UNIVERSITETET I OSLO
Institutt for informatikk

Studie av isolasjonsnivåer i Oracle

Masteroppgave
60 studiepoeng

Vibeke Stoltenberg

14. des. 2009

1. Forord

En stor takk til min veileder Ragnar Normann som har kommet med konstruktiv kritikk og innspill, og holdt meg i gang med oppgaven. Og en like stor takk til min gode venninne Heidi Løken for hjelp med korrekturlesing, og som hjalp meg med å holde humøret oppe.

2. Innhold

3. Innledning

Fra datamaskinenes spede barndom og fram til i dag har behovet for og etterspørselen etter lagring og bruk av data bare økt. Datamaskiner benyttes på stadig nye måter og blir en stadig større del av samfunnet vårt og våre daglige liv. Mer og mer data lagres og aksesseres hver dag, og behovet for samtidig bruk av disse dataene blir bare større og større. I dag vil det være utenkelig å drive f.eks. en nettbank uten noen form for samtidighetskontroll, dvs. isolering, mellom dataene.

Det finnes imidlertid forskjellige grader av isolering som kan benyttes mellom dataene, og man må spørre seg: Hvor strenge regler skal vi legge på samtidighetskontrollen?

Hva skal tillates av samtidig bruk av dataene, og hva kan tillates?

Dette avhenger naturligvis av hva slags data det er snakk om og hva de skal brukes til. I denne oppgaven vil jeg se nærmere på hvordan de forskjellige isolasjonsnivåene påvirker gjennomstrømningen i databasen.

Det er viktig å finne en passende balanse mellom regler og ytelse i forbindelse med isolasjonsnivåer i databasen. Med denne oppgaven ønsker jeg å bidra til å si noe om denne balansen.

Videre i denne oppgaven tar jeg for meg litt terminologi og går igjennom de verktøyene jeg har brukt for å kunne teste på samtidighet og gjennomstrømning av data. Så tar jeg for meg prosjektbeskrivelsen med de feil og problemer som oppsto underveis. Resultatene har jeg presentert i et eget kapittel før jeg oppsummerer mot slutten.

Oversettelser

Enkelte ord i denne masteroppaven vil stå på engelsk, fordi jeg ikke har funnet et god nok norsk oversettelse for ordet. Feks commit/committed = transaksjonen fullfører korrekt**.**

4. Terminologi

4.1 Transaksjoner

Mot større databaser brukes ofte transaksjoner i stedet for å kjøre enkeltspørringer. En transaksjon kan ses på som et program med en samling med spørringer som henger sammen. Eks: Fra en filmdatabase henter man ut informasjon om hvilke filmer en skuespiller har spillt i.

Transaksjoner består av er en eller flere databaseoperasjoner som en lese (read-only)- eller skriveoperasjon (oppdatering, sletting, og endring), som sammen endrer innholdet i en database på en konsistent måte. En transaksjon har alltid en start-operasjon, og avslutter enten med en commit- eller abortoperasjon.

Commit:

Hvis transaksjonen går igjennom og er vellykket, så blir transaksjonen committed. Alle endringer blir lagret permanent i databasen. Commit signaliserer at transaksjonen er vellykket.

Abort:

Hvis derimot transaksjonen av en eller annen grunn feiler, så aborterers transaksjonen og den anses om mislykket. Det vil si at ingenting lagres, og databasen rulles tilbake (rollback) til den tilstanden den var i før transaksjonen startet. Alle data i databasen vil så være som før transaksjonen startet.

Transaksjon:

En transaksjon ti er en sekvens, eller streng ordning [1]<, av lese- og skriveoperasjoner på objekter fra én eller flere databaser, sammen med et sett av start- og commit-operasjoner. Alle startoperasjoner kommer før alle lese- og skriveoperasjoner i rekkefølgen < og alle commit-operasjoner etterfølger alle andre operasjoner med tanke på <. [8]

Historie:

En historie for transaksjonene $T = \{t1, \ldots\}$ er en sekvens, eller strengordning <, av unionen av alle operasjoner i transaksjonene i T slik at rekkefølgen av operasjonene innad i transaksjonene blir ivaretatt. [8]

En plan er et prefiks av en historie.

[1] Streng ordning: Kjøres det transaksjoner mot flere databaser, vil det være en commit-operasjon for hver database. Det samme gjelder for abort; ved transaksjoner mot flere databaser vil det være en abort i hver av databasene. Men jeg skal kun operere i en database, derfor vil jeg kun ha bruk for enten commit eller abort.

4.2 ACID egenskapene

Hensikten med et DBMS (DataBase Management System) er å kunne kjøre flere transaksjoner samtidig i en database. Men når flere transaksjoner kjøres samtidig, så oppstår det lett feilsituasjoner.

En feil som kan oppstå er når to transaksjoner aksesserer samme objekt, der den ene oppdaterer objektet rett før eller etter at den andre transaksjonen har aksessert det samme objektet. Inkonsistent database blir resultatet, Inkonsistens kommer av at den ene transaksjonen er blitt utført på grunnlag av uriktige verdier. For å unngå alle slike feilsituasjoner må samtidighetskontrollen sørge for at resultatet blir det samme som ved en seriell historie. Dvs. vil si at kun en transaksjon blir utført om gangen, der hver transaksjon blir startet og avsluttet før neste transaksjon starter. Slike historier kalles serialiserbare.

Samtidighetskontrollen i et DBMS er også pålagt å følge ACID-kravene for å sikre at en database alltid er konsistent.

ACID-kravene

ACID er et akronym for Atomicity, Correctness, Isolation og Durability, og er fire krav for å sikre at operasjoner som utføres i databaser er pålitelige.

- atomær (eng: atomicity)

En operasjon sies å være atomær hvis enten alle operasjoner eller ingen operasjoner for en transaksjon utføres. Feiler en av operasjonene til transaksjonen, så blir hele transaksjonen satt som feil. For at en transaksjon skal anses som vellykket og committed, må alle operasjoner for transaksjonen være vellykkede.

- konsistent (eng: consistency)

En transaksjon tar databasen fra en konsistent tilstand til den neste, og sikrer at databasen er i en konsistent tilstand ved at kun gyldige transaksjoner blir committed. Databasen er dermed konsistent både før og etter transaksjonen, men kan være inkonsistent mens operasjonene i den utføres i en transaksjon. Konsistensen sikres med integretetsregler som primærnøkler, fremmednøkler og kandidatnøkler.

- isolasjon (eng: isolation)

Resultatet av en transaksjon skal være usynlig for andre transaksjoner inntil transaksjonen har committed. I-en i ACID står for Isolasjon og krever full isolasjon.

Målet med isolasjon er å kunne kjøre flere synkroniserte (samtidige) transaksjoner uten at de lager problemer for hverandre.

- Varighet (eng: durability)

Med en gang 'commit' er foretatt på en transaksjon, vil oppdateringene i databasen være varige, inntil en annen transaksjon oppdaterer dem igjen. De lagres på disk, slik at de er sikret å overleve ved evnt. feil.

For å sikre at dataene blir persistente og overlever en systemfeilhendelse, må DBMS'et ha gjenopprettingsmuligheter (backup, systemlog, redo log, sjekkpunkter i databasen). En gjenopprettelsesmodul i databasen tar seg av dette.

4.3 Isolasjonsnivåer

Isolasjonsnivå brukes for å bestemme hvor isolert en transaksjon skal kjøres fra andre transaksjoner, og dermed hvilke låser som kan oppstå. Transaksjoner vil da ha begrenset tilgang til å se hva andre transaksjoner gjør, og hva slags resultater det gir. Jo strengere nivå som settes, jo mer koster det å sikre serialiserbarhet. Et høyt isolasjonsnivå gjør at mer data blir isolert tidligere, men dette kan redusere grad av samtidighet da andre transaksjoner må vente på tilgang til isolerte data.

Parameteren ISOLATION angir isolasjonsnivået som programmet bruker når det arbeider med relasjonsdatabasen: 'set transaction isolation level <isolation level>'

Det eksisterer fire forskjellige isolasjonsnivåer, der det strengeste går på full isolering, hvor transaksjonen har inntrykk av at den kjører helt alene i systemet selv om det er flere samtidig. De andre transaksjonene får i tillegg ikke får lov til å lese eller oppdatere de data som er endret og som ikke er committed ennå.

Ved bruk at isoleringsegenskapene kan man unngå feilsituasjoner som:

Lost Updates:

Når transaksjoner gjør oppdateringer på samme objekt, kan det være at de skriver over hverandres resultater, der den siste oppdateringen vil være den som blir gyldig, mens den første oppdateringen vil forsvinne.

Cascading Abort:

Cascading abort er det som skjer når en transaksjon leser et objekt som er endret av en annen transaksjon som så aborterer, slik at også denne transaksjonen må abortere.

ANSI/ISO SQL STANDARDEN har 4 isolasjonsnivåer (4 ulike nivåer av serialiserbarhetskrav):

Hvis vi har fire like transaksjonsscenarios, med samme arbeid og samme input på de fire isolasjonsnivåene vil vi avhengig av isolasjonsnivå, vil få forskjellig resultat ved en gjennomkjøring av transaksjonene.

De 4 isolasjonsnivåene er:

READ-UNCOMMITTED

READ COMMITTED

REPEATABLE READ

SERIALIZABLE

Isolasjonsnivåene i seg selv sikrer ikke konsistente data i en database, men det at en database følger ACID-kravene gjør det.

Isolasjonsnivåer i Oracle:

Det er verdt å merke seg at ikke alle DBMS tilbyr alle isoleringsnivåene.

