfelt,
rigtigtnavnsfelt)"
newid = execute query "SELECT MAX(id) FROM brugere"
execute update "INSERT INTO statistik (brugerid, n) VALUES (newid, 0)"
```

Og umiddelbart virker den logik jo meget fornuftig.

Og en hurtig unit test afslører heller ikke nogen problemer.

Der er imidlertid et alvorligt samtidigheds problem.

Og et problem som typisk først vil vise sig ved høj belastning (læs: det mest uheldige tidspunkt !).

Forestil os to samtidige brugere.

```
bruger 1                                bruger 2
--------                                --------
INSERT INTO brugere ...
newid = SELECT MAX(id) FROM brugere
INSERT INTO statistik ...
                                        INSERT INTO brugere ...
                                        newid = SELECT MAX(id) FROM brugere
                                        INSERT INTO statistik ...
```

er ikke noget problem.

```
bruger 1                                bruger 2
--------                                --------
INSERT INTO brugere ...
                                        INSERT INTO brugere ...
newid = SELECT MAX(id) FROM brugere
                                        newid = SELECT MAX(id) FROM brugere
INSERT INTO statistik ...
                                        INSERT INTO statistik ...
```

vil derimod være fatal idet bruger 1's SELECT MAX faktisk vil hente værdien af bruger 2's INSERT.

Medmindre koden gør noget specielt, så vil man ikke kunne antage noget som helst om hvordan kode i forskellige processer/tråde udføres i forhold til hinanden.

For lige at gøre eksemplet færdigt, så er den korrekte løsning:

MS Access:

```
execute update "INSERT INTO brugere (brugernavn, rigtigtnavn) VALUES (brugernavnsfelt,
```

```
rigtigtnavnsfelt)"
execute update "INSERT INTO statistik (brugerid, n) VALUES (@@IDENTITY, 0)"
```

MS SQLsServer:

```
execute update "INSERT INTO brugere (brugernavn, rigtigtnavn) VALUES (brugernavnsfelt,
rigtigtnavnsfelt)"
execute update "INSERT INTO statistik (brugerid, n) VALUES (SCOPE_IDENTITY(), 0)"
```

MySQL:

```
execute update "INSERT INTO brugere (brugernavn, rigtigtnavn) VALUES (brugernavnsfelt,
rigtigtnavnsfelt)"
execute update "INSERT INTO statistik (brugerid, n) VALUES (LAST_INSERT_ID(), 0)"
```

hvor tricket er at funktionerne @@IDENTITY, SCOPE_IDENTITY() og LAST_INSERT_ID() returnerer den sidste genererede primary key for connection. Og fordi det er per connection så virker det netop i flerbruger sammenhæng også.

Oracle og visse andre databaser bruger et helt andet approach baseret på sequences.

Generalisering

Problemet i sidste afsnit var nemt at løse. Desværre findes der andre problemer som er vanskeligere at løse.

Her er et eksempel:

```
saldo = execute query "SELECT saldo FROM konto WHERE kontonr=xxxx"
if saldo >= pris then
   execute update "UPDATE konto SET saldo = saldo - pris WHERE kontonr=xxxx"
endif
```

Den som har studeret eksempelt i sidste afsnit grundigt kan se, at hvis to brugere begge får udført SELECT inden UPDATE så kan saldo faktisk gå i negativ.

Og der er ikke nogen smart funktion der kan redde os her.

Applikations tråd synkronisering

En måde at løse problemet på er at få applikationen til at sikre at der kun er en tråd som kører den kode samtidigt.

Java:

```
synchronized(sharedobject) {
   // kode
}
```

C#:

```
lock(sharedobject)
{
   // kode
}
```

Det er meget nemt.

Men der er også nogle problemer ved løsningen:

* det virker kun med en enkelt multithreaded applikation
  hvilket i praksis betyder en web applikationer som kun
  kører på en server og som har eksklusiv adgang til de
  pågældende data

* performance er ikke specielt god

Så derfor vil man normalt foretrække smartere løsninger som
bruger databasen.

Transaktioner

Inden vi går videre skal vi lige repetere lidt basal viden om
transaktioner.

Transaktioner er grundliggende en bundtning af flere SQL
opdateringer i en enkelt transaktion hvor enten udføres alle
opdateringer eller så udføres ingen af dem.

Problemet illustreres ved:

```
UPDATE konto SET saldo = saldo - pris WHERE kontonr = a
UPDATE konto SET saldo = saldo + pris WHERE kontonr = b
```

Hvad sker der hvis applikationen crasher mellem de 2 SQL sætninger ?

Tja - penge er forsvundet lige ud i den blå luft. Og bytter man om
på de to sætninger er problemet lige så galt, så vil et crash imellem
dem betyde at der er opstået penge ud af den blå luft.

Løsningen er at bundte de to opdateringer i en transaktion.