Oracle bruker kun to av isolasjonsnivåene:

READ COMMITTED

SERIALIZABLE

Read Uncommitted – En plan *s* kjøres med isolasjonsnivået *read uncommitted* dersom skrivelåser settes og slippes etter S2PL-regelen. Ingen leselåser er nødvendig. [8]

Dette isolasjonsnivået tillater dirty reads, noe som vil si at transaksjonene kan lese data som ikke er committed.

Default for transaksjoenen er imdlertid ***nonblocking reads.*** Det vil si at Oracle sjekker om data er endret og bryr seg ikke om disse dataene på nåværende tidspunkt er låst (noe som impliserer at data er endret). Oracle vil i så tilfelle hente tilbake den gamle verdien fra rollback-segmentet og fortsette videre til neste blokk av data. Dvs. at det brukes snapshot som er tatt tidligere på forskjellige tidspunkter, noe som gir lese-konsistente queries og non-blocking queries. Spørringer som utfører lese-consistente operasjoner er select, insert, update, merge og delete. Unntaket er insert som ikke gjøre en lese-operasjon, men som kun består av en skriveoperasjon.

Non-blocking queries vil si spørringer/transaksjoner som ikke blir isolert fra hverandre.

Read Committed – en plan *s* kjøres under isolasjonsjonsnivået *read committed* dersom skrivelåser settes og slippes etter S2PL-reglene, og leselåser er av kort varighet – dvs leselåsene slippes idet den aktuelle leseopersjonen avsluttes [8] Read committed er muligens den mest brukte isoleringsnivået for databaser, og er default innenfor Oracle Databaser.

Transaksjonene under Read Commited kan kun lese data som er blitt committed i databasen, og isolasjonsnivået sikrer mot dirty reads.

Det kan ha nonrepeatable reads. Dette er et fenomen som leser opp igjen samme rad, noe som kan resultere i forskjellig svar i samme transaksjon da resultatet kan ha endret seg i mellomtiden. Det kan også ha phantoms read, et fenomen hvor nylig oppdaterte og committede rader blir synlig for en spørring selv om de ikke var synlige tidligere i den transaksjonen.

Repeatable Read – en plan *s* som kjøres under isolasjonsnivået *repeatable read* dersom skrivelåser og leselåser på dataobjekter settes og slippes etter S2PL-reglene. Leselåser på predikater er derimot av kort varighet, og slippes i det predikatet er lest [8].

Predikater er et sett med verdier om inngår i et logisk uttrykk, feks et 'where'-uttrykk i SQL.

Målet for repeatable read er å tilby et isoleringsnivå som gir konsistente og korrekte svar, og forhindrer lost updates.

Resultatet fra en spørring må være konsistent med respekt til et punkt i tid.

Ved bruk at multiversioning i Oracle Database, så får man svar som er konsistente ved starten av spørringer.

Serializable -- en plan *s* kjøres under isolasjonsnivået *serializable* dersom den kan genereres av S2PL-protokollen [8]

Serializable er et isolasjonsnivå som settes på transaksjoner. Det vil føles som om transaksjonene blir utført en om gangen dvs. seriell, selv om det er flere samtidige transaksjoner.

Serialiserbarhet (serializable) er det isolasjonsnivået som har strengeste nivå på isolering.

For at en parallell utførelse av transaksjoner skal være korrekt, må den være serialiserbar, dvs den må være ekvivalent med en seriell utførelse av de samme transaksjonene.

I tillegg til de 4 definerte SQL isolasjonsnivåene over fins nivået:

Read-only

Read only er ikke et isolasjonsnivå, men et nivå som kan settes på transaksjoner som kun skal gjøre lese-operasjoner. Insert, update, og delete er ikke tillatt på dette nivået.

'read only' transaksjonen er ekvivalent til repeatable read eller serializable transaksjoner som ikke kan utføre noen form for endringer i SQL.

Fordelen med dette nivået er at det ikke eksisterer noen konflikter mellom to read-only transaksjoner, så hvis man kun skal hente data er dette definitivt det beste nivået med tanke på ressursbruk og ytelse.

redo-logg

Redo logg kan brukes til å gjenopprette databasen fra en tidligere versjon, eller et såkalt sjekkpunkt (lagringspunkt), der man sist vet at databasen var konsistent. Redo loggen tar utgangspunkt i siste versjon og gjenoppretter alle verdier som er merket commit.

savepoints er en peker til en logglinje i en logg med alle operasjoner. Trengs det en restore, kjøres en undo for alle operasjoner som er utført etter det aktuelle savepointet. Slik at vi ruller kun tilbake til nærmeste lagringspunkt før vranglås oppsto (ikke helt tilbake), og fortsetter så derfra.

.

4.4 Samtidighetsfenomener

Når flere samtidige transaksjoner kjører mot en database, så er det viktig med en samtidighetskontroll (eng. concurrency control) for å koordinere transaksjonen for å forhindre forkludring og for å opprettholde databasen i en konsistent tilstand. Effekten vil bli den samme som om transaksjonene ble kjørt en og en (seriell).[16], og det er viktig at transaksjonene ikke forstyrrer hverandre.

Det er viktig med mekanismer for å ivareta samtidigheten og samtidig konsistensen i databasen. Målet er at transaksjonene skal ses som om de er utført i seriell utførelse, altså serialiserbar.

Støtte for de fleste databasesystemer er at man får transaksjonene kjappere igjennom, og får dermed bedre responstid og bedre system utnyttelse.

Samtidighet

Samtidighet henspiller på de problemstillingene som dukker opp når vi tar i betraktning at flere transaksjoner jobber mot databasen samtidig. Det er naturligvis viktig at disse transaksjonene ikke forstyrrer hverandre. Vi ser på tre kjente problemstillinger hva samtidighet angår*:*

Kjente problemstillinger vedr. samtidighet er:

- the lost update problem (tapt oppdatering)

- the uncommitted dependency problem (midlertidig oppdatering (“dirty read”)

- the inconsistent analysis problem (ukorrekt analyse (summeringsproblemet))

Tabell: Samtidighetsfenomener som kan oppstå ved de forskjellige isolasjonsnivåene:

Samtidighetsfenomener	Read Uncommitted	Read Committed	Repeatable Read	Serializable
Dirty Write	Nei	Nei	Nei	Nei
Dirty Read	Ja	Nei	Nei	Nei
Non-Repeatable Read	Ja	Ja	Nei	Nei
Lost Update	Ja	Ja	Nei	Nei
Read Skew	Ja	Ja	Nei	Nei
Write Skew	Ja	Ja	Nei	Nei
Read-Only Anomaly	Ja	Ja	Nei	Nei
Phantoms	Ja	Ja	Ja	Nei

Subsekvenser av operasjoner kan føre til at feilsituasjoner (phenomena) og faktiske feilsituasjoner (anomalies) oppstår.

Isolasjonsnivåene er definert ut i fra om de tillater hver av de følgende samtidighetsfenomenene (phenomen).

Med andre ord, Read uncommitted tillater alle de 3 fenomenene: dirty read, nonrepeatable read, og phantom read. Mens serializable ikke tillater noen av fenomenene.

At et **isolasjonsnivå** ikke tillater fenomener, er et sterkere forbud enn å ikke tillate en bestemt feil: samtidighetsfenomer behøver ikke alltid å føre til en feilsituasjon. [8]

En Dirty Read kan for eksempel være uproblematisk dersom begge transaksjonene faktisk comitter.

Samtidighetsfenomer trenger ikke alltid å føre til en feilsituasjon.

Samtidighetsfenomener:

DIRTY READ

FUZZY/NON-REPEATABLE READ

PHANTOM READ

Andre fenomener som kan oppstå:

LOST UPDATE

DIRTY WRITE

Beskrivelse av de 3 fenomenene brukt i tabellen over:

DIRTY READ – får lese uncommittede(dirty) data .

Dirty read oppstår når en transaksjon leser data som som en annen transaksjon har endret på, men hvor dataene ikke er committed og det er en sjanse for at den andre transaksjonen kan bli rullet tilbake. Da har transaksjonen lest data som er feil, da de ikke lenger eksisterer.

Dvs. Hvem som helst kan åpne filer som endres på/leses av en annen.

Et problem med dette er at dataintegriteten er ødelagt/brutt, fremmednøkler er endret, og primær nøkler er ignorerte.

NON-REPEATABLEREAD (Fuzzy) – prøver å lese en rad på to forskjellige tidspunkter. Problem er at data kan være endret i mellomtiden og/eller forsvunnet

Non-repeatable read eller fuzzy read oppstår altså når en transaksjon ser på de data den leste tidligere og finner ut at dataene er endret siden sist av en annen transaksjon.

PHANTOM READ – (fantom-lesing) kjører/execute en spørring (query) ved to forskjellige tidspunkter. Problemet kan være at rader kan være lagt til etter 1. kjøring, noe som vil ha effekt på resultatet.

Forskjellen fra non-repeatable read er at data du allerede har lest ikke er endret, men mer data tilfredsstiller spørringen enn tidligere.

Phantom read oppstår altså når en transaksjon kjører en spørring ved 2 forskjellige tidspunkter, og finner ut at mer data kommer frem ved andre gangs lesing. Dette kan komme av at en annen transaksjon har oppdatert tabellen og lagt til nye rader - og at disse nye radene tilfredsstiller where-betingelsen til den kjørende transaksjonen og derfor kommer med.

De resterende fenomenene er:

DIRTY WRITE – *t*1 oppdaterer *x*, deretter oppdaterer *t*2 *x* før *t*1 committer eller avbryter. Dersom én av transaksjonene forsøker å gjøre en rollback vil det være uklart hvilken verdi av *x* som er den korrekte — *t*2's endringer forsvinner dersom *t*1 gjør en rollback. Tillates Dirty Writes vil ikke gjenopprettelsessystemer fungere [8]

LOST UPDATE er når transaksjoner gjør oppdateringer på samme objekt. Det kan være at de skriver over hverandres resultater, der den siste oppdateringen vil være den som blir gyldig , mens det første oppdateringen vil forsvinne.

Lost update kan oppstå når en transaksjon prøver å lese data mens de blir oppdatert av en annen transaksjon.

Eks. Transaksjon A leser verdi a fra tabell A, og skal til å oppdatere denne. Samtidig leser transaksjon B verdi a fra samme tabell og updater denne etterfulgt av commit. Når transaksjon A da skal oppdatere sin a-verdi er denne ikke korrekt for verdien som faktisk er lagret i database, siden a har fått en ny verdi av transaksjon B.

Fenomener som kan oppstå:

Read Skew - oppstår når en transaksjon leser to forskjellige objekter med en integritetsregel seg i mellom, og objektene oppdateres av en samtidig transaksjon. [8]

Write Skew - kan oppstå ved en integritetsregel i en bank, hvor saldo på én enkelt konto kan bli negativ så lenge total saldo på alle kundens kontoer er positiv. [8]

Read-Only Anomaly – feilsituasjon som oppstår når en transaksjon leser to objekter med en integritetsregel mellom seg, hvorav det ene modifiseres av en samtidig transaksjon.

4.5 Snapshot isolation

Snapshot Isolation er en MVCC-protokoll, der Multiversjons samtidighetskontroll (Multiversion Concurrency Control – MVCC) er en metode som brukes for å tilby flere samtidige brukere tilgang til en database. Denne metoden består i at for alle skriveoperasjoner som blir utført, så lagres den gamle verdien med et tidsstempel. Dette tidsstemplet brukes av transaksjoner slik at de kan lese de verdiene som er endret innenfor transaksjonens levetid. Men transaksjonen kan ikke endre verdien hvis tidspunktet på verdien er et senere tidspunkt enn da transaksjonen startet, da det vil si at en annen transaksjon har endret den. Det betyr i så tilfelle at transaksjonen må rulles tilbake pga. feil og starte helt på nytt.

MVCC bruker enten tidsstempling fra systemklokke eller et transaksjonsnummer på transaksjoner. Vi får dermed versjonerte lese- og skrive-operasjoner som inngår i transaksjonen(e).

SI begrenser hva samtidige transaksjoner får "se" av hverandres operasjoner, og sier i tillegg at dersom to samtidige transaksjoner har disjunkte skrivesett, må én av dem avbrytes.

En transaksjon som utføres under Snapshot Isolation leser alltid data fra et commit-snapshot som implementeres ved å tidsstemple alle transaksjoner ved oppstart og commit.

Et commit-snapshot for en transaksjon t'i vil derned være databasens "siste" konsistente tilstand før t's første operasjon.

Når en transaksjon leser fra et commit-snapshot, vil den aldri kunne se "midlertidige" eller delvise data fra andre, samtidige transaksjoner.

To transaksjoner regnes som samtidige dersom transaksjonsperiodene deres overlapper.

Snapshot Isolation er attraktiv fordi den tilbyr et isolasjonsnivå som unngår mange av de vanligste samtidighetsfenomenene, og både Oracle, postgresSQl og Microsoft SQL Server har impplementert en versjon av den [8]

Oracles implementasjon av Snapshot Isolation benytter en versjon av «first committer wins» som Østby har kalt «first updated wins». Istedet for å kontrollere om samtidige transaksjoner oppdaterer de samme dataene ved commit, gjøres denne sjekken på oppdateringstidspunktet.

3 ting som slipper i gjennom:

-write skew - 2 transaksjoner som leser hvert sitt element og skriver det elementet som de ikke har lest -> mister kontroll. = skjev kontroll = skriver skjevt

- lese-anomali - ingen skrivekonflikt, read-only

- phantoms - lese noe, noen andre skriver noe etterpå som tilfredsstiller WHERE-betingelsen, noe som blir feil når man skal skrive

Phantom-katastrofe : leser predikatet 2 ganger -> oppdager at noe er forandret -> SI oppdater det fordi ting ikke kan endre seg-regel

Snapshot isolasjon (Snapshot Isolation – SI)

SI er en multiversjonsprotokoll for samtidighetskontroll, som garanterer at alle operasjoner i en transaksjon vil se et konsistent snapshot i databasen, uansett om det er lese- eller skriveoperasjoner.

En transaksjon som kjører under SI vil få sitt eget personlig snapshot av databasen som blir tatt ved transaksjonens starttidspunkt. Transaksjonen vil føle det som om ingen andre transaksjoner er tilstede i databasen, og alle andre transaksjoner kan ikke se hva som blir gjort av oppdateringer. SI tillater en mye høyere grad av samtidighet enn feks S2PL.

Men siden SI tillater endel samtidighetsfenomener, så er isolasjonsnivået serializable sterkere enn SI. [7]

Snapshot isolasjon kalles ”serializable” i Oracle, selv om SI ikke opprettholder konsistenskravet. [15]

Snapshot Isolation er ikke serialiserbar slik som ACID-kravet vel krever, men den beskytter allikevel mot at data kan rekke å bli overskrevet før transaksjonen er commited av en annen transaksjon e.l. Dette gjør den ved å ta et snapshot av databasen når den starter, og så terminere hvis verdiene den skal endre har blitt oppdatert siden snapshot'et ble tatt. Det betyr at den ikke har den samme robustheten som serializable, men at den vel stort sett går fortere. Går fortere, men feiler også oftere.

Hvis transaksjonen ikke har noen oppdateringer som er i konflikt med oppdateringer til andre eksterne transaksjoner som er gjort etter at snapshot ble tatt, vil en commit fullføres, og endringene som er gjort vil bli synlige for andre transaksjoner.

Hvis det derimot forekommer noen konflikter finnes det 2 konflikt-typer:

write-write-konflikt, som vil gjøre at transaksjonen aborterer,

read-write, der transaksjonen vil gå igjennom

Write-skew (skjev-skriving) er en anomali som er tillatt under SI, og ville ikke forekommet hvis systemet hadde vært serializable

Write-skew ser slik ut: r1(x) r2(y) w2(x) w1(y) .

SI er det strengeste nivået for isolasjon i Oracle og kalles der for ”serializable” . [9]

SQL-s isolasjonsnivå serializable: resultat av en kjøring skal være det samme som en eller annen seriell kjøring

Ved å bruke SI oppnår databasen bedre ytelse enn ved serialiserbarhet.

Mange av de vanligste samtidighetsfenomene unngås.

Den formelle definisjonen av Snapshot Isolation begrenser altså hva samtidige transaksjoner får “se” av hverandres operasjoner, og sier i tillegg at dersom to samtidige transaksjoner ikke har disjunkte skrivesett, må én av dem avbrytes.

Snapshot som blir tatt av databasen vil alltid være etter en commit, og kalles derfor et **commit-snapshot**. Dette er databasens siste konsistente tilstand. Et commit-snapshot

tidsstempler alle transaksjoner når de starter og når de committer. Alle andre transaksjoner vil ikke kunne se hva som er gjort før etter at commit er utført.

Det er en sjekk som gjøres ved oppdateringstidspunket, og som går ut på å kontrollere om samtidige transaksjoner oppdaterer de samme dataene ved commit.

Samtidighetsfenomener i Snapshot Isolation

Lister opp samtidighetsfenomenene som kan oppstå/ikke oppstå under Snapshot Isolation:

Samtidighetsfenomener som kan oppstå under Snapshot Isolation

- Phantoms

- Write Skew

- Read-Only Transaction Anomaly

Samtidighetsfenomener som *ikke* kan oppstå under Snapshot Isolation

Dirty Write

Dirty Read

Non-Repeatable Read

Phantoms

Lost Update

Read Skew

5. Verktøy

5.1 stORM

StORM [3] er et modelleringsverkøty som benyttes til å modellere ORM-diagrammer og generere SQL-tabeller. StORM presenterer data grafisk, og man designer systemer ved help av NIAM symboler som objekter, broer og skranker. Etter å ha gruppert diagrammet, så genereres SQL kode for Oracle database. Denne koden kjøres rett inn i databasen, og vil opprette tabeller, indekser, nøkler osv. som diagrammet viser. UiO har lisens på stORM, og siden det skulle være relativt enkelt å sette seg inn i, er det grunnen til at jeg valgte stORM.

5.2 Hammerora

Hammerora er et gratis open source verktøy som kan lastes ned fra *http://hammerora.sourceforge.net/*. Siden det er et stress-testings verktøy for Oracle 8i, 91, 10g og 11g, MySQL, og web applikasjoner på Linux/UNIX og Windows, så passet det godt for formålet. [1]

Hammerora brukes til å simulere transaksjonsload, og kan simulere en reell workload med så mange brukersesjoner som systemet kan håndtere. Dette gjør den perfekt til å måle gjennomstrømmingen av transaksjoner i databaser.

TPC (Transaction Processing Performance Council) [2]er en organisasjon av bedrifter med en spesiell interesse for databaseytelse, både hardware-, operativsystemprodusenter, og databaseleverandører. Deres formål er å lage objektive, verifiserbare ytelsesresultater av forskjellige databasekonfigurasjoner.

Til dette formålet har de laget et sett standardiserte tester, benevnt TPC-A til TPC-H på www.tpc.org .