```
BEGIN
UPDATE konto SET saldo = saldo - pris WHERE kontonr = a
UPDATE konto SET saldo = saldo + pris WHERE kontonr = b
COMMIT
```

I nogle databaser skal man eksplicit disable auto commit fordi ellers sker der en automatisk commit efter hver opdatering.

COMMIT udfører transaktionen. Hvis man fortryder skal man bruge ROLLBACK.

Husk at MySQL med MyISAM tabeller ikke understøtter transaktioner. Stort set alle andre databaser og MySQL med InnoDB tabeller understøtter transaktioner.

Transaction isolation level

Selvom man bruger transaktioner er der stadig potentielle muligheder for samtidigheds problemer.

Omfanget af disse afhænger af det valgte transaction isolation level.

Standard transaction isolation levels er:

Uncommitted Read = ingen isolering overhovedet
Committed Read = man kan kun læse data som er committed - ikke nogle midlertidige data som kan blive rollbacked
Repeatable Read = de data man har læst kan ikke ændres førend transaktionen er færdig
Serializable = som repeatable read men derudover vil der heller ikke kunne tilføjes nye data førend transaktion er færdig

Som det ses vil brug af transaction isolation level repeatable read eller serializable løse mange samtidigheds problemer herunder eksemplet ovenfor hvor vi skal undgå at konto saldo bliver negativ.

Og vel at mærke på en måde som virker med flere applikationer eller samme applikation kørende på flere servere.

Der er dog stadigt et vist performance overhead, da databasen skal låse de rækker/sider som man bruger.

Nærlæs din databases dokumentation omkring supporterede transaction isolation levels og default transaction isolation level.

Det er kritisk for data integritet.

Explicit locking

Udover at lade database selv lave låse udfra SQL sætningerne og transaction isolation level så har de fleste databaser også muligheder for at styre låsning via eksplicitte angivelser i SQL sætningerne.

Vores tidligere eksempel kunne f.eks. i visse database løses som:

```
BEGIN
saldo = execute query "SELECT saldo FROM konto WHERE kontonr=xxxx FOR UPDATE"
if saldo >= pris then
   execute update "UPDATE konto SET saldo = saldo - pris WHERE kontonr=xxxx"
endif
COMMIT
```

som vil virke uanset transaction isolation level fordi vi eksplicit
låser den record vi selecter.

Fordelen ved denne metode er at den normalt vil være mindre
ressource krævende for databasen, da vi eksplicit fortæller den
hvad den skal låse.

Ulempen er at SQL koden ikke er umiddelbart portabel da syntaxen
for den slags varierer mellem forskellige databaser.

Long time locking

Både implicitte låse via transaction isolation level og eksplicitte
låse via SQL syntax som f.eks. SELECT FOR UPDATE er short term låse.

De er beregnet til at blive holdt i databasen i millisekunder.

Og der er normalt timeouts på dem som creater exceptions som
applikationen så skal kunne håndtere.

Der eksisterer nogle helt andre samtidigheds problemer hvor
tiden tælles i minutter eller timer.

Et typisk scenarie er:

```
SELECT navn,adresse,postnr FROM kartotek WHERE id=x
brugeren retter i et skærm billede i 5 minutter udfra noget papir
UPDATE kartotek SET adresse=y,postnr=z WHERE id=x
```

Og det kan også give samtidigheds problemer.

```
bruger 1                                bruger 2
--------                                --------
SELECT
ret
UPDATE
                                        SELECT
                                        ret
                                        UPDATE
```

virker

```
bruger 1                                bruger 2
--------                                --------
SELECT
                                        SELECT
ret
                                        ret
UPDATE
                                        UPDATE
```

så overskriver bruger 2 de rettelser bruger 1 lavede.

Og det kan ikke løses ved nogle af de tidligere beskrevne teknikker, da database låse forlængst ville time ud (eller performance ville gå totalt ned).

Long time pessimistic locking

Ideen er at sætte en markering af om en række er i brug.

Tilføj en kolonne inuse af type BOOLEAN i tabellen og brug den til at markere om en række er i brug.

(det er også muligt at bruge en separat lock tabel, men jeg vil holde mig til den simple løsning her)

Logikken kan implementers på flere forskellige måder.

Med SELECT + UPDATE i en transaktion med passende højt transaction isolation level:

```
BEGIN
inuse = execute query "SELECT inuse FROM kartotek WHERE id=x"
if inuse then
  informer bruger om at det ikke er muligt at editere
else
  execute update "UPDATE kartotek SET inuse=TRUE WHERE id=x"
  execute query "SELECT navn,adresse,postnr FROM kartotek WHERE id=x"
endif
COMMIT
```

brugeren editerer data

```
UPDATE kartotek SET adresse=y,postnr=z,inuse=FALSE WHERE id=x
```

Med kun UPDATE hvis man kan få returneret antal modificerede rækker:

```
rowsmodified = execute update "UPDATE kartotek SET inuse=TRUE WHERE id=x AND NOT inuse"
if rowsmodified = 0 then
  informer bruger om at det ikke er muligt at editere
else
  execute query "SELECT navn,adresse,postnr FROM kartotek WHERE id=x"
endif
```

brugeren editerer data

```
execute update "UPDATE kartotek SET adresse=y,postnr=z,inuse=FALSE WHERE id=x"
```

Ulemperne ved pessimistic locking er:

* der er overhead ved den ekstra opdatering

* hvis en bruger henter data man aldrig gemmer data igen så forbliver inuse flaget
  sat og man har brug for en cleanup process

Long time optimistic locking

Ideen er at checke om data er blevet ændret af en anden bruger siden man selv hentede
data når man gemmer sin rettelse.

Tilføj en kolonne version af type INTEGER i tabellen og brug den til tælle op
for ændringer.

Logikken kan implementers på flere forskellige måder.

Med SELECT + UPDATE i en transaktion med passende højt transaction isolation level:

```
execute query "SELECT navn,adresse,postnr,version FROM kartotek WHERE id=x"
myversion = result[version]
```

brugeren editerer data

```
BEGIN
currentversion = execute query "SELECT version FROM kartotek WHERE id=x"
if currentversion = myversion then
    UPDATE kartotek SET adresse=y,postnr=z,version=version+1 WHERE id=x
else
  informer bruger om at det ikke er muligt at gemme data
endif
COMMIT
```

Med kun UPDATE hvis man kan få returneret antal modificerede rækker:

```
execute query "SELECT navn,adresse,postnr,version FROM kartotek WHERE id=x"
myversion = result[version]
```

brugeren editerer data

```
rowsmodified = execute update "UPDATE kartotek SET adresse=y,postnr=z,version=version+1 WHERE
id=x AND version=myversion"
if rowsmodified = 0 then
  informer bruger om at det ikke er muligt at gemme data
endif
```

Performance er bedre ved optimistic locking end ved pessimistic locking.

Ulemperne ved optimistic locking er:

* den bruger der forsøge at gemme sidst skal starte forfra med sine
  rettelser

Derfor anbefales optimistic locking normalt kun hvis sandsynligheden
for at to vil forsøge at rette i samme data er tilpas lille.

Visse database har indbygget en datatype for den slags versions nummer.

Kommentar af arriva d. 11. Jul 2006 | 1

Giver en god indsigt i de problemer der kan opstå ved flerbrugersystemer hvis man ikke tænker sig om.

Kommentar af trp79 d. 12. Jul 2006 | 2

Endnu en informativ og let læselig artikel - bliv endelig ved i den dur Arne.
Mvh trp79

Kommentar af qtax87 (nedlagt brugerprofil) d. 21. Aug 2006 | 3

God artikel, dem må vi have nogen flere af fra arne_v.

Kommentar af imago-dei d. 19. Jul 2006 | 4

Fin artikel. Jeg vil dog tilføje at flere databaser (i hvert fald MSSql og Oracle) kan udlevere en timestamp eller en rowversion, som er unik for hver record, og bliver opdateret af basen ved insert og update. På den måde kan du lave concurrency check (det som du kalder "Long time optimistic locking" uden at have besværet med at selv opdatere rowversion.

Kommentar af trer d. 09. Jul 2006 | 5

Rigtig fin artikel - og gid alle udviklere ville læse den :-)

Kommentar af speedpete d. 01. Nov 2007 | 6

Kommentar af sharon (nedlagt brugerprofil) (nedlagt brugerprofil) d. 11. Jul 2006 | 7

kanon god artikel for dem der er ny i SQL, så man ikke laver en masse basale fejl

//sharon

Kommentar af emmo d. 07. Apr 2007 | 8

Rigtig god artikel, ja dem vil vi gerne se nogle fler af.