TPC produserer benchmarks som måler transaksjonsprosesseringen (transactions processing TP) og databaseytelse for hvor mange transaksjoner et gitt system og database kan utføre pr.sekund/pr. minutt (tpmC). TpmC-tallet beregnes på bakgrunn av totalt medgått klokketid.

TPC typer:

Foreldet: TPC-A, TPC-App, TPC-B, TPC.D, TPC.R og TPC-W er utdaterte og brukes ikke.

TPC-C er en test for transaksjonsprosessering, og måler ytelsen til ”on-line transaction processing” (OLTP) databaser. OLTP er prosessering der det skjer mye skriving av små, isolerte deler av databasen. TPC-C har multiple transaksjonstyper, mer kompleks

database. Blir målt i transaksjon pr. minutt (tpmC). TPC-C er den mest populære benchmark testen, og resultater fra Oracle er de mest sette. Blir ofte brukt i computer miljøer for å simulere en gruppe brukere som kjører transaksjoner mot en database,

TPC-H er en test for ad-hoc spørringer til beslutningsstøttesystemer

TPC-E simlerer OLTP workload til meglerfirmaer [12]

TPC-C ble brukt i mine målinger.

Hammerora bruker Oratcl pakken(som er en utvidelse av Tcl-språket) med tcl-kommandoer for å integrere med Oracle, og kan på den måten oversette Oracle trace filer og kjøre de på nytt mot Oracle databasen. Tcl (Tool Command Language) [11] er et open source dynamisk programmeringsspråk som brukes for web og desktop applikasjoner, nettverk med mer.

Hammerora har også endel før-definerte simuleringer som er basert på TPC-C og TPC-H benchmarks spesifikasjoner. Jeg kjører Hammerora versjon: v.2.3

5.3 SQL*Plus

Sqlplus er et klientprogram som kan brukes til å kjøre SQL-kommandoer mot en Oracle-database fra UNIX. Sqlplus er det primære grensesnittet til en Oracle database server, som tilbyr et kraftfull men enkel måte å spørre, definere og kontrollere data på. Sqlplus leverer en full implementasjon av Oracle SQL og PL/SQL, sammen med en stor mengde utvidelser.

5.4 Statspack

Statspack [4] er et verktøy for å tune Oracle databaser som feks har fått problemer med ytelsen. Statspack pakken består av en mengde SQL, PL/SQL, og SQL*Plus scripts som samler inn og viser frem ytelsesdata. Dataene som samles inn lagres permanent som ytelsesstatistikk i Oracle tabeller, som senere kan brukes i rapporter. Rapportene vil vise standarden av helsen på databasen, som feks hvilke sql-er som bruker mest ressurser, oppsummering av load, wait event etc.

5.5 Oracle Database

En database består av en samling relaterte data der et databasesystem ofte kalt DBMS (Database Management System), brukes til å administrere og utføre operasjoner på dataobjektene. Eksempel på slike databasesystemer er Oracle, MySQL, og PostgresSQL.

Oracle Database er et relasjonsdatabasesystem (RDBMS) lagd av Oracle Corporation. Oracle er et lisensiert system.

5.5.1 Låsing i Oracle

Oracle bruker låser og multiversjons concurrency control system, oracle bruker row-level låsing, og oracle vil automatisk plassere låsen for meg og lagre lås-informasjonen I en data-block, der låsen holdes inntil transaksjonen enten er committet eller rullet tilbake. Multiversjons concurrency er en timestamp tilnærming for å lese originale data, der oracle vil skrive de originale data til et undo record I undo tablespace, queries har så et consistent view av data mens den tilbyr read konsistens - som vil se at den kun ser data fra en enkelt tidspunkt I tid.

Det eksisterer forskjellige typer låser I Oracle og tabeller, der du kan finne info om låsene:

- DML locks: row-level locks - beskytter daden mens den blir endret, der låsen ikke vil blokkere en som leser samme rad. En tabell-lock blir også plassert, men den sikrer at ingen Ddl blir brukt på tabellen

-DDL locks - ved endring på attributter så plasserer Oracle en exclusive lås på tabellen for å forhindre endringer på radene. Denne låsen er også brukt ved DML transaksjoner for å være sikre på at tabellen ikke blir endret ved endring/insert av data

- Latches - latches beskytter memory strukturen med SGA, som controller prosessene som aksesserer memory area's

-Internal locks - blir brukt av oracle for å beskytte structures som datafiler, tablespaces, og rollbacksegmenter

- Distributed locks - spesialiserte låsemekanismer brukt I distribuerte systemer

- Blocking locks - oppstår når en lock er plasser på et object av en bruker for å forhindre at andre aksesserer samme object

- deadlocks - oppstår når 2 sesjoner blokkerer hverandre mens de venter på en ressurs som den andre sesjonen holder på. Oracle stepper alltid inn for å løse problemet ved å kille en av sesjonene, og sjekke alert.log for deadlocks

5.6 PostgreSQL

PostgreSQL [10] er et gratis open-source objekt-relasjonsdatabase system, som har de samme funksjoner som de tradisjonelle kommersielle databasesystemer. Den kjører på alle typer operativsystem, inkl. Linux, UNIX, og Windows. PostgreSQL har full støtte for fremmednøkler, joins, views, triggere, og lagrede procedurer. Den inkluderer de fleste datatyper som integer, numic, varchar, boolean, date, interval, og timestamp. Den støtter også lagring for store binære objekter som bilder, lyd og video.

5.7 Plassering av verktøy

I min testsituasjon ligger både Oracle Database og testverkøtyet Hammerora på samme server. Det blir egentlig anbefalt av Hammerora at de helst bør ligge på hver sin server, for å sikre at de ikke påvirker hverandre når det gjelder ressurser på server. Siden jeg kun hadde én server tilgjengelig for min masteroppave, ble de begge to lagt på samme server.

6. Prosjektbeskrivelse

Denne oppgaven er en studie i hvordan isolasjonsnivåer påvirker gjennomstrømmingen av transaksjoner i en Oracle database. Ved å stress-teste en Oracle database, simulerte jeg en forventet topp av belastning, og observerte hvordan databasen taklet det ytelsesmessig, ved hjelp av statistikkverktøyet statspack. Testsituasjonen gikk ut på å simulere et variabelt antall brukere som samtidig kjørte samme transaksjon mot samme database, med samme isolasjonsnivå. Deretter sammenlignet jeg mellom isolajonsnivåene gitt henholdsvis 2,3,4,7 og 10, brukere. Jeg kjørte også igjennom den samme transaksjonen med 25 brukere gitt nivå Read Only, for å se hvordan forskjellen i ytelse var.

Disse testene ble kjørt på følgende hardware:

Oracle versjon: Oracle Database 10g Enterprise Edition Release 10.2.0.4.0

Databaseinstans: IFIFORSK.ifi.uio.no

Server OS: RedHat Enterprise Linux 5.4, 32 bit

Servernavn: blot.ifi.uio.no

Disk/lagring: intern disk (ikke RAID)

Filsystem: ext3

Memory(RAM): 2GB

CPU type: Intel(R)Xeon(TM)CPU2.40GHz

Ant. CPU: 2

Ant. kjerner: 2

PGA (System Global Area): 800 MB

SGA(Program Global Area): 192 MB

PGA er en del av memory (RAM) som brukes til å lagre data og kontrollinfo. for Oracle's server-processer. Eksempel på det er sort area, hash area, session cursor cache etc,
SGA er en del av memory (RAM) som allokeres når en Oracle instans startes opp. Størrelsen og funksjon til SGA blir kontrollert av initialiseringsparametre i init.ora eller spile. Brukes til å lagre data og kontrollinfo. [14]

Det interessante var å se hva som skjer vedrørende:

- ytelsen for hvert isolasjonsnivå
- hvilke flaskehalser som dukket opp
- hvilke typer låser som oppsto
- hvilket isolasjonsnivå som slapp igjennom flest brukere

- det å teste hvor den fysiske grensen til server/maskinen lå - hvis antall samtidige brukere økte

- hvilket isolasjonsnivå er det beste totalt

- klarer hardware å håndtere den forventede load

6.1 Testkjøring

Tidkrevende tester:

Å kjøre en del av testene var svært tidkrevende:

For å sikre likt utgangspunkt for alle tester, måtte alle parametre nullsettes. Det som ble gjort var:

Sletting av data som lå i tabellene

Dropping av bruker

Oppretting av bruker på nytt med riktig tablespace

Gi riktige rettigheter til bruker

Importere alle data inn i tabeller på nytt for hver bruker, siden forrige testkjøring endret dataene, og testene måtte ha samme startsituasjon

Restarte dababasen for å flushe cache. For å unngå at nye testrunder hentet resultater fra cache, ble databasen restartet mellom hver ny kjøring. Dette er den beste måten å fjerne innholdet i cache'n på. Det finnes også to manuelle kommandoer, men en restart er det sikreste

Starte Hammerora

Endre parametre i Hammerora

Ta et snapshot av databasen for statspack

Kjøre ny test

Når testkjøring er ferdig tas et nytt snapshot

Kjør en statspack rapport som samler inn informasjon mellom de snapshots-tidspunktene nettopp tatt

Totalt for en test tok dette ca. 30 min + tiden det tok for testkjøring - som varte alt fra 20 min til flere timer.

Å kjøre en import krevde ekstremt mye ressurser på serveren, og det kunne derfor kun gå én import om gangen.

Hver testkjøring tok også mye ressurser, og kunne også kun gå én om gangen, da flere på likt kunne hatt innvirkning på testvilkårene.

Fremgangsmåte

Alle kjøringer skjedde mot samme database. Det ble opprettet en ny bruker for hver kjøring. Mellom hver kjøring ble databasen restartet.

For hvert isolasjonsnivå (read committed og seralizable) ble det kjørt

- samme antall brukere

- samme update

- endret de samme dataene

Antall testbrukere var 2,3, 4,7, 10, 25, men som nevnt over kjørte jeg med 25 brukere kun med nivå read committed.

Observert i Hammerora mens kjøringene gikk:

- Belastning på systemet: mye/lite

- Observerte låser som oppsto

-Målte tid på kjøringene

Jeg tok utgangspunkt i en Postgres-database fra et informatikk-kurs (inf3100), som data for testene. Deretter brukte jeg verktøyet stORM til å modellere en tilnærmet lik film-database for Oracle. Jeg tegnet opp alle tabeller, med hvert sitt entydige tabellnavn, alle relasjoner mellom tabellene, og attributter på de som skulle ha det. stORM genererte tabeller, indekser, nøkler, og triggere (filmdb.txt) for Oracle etter at databasesystem ble valgt.

Så lagde jeg et java-script som eksporterte alle data fra postgresdatabasen og importerte de inn tabellene mine. For å ha flere testmuligheter, opprettet jeg flere brukere, og tok en eksport av tabellene med bruker vibekes over i de andre brukerne

Jeg brukte litt tid på å finne den rette spørringen med update's som ville bruke passende tid på å kjøre. Jeg endte på spørringen under:

update filmparticipation set parttype = 'producer' where personid IN (select p.personid from person p, filmparticipation fp where p.personid = fp.personid and fp.parttype like '%' and p.personid IN (select bi.personid from biographyitem bi, person p where bi.personid = p.personid))

Før jeg satte i gang med en test, satte på tid (set time on) for å måle hvor lang tid update-setningen brukte på å kjøre igjennom. Deretter slo jeg på tracing (ALTER SESSION SET sql_trace = true) for å opprette en trace-fil som kunne brukes videre i Hammerora.

Trace-filene legger seg under USER_DUMP_DEST, som hos meg blir /ifi/blot/ora_drift/IFIFORSK/udump

6.2 Hammerora-kjøring

For å starte verktøyet Hammerora:

```
vibekes@blot ~> cd /local/opt/hammerora
vibekes@blot /local/opt/hammerora> sh ./hammerora.tcl &
```

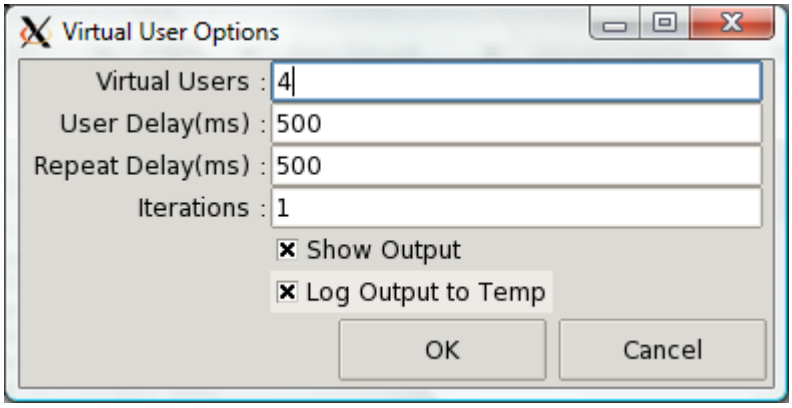
For å se at Hammerora-prosessen (wish8.5) på Linux kjører brukes:
'ps -ef |grep wish' som gir

```
vibekes  13535  6996  2 22:19 pts/0    00:00:00 wish8.5 -file ./hammerora.tcl:
```

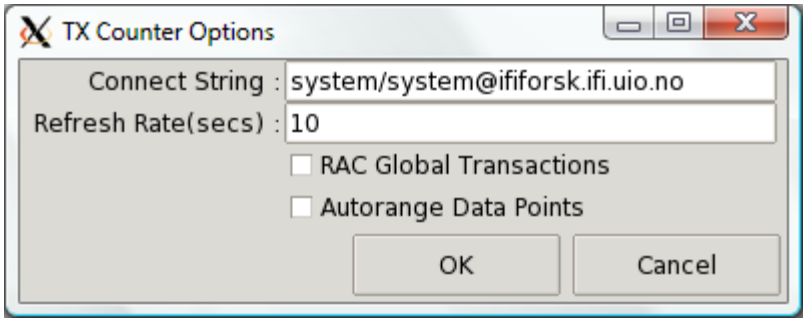
I Hammerora åpnes jeg den raw SQL trace filen som jeg genererte ved å kjøre spørringen min i databasen. Den ble konvertert til Tcl script (som er det programmeringsspråket Hammerora forstår), og kan så kjøres om igjen for hver test mot databasen.

Under vises de parametrene som måtte endres før en loadtest kunne settes i gang:

I Virtual user Options så velger man hvor mange samtidige virtuelle brukere (vusers) som skal kjøre, og man kan krysse 'Show output' for å få utskrift på skjermen. Her kan det også velges om output skal logges til Temp:



Så må connect-string som system-passord og databaseinstansnavn endres for å kunne kjøre TX counter (evt. endre system-pw til et passende for anledningen). TX Counter viser en graf over antall transaksjoner pr minutt. System-brukeren logger inn i databasen for å få vist dette. Se bilde.



Så må type benchmark TPC-C velges, og man må samtidig endre system-passord og instansnavn

Så krysses det av for hvilket TPC-C driver script som skal brukes: AWR snapshot Driver script

Det må også endres til riktig bruker inni oratcl scriptet for at man skal kunne koble seg til databasen. Det må også settes navn på databaseinstans, og ikke minst et isolasjonsnivå

Jeg la også inn noen ekstra linjer som skriver ut kommentarer for å lettere se hvor langt Hammerora er kommet i mitt oraTcl-script. Se under:

```
#!/usr/local/bin/tclsh8.5
package require Oratcl
####UPDATE THE CONNECT STRING BELOW###
set connect vibeke3/<pw>@IFIFORSK.ifi.uio.no
set lda [oralogon $connect]
puts "connecter"
set curn2 [oraopen $lda ]
set sql2 "set transaction isolation level read committed "
orasql $curn2 $sql2
puts "isolation: read committed"
set curn5 [oraopen $lda ]
set sql5 "alter session set sql_trace= true"
orasql $curn5 $sql5
puts "setter på trace=true"
set curn1 [oraopen $lda ]
set seconds [clock seconds]
puts "før update filmparticipation: The time is [clock format $seconds]"
set sql1 "update filmparticipation set parttype = 'producer' where personid IN (
select p.personid from person p, filmparticipation fp where p.personid = fp.pers
onid and fp.parttype like '%' and p.personid IN (select bi.personid from biograp
hyitem bi, person p where bi.personid = p.personid )) "
orasql $curn1 $sql1
set seconds [clock seconds]
puts "Etter update filmparticipation: The time is [clock format $seconds]"
set curn3 [oraopen $lda ]
set sql3 "commit "
orasql $curn3 $sql3
puts "kjørt commit"
oraclose $curn1
set curn1 [oraopen $lda ]
set sql1 "alter session set sql_trace= false "
orasql $curn1 $sql1
puts "etter: trace er satt til false"
```

oraclose $curn5
oraclose $curn1
oralogoff $lda

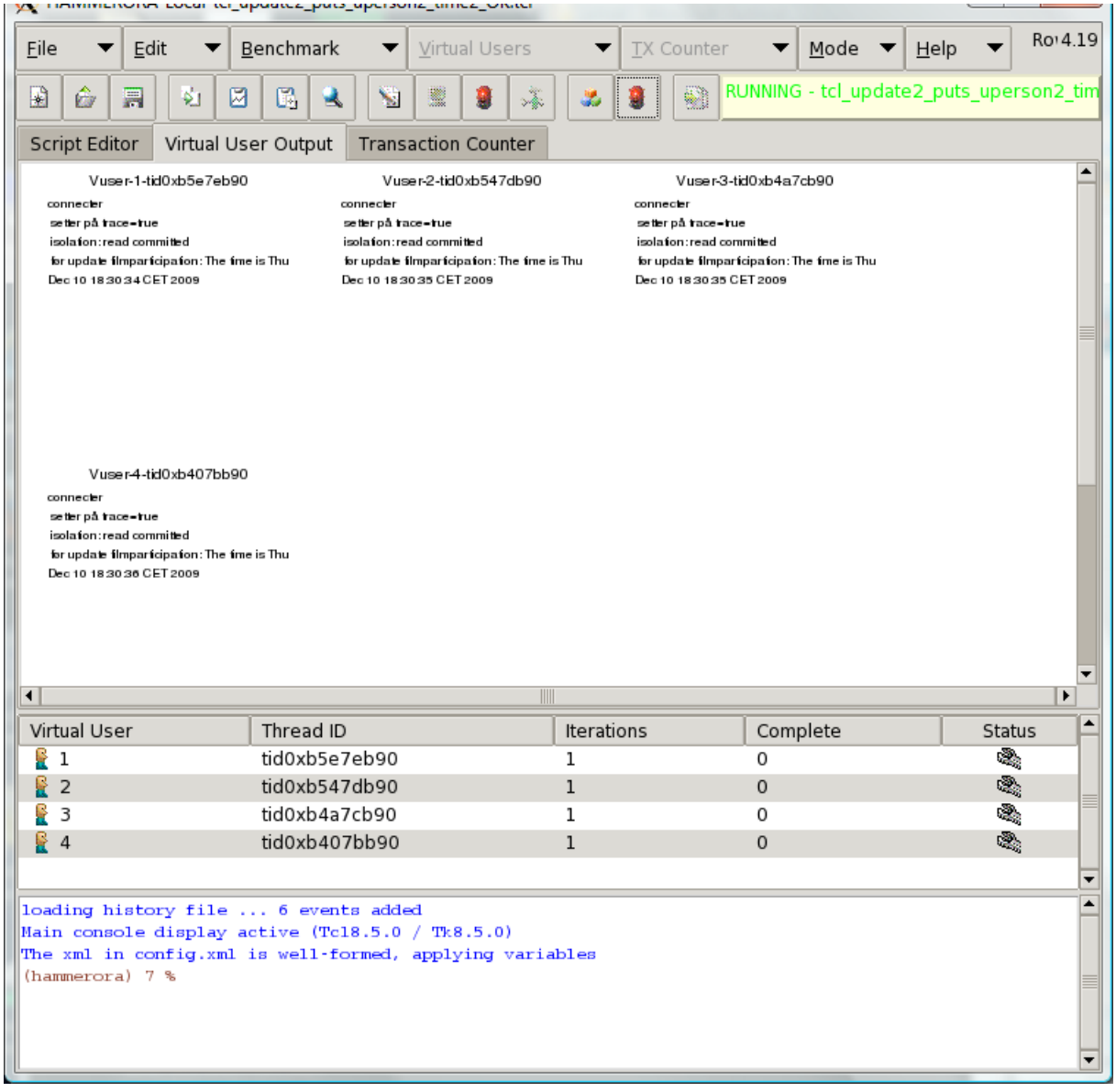
Deretter:

--velg load for å hente inn de virtuelle brukerne

--ta manuellt et snaphot av databasen som skal brukes som statistikk for statspack

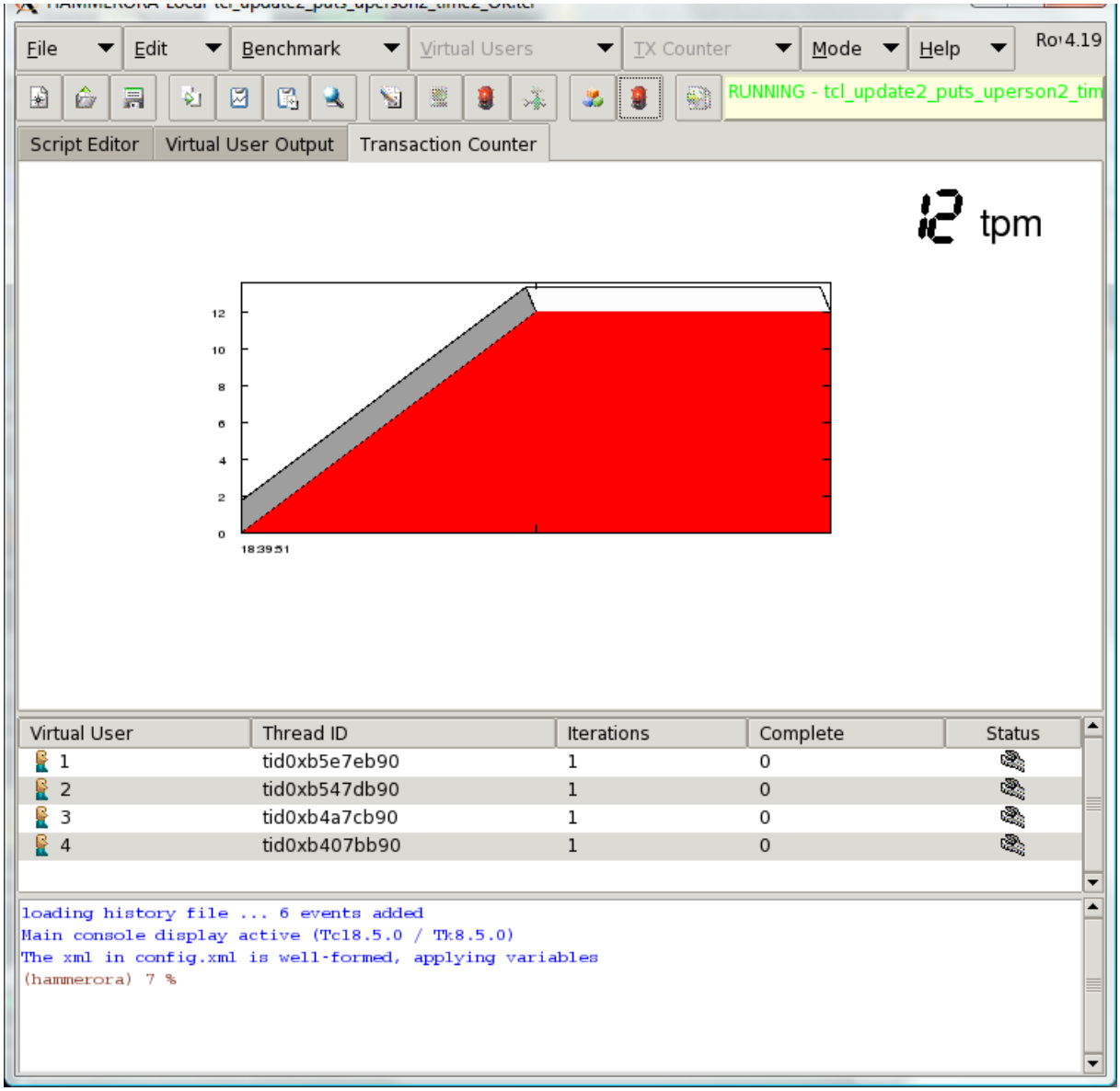
--start Hammerora loadtest

--start TX counter slik at den begynner å hente inn ant. transaksjoner pr. minutt



Bilde viser en Hammerora-kjøring med isolasjonsnivå=read committed og med 4 vusers.

De fire vusers har startet å kjøre update omtrent samtidig.



Bildet viser grafen for Transaction Counter. Klokken 18:39:51 ser vi at ant transaksjoner pr.minutt er her 12.

Med Linux kommandoen ’top’ kan man få frem en liste over prosessaktiviteten som går på server akkurat nå. Det er spesiellt de prosessene som bruker mest CPU som vises i listen, og det er en fin måte å følge med på hvor mye CPU oracle-prosessene tar mens Hammerora kjører:

top - 18:48:20 up 18 days, 4:08, 7 users, load average: 4.86, 5.14, 4.09
Tasks: 166 total, 3 running, 162 sleeping, 0 stopped, 1 zombie
Cpu(s): 29.5%us, 3.7%sy, 0.0%ni, 22.1%id, 43.6%wa, 0.0%hi, 1.2%si, 0.0%st
Mem: 2075388k total, 2021276k used, 54112k free, 5772k buffers

Swap: 6144852k total,   22312k used, 6122540k free, 1602004k cached

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
25712 oracle    18   0  934m  51m  27m R 28.3  2.5   3:32.28 oracle
25716 oracle    17   0  934m  48m  24m D 28.3  2.4   3:04.62 oracle
25651 oracle    15   0  920m 529m 513m S  6.0 26.1   1:49.85 oracle
24984 oracle    16   0  914m 499m 483m D  2.7 24.6   0:09.80 oracle
  194 root      10  -5     0    0    0 S  1.0  0.0  17:48.16 kswapd0
24986 oracle    15   0  913m  34m  32m S  1.0  1.7   0:05.44 oracle
25300 vibekes   15   0  2504 1216  920 R  0.7  0.1   0:04.36 top
 4135 root      34  19     0    0    0 R  0.3  0.0 118:21.04 kipmi0
25298 vibekes   15   0  109m  28m 9892 S  0.3  1.4   0:04.55 wish8.5
25705 oracle    16   0  899m  31m  28m S  0.3  1.5   0:00.26 oracle
```

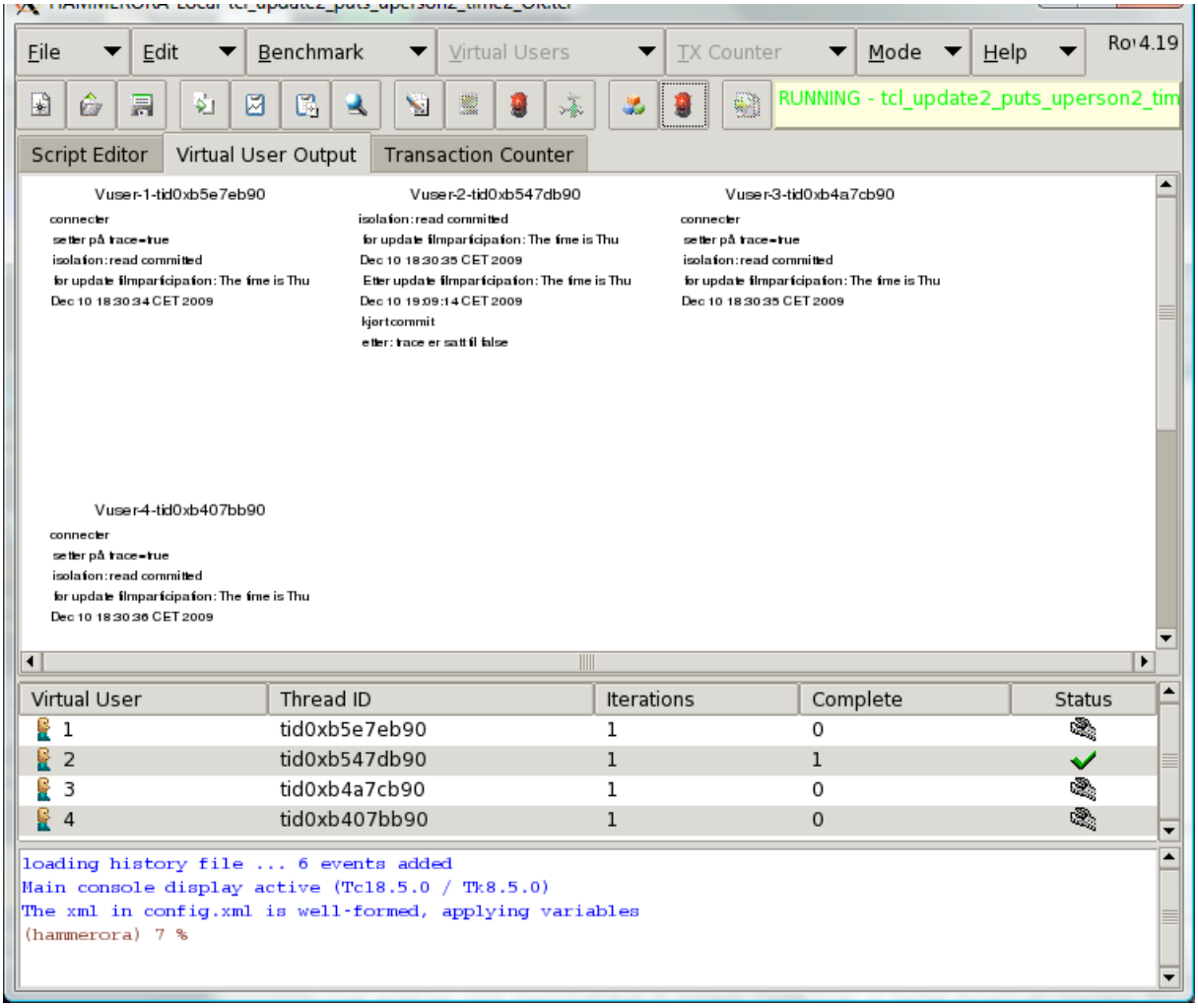
Man kan se at de fire øverste prosessene er mine oracle-prosesser for vusers i Hammerora.

Under er der noen select-setninger som viser hva som skjer i databasen mens kjøringen pågår. Vi ser at prosess 142 blokkerer for resten av prosessene, slik at de ikke kan fortsette før 142 har låst opp sin lås:

```
SQL> SELECT substr(DECODE(request,0,'Holder: ','Waiter: ')||sid,1,12) sess,
       id1, id2, lmode, request, type, inst_id
 FROM GV$LOCK
WHERE (id1, id2, type) IN
  (SELECT id1, id2, type FROM GV$LOCK WHERE request>0)
    ORDER BY id1, request;

SESS                ID1       ID2      LMODE    REQUEST TY    INST_ID
------------------- ---------- ---------- ---------- ---------- -- ----------
Holder: 142         262147     91094          6          0 TX         1
Waiter: 143         262147     91094          0          6 TX         1
Waiter: 140         262147     91094          0          6 TX         1
Waiter: 141         262147     91094          0          6 TX         1
```

Vi ser også at det er satt rad lås (row lock) på sesjonene. En radlås kalles også for en TX-lås, og en er lås som settes på en enkelt rad i en tabell. En transaksjon som gjør endringer med insert, update, merge, eller select … for update, krever at det skal settes en rad lås på hver rad i tabellen. Denne rad låsen vil eksistere inntil transaksjonen blir committed eller tar en rollback. Rad låser brukes i databaser for å forhindre at to transaksjoner endrer på samme rad. Og i dette tilfellet vil sesjon 142 holde på sin lås inntil den er ferdig med update.



Bildet over viser at en vuser har fullført sin transaksjon, men de andre transaksjonene fortsatt prosesserer. Vuser 2 brukte ca 39min på å kjøre sin update.

På flere av testkjøringene oppsto det feilmelding med deadlock. Jeg viser kjøring som ble kjørt med 7 vusers og isolasjonsnivå read committed.

Select viser hvilke sesjoner som blokkerer andre sesjoner:

```
SQL> SELECT substr(DECODE(request,0,'Holder: ','Waiter: ')||sid,1,12) sess,
       id1, id2, lmode, request, type, inst_id
 FROM GV$LOCK
WHERE (id1, id2, type) IN
  (SELECT id1, id2, type FROM GV$LOCK WHERE request>0)
    ORDER BY id1, request;
```

SESS	ID1	ID2	LMODE	REQUEST	TY	INST_ID
Holder: 140	393241	121251	6	0	TX	1
Waiter: 143	393241	121251	0	6	TX	1
Waiter: 147	393241	121251	0	6	TX	1
Holder: 132	524312	121072	6	0	TX	1

```
Waiter: 140          524312    121072        0       6 TX       1
Waiter: 134          524312    121072        0       6 TX       1
Waiter: 130          524312    121072        0       6 TX       1
Waiter: 137          524312    121072        0       6 TX       1
8 rows selected.
```

Vi ser at
140 blokkerer for 143, 147,
132 blokkerer for 140, 134, 130 og 137
132 og 140 blokkerer for hverandre: dvs. har fått en deadlock situasjon

Dette er en annen select som viser sessionid, om den blokkerer andre, type lås, og mode:

```
SQL> select SESSION_ID, BLOCKING_OTHERS,LOCK_TYPE,MODE_HELD from
DBA_LOCKS;

SESSION_ID BLOCKING_OTHERS LOCK_TYPE           MODE_HELD
---------- -------------- -------------------- ----------
       140 Not Blocking   Transaction          None
       143 Not Blocking   Transaction          None
       147 Not Blocking   Transaction          None
       130 Not Blocking   Transaction          None
       137 Not Blocking   Transaction          None
       134 Not Blocking   Transaction          None
       143 Not Blocking   DML                  Row-X (SX)
       147 Not Blocking   DML                  Row-X (SX)
       140 Not Blocking   DML                  Row-X (SX)
       137 Not Blocking   DML                  Row-X (SX)
       134 Not Blocking   DML                  Row-X (SX)
       130 Not Blocking   DML                  Row-X (SX)
       132 Blocking       Transaction          Exclusive
       140 Blocking       Transaction          Exclusive

30 rows selected.
```

Vi ser at det kun er to sesjoner som sperrer, og det er 132 og 140.

Alert loggen viser mer detaljer om hva som er feil, som at det er en deadlock forårsaket av brukerfeil, hvilke sesjoner som har problemer, hvilke rader de prøver å oppdatere, hvilken update-setning som kjøres, og generelt litt system-detaljer:

```
Mon Dec  7 18:50:16 2009
ORA-00060: Deadlock detected. More info in file
/local/oracle/admin/IFIFORSK/udump/ififorsk_ora_9908.trc.
--/local/oracle/admin/IFIFORSK/udump/ififorsk_ora_9908.trc

Under ser du trace-filen /local/oracle/admin/IFIFORSK/udump/ififorsk_ora_9908.trc
-------------------------------------------------------------------------------------------------------------
*** 2009-12-07 18:50:16.268
DEADLOCK DETECTED ( ORA-00060 )
```

[Transaction Deadlock]
The following deadlock is not an ORACLE error. It is a deadlock due to user error in the design of an application or from issuing incorrect ad-hoc SQL. The following information may aid in determining the deadlock:
Deadlock graph:

	---------Blocker(s)--------				---------Waiter(s)---------			
Resource Name	process	session	holds	waits	process	session	holds	waits
TX-00080018-0001d8f0	30	132	X		27	140	X	
TX-00060019-0001d9a3	27	140	X		30	132	X	

session 132: DID 0001-001E-00000001 session 140: DID 0001-001B-00000001
session 140: DID 0001-001B-00000001 session 132: DID 0001-001E-00000001
Rows waited on:
Session 140: obj - rowid = 00016624 - AAAWYkAAGAAAogHAD+
(dictionary objn - 91684, file - 6, block - 165895, slot - 254)
Session 132: obj - rowid = 00016624 - AAAWYkAAGAAA0IJACw
(dictionary objn - 91684, file - 6, block - 213513, slot - 176)
Information on the OTHER waiting sessions:
Session 140:
pid=27 serial=3 audsid=2351391 user: 109/VIBEKE3
O/S info: user: vibekes, term: pts/5, ospid: 9488, machine: blot.ifi.uio.no
program: wish8.5@blot.ifi.uio.no (TNS V1-V3)
application name: wish8.5@blot.ifi.uio.no (TNS V1-V3), hash value=1624437838
Current SQL Statement:

```
update filmparticipation set parttype = 'producer' where personid IN (select p.personid
from person p, filmparticipation fp where p.personid = fp.personid and fp.parttype like
'%' and p.personid IN (select bi.personid from biographyitem bi, person p where
bi.personid = p.personid ))
```

End of information on OTHER waiting sessions.
Current SQL statement for this session:

```
update filmparticipation set parttype = 'producer' where personid IN (select p.pers
onid from person p, filmparticipation fp where p.personid = fp.personid and fp.part
type like '%' and p.personid IN (select bi.personid from biographyitem bi, person p
 where bi.personid = p.personid ))
```

Løsning for deadlock:
I dette tilellet skulle Oracle databasen selv ha løst opp i deadlock'en, ved å rulle tilbake den ene transaksjonen, og gi melding om at feil en var oppstått. Dette skjedde imidlertig ikke i Hammerora. Det kom melding om deadlock, og så sto Hammerora og gikk i 8 timer. Da kom en ny deadlock, og siden en ny noen timer etter det igjen. Jeg fikk dessverre aldri testet om denne testen faktisk ville gått igjennom, siden putty-sesjonen min ikke ble tillatt å opprettholde kontakt så lenge. Jeg antar at 'gitt nok tid', så ville alle bortsett fra en vuser feilet mens den siste ville ha gått igjennom og fullført.
Dette testet jeg manuelt, ved å drepe den bruker -sesjonen som var inaktiv etterhvert som deadlock's oppsto. Tilslutt når det var igjen en bruker-sesjon igjen, gikk den igjennom og fullførte.

Jeg fikk to ORA-feilmeldinger underveis i testingen.

En fordi isolasjonsnivået serializable sammen med Hammerora ikke klarte å kjøre igjennom med mer enn én vuser. Uansett stoppet alle vusers med feilmeldingen:

ORA-08177: can't serialize acces for this transaction

Denne feilmeldingen kommer når en sesjon prøver å oppdatere en rad som har blitt endret etter at din transaksjon startet å lese den. [5] Graden av isolasjon kommer med en pris,og prisen er den følgende mulige feilen, høyere bruk av ressurser, og lavere ytelse i databasen.

Oracle Database tar en **optimistisk** tilnærming til serialization ved at den gambler med det faktum at de data din transaksjon vil oppdatere ikke vil bli oppdatert av andre transaksjoner. Som regel lønner denne gamblingen seg, og spesielt ved raske transaksjoner som i OLTP-type systemer. Hvis ingen andre transaksjoner ikke oppdaterer de samme dataene i løpet av transaksjonen, så gir dette isolasjonsnivået samme grad av ytelse som andre isolasjonsnivåer. Ulempen er feilen ORA-01877 som oppstår hvis det skulle være mange transaksjoner som skal oppdatere nettopp de samme verdiene på likt. Oracle har valgt å ta sjansen på at denne feilmeldingen kan oppstå, og bruker snapshot isolation som serializable isolation. De har satset på ytelse framfor mindre feilmeldinger.

Den andre feilmeldingen "ORA-00060: deadlock detected while waiting for resurces" dukket opp for isolasjonsnivå read committed.

Denne feilmeldingen kom når antall brukere ble økt til 7 vusers. Det var tydelig at det var to transaksjoner som sto og ventet på hverandre, slik at ingen av dem fikk gjort noe annet enn nettopp det.

6.3 Statspack-kjøring

Statspack er et ytelse- og rapporteringsverktøy som er levert av Oracle for Oracle8i og oppover. [6] Under installering blir brukeren PERFSTAT opprettet automatisk, og blir satt som eier av alle objekter til denne pakken. Statspack samler og analyserer databasestatistikk i en rapport, som viser bl.a. belastning, bufferstatistikk, wait events, Top 5 wait events og diskaktivitet.

For å kjøre en statspack rapport, tas det snapshot av databasen på forskjellige tidspunkter. Gjerne rett før og etter en transaksjon. Utfra en oversikt over snapshotsid velges tidsrom for når statspack skal samle data og lage statistikk. Manuelle snapshot av databasen på forskjellige tidspunkter, gir mulighet for rapporter for sammenligning.

Fordelen er å kunne sammenligne en rapport tatt fra når du har et ytelsesproblem med en rapport som er tatt fra systemet når ytelsen for systemet er normalt. På den måten er det

enklere å se avvikene på verdiene.

Ved innstallering av statspack ble det automatisk satt et nivå for hvor mye statistikkinformasjon som skulle innhentes. Dette nivået kan endres til et høyere nivå, hvis det trengs mer inngående detaljer om hva som skjer. Jeg kjørte med standard nivå som er lik 5.

Under og i resten av dette kapitlet vises for det meste et eksempel (eks) på hvordan kjøre en statspack-rapport. Under vises et eksempel på en rapport der jeg valgte to snapshot-tidspunkter som var tatt rett før og rett etter at Hammerora-kjøring med isolasjonsnivå 'read committed' og med 4 virtuelle brukere,:

Eks:

```
IFIFORSK@blot:~[3]sqlplus 'vibekes / as sysdba'
SQL*Plus: Release 10.2.0.4.0 - Production on Sat Dec 12 03:29:26 2009
Copyright (c) 1982, 2007, Oracle.  All Rights Reserved.
Enter password:
Connected to:
Oracle Database 10g Enterprise Edition Release 10.2.0.4.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> @?/rdbms/admin/spreport.sql

Current Instance
~~~~~~~~~~~~~~~~
   DB Id    DB Name      Inst Num Instance
----------- ------------ -------- ------------
  627831043 IFIFORSK            1 IFIFORSK


Instances in this Statspack schema
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   DB Id    Inst Num DB Name      Instance     Host
----------- -------- ------------ ------------ ------------
  627831043        1 IFIFORSK     IFIFORSK     blot.ifi.uio
                                               .no

Using  627831043 for database Id
Using          1 for instance number


Specify the number of days of snapshots to choose from
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Entering the number of days (n) will result in the most recent
(n) days of snapshots being listed.  Pressing <return> without
specifying a number lists all completed snapshots.


Listing all Completed Snapshots

                                            Snap
Instance     DB Name        Snap Id   Snap Started    Level Comment
```

```
------------ ------------ --------- ------------------ ----- --------------------
IFIFORSK     IFIFORSK           243 10 Dec 2009 18:24     6
                                254 10 Dec 2009 19:52     6
Specify the Begin and End Snapshot Ids
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Enter value for begin_snap: 243
Enter value for end_snap: 254

Specify the Report Name
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Enter value for report_name:

Enter value for report_name: read_committed_4users_nyest.log
```

Navnet på rapporten er 'read_committed_4users_nyest.log', og den blir lagret i den katalogen som du sto i da du kjørte statspack.

Eks: under viser rapporten som ble opprettet av Statspack:

STATSPACK report for

```
Database    DB Id    Instance     Inst Num Startup Time    Release     RAC
~~~~~~~~~ ----------- ------------ -------- --------------- ----------- ---
        627831043 IFIFORSK           1 10-Dec-09 18:21 10.2.0.4.0  NO

Host  Name:   blot.ifi.uio.no  Num CPUs:    2       Phys Memory (MB):    2,027
~~~~

Snapshot      Snap Id     Snap Time      Sessions Curs/Sess Comment
~~~~~~~~~    ---------- ------------------ -------- --------- -------------------
Begin Snap:       243 10-Dec-09 18:24:39       20       4.2
  End Snap:       254 10-Dec-09 19:52:22       21       5.4
   Elapsed:            87.72 (mins)
Cache Sizes                   Begin       End
~~~~~~~~~~~~~                 ---------- ----------
          Buffer Cache:       472M       484M  Std Block Size:         8K
        Shared Pool Size:      92M        80M      Log Buffer:     6,841K
```

Dette er starten på rapporten, og viser en oppsummering av miljø og de to valgte snapshots for rapporten. I situasjoner der man har rapporter tatt fra når databasen er normal, kan man med letthet se om det har skjedd noen endringer på versjoner som Oracle software og størrelse på SGA [13]

De resultatene jeg har konsentrert meg om fra Statspack er:

- Snapshot: Elapsed
- Load Profile: Logical reads, Physical reads, Physical writes, Hard parses
- Instance Efficiency Percentages: Buffer hit ratio
- Top 5 Timed Events: enq: TX - row lock contention, db file sequential read, log file parallel write, CPU Time
- Host CPU: Load Average

- Statistic: parse time elapsed, DB CPU, DB Time

Snapshot
‘ Elapsed
Total tid på alle sql-kall i snapshot

Users/ isolasjonN	2	3	4	7	10	25
Serializable	25.03	19.08	57.90	50.90	41.95	34.38
Read Committed	41.88/76 .48	43.27	87.72 (27.78 heng)	76.48	26.65	

Forskjeller: Sammenligner jeg de to isolasjonsnivåene, så er det ’Read committed’ som generelt bruker mest tid på sql-kall, men vi må huske på at isolasjonsnivå serializable kun kjører én vuser igjennom. Og f.o.m 7 vusers, så oppstår det deadlock i transaksjonskjøringen for ’Read committed’

Load Profile:
Load profile viser om loaden endres over tid.

Eks.

Load Profile ~~~~~~~~~~~~	Per Second ----------	Per Transaction ----------
Redo size:	1,349,006.75	308,687,936.35
Logical reads:	5,433.79	1,243,392.39
Block changes:	10,237.48	2,342,601.61
Physical reads:	88.91	20,344.83
Physical writes:	349.16	79,897.91
User calls:	4.67	1,067.83
Parses:	3.51	802.87
Hard parses:	0.49	111.09
Sorts:	2.78	636.00
Logons:	0.01	3.26
Executes:	8.80	2,014.52
Transactions:	0.00	

% Blocks changed per Read: 188.40 Recursive Call %: 95.36
Rollback per transaction %: 4.35 Rows per Sort: 15.17

I denne delen kan man se om load endrer seg over tid. Hvis applikasjonen ikke er stabil over tid, vil verdien for ’pr.transaksjon’ endre seg ganske dramatisk gjennom en tidsperiode. Dette gjelder også verdien for ’pr.sekund’.
Det fins ingen korrekte verdier på load, men det er enkelte ting som kan ses nærmere på hvis de har en høyere verdi enn 100 pr.sek. for ’hard parses’, og andre parse-verdier på mer enn 300 pr.sek.

- **Logical reads:**

Leser i minnet (memory/cache) etter de tabelldata den leter etter.

Serializable

Users	2	3	4	7	10	25
Pr. Sek	4,984.99	6,478.28	5,331.50/ 3,775.76	3,775.76	4,957.47	4,071.76
Pr. trans	1,247,909.17	1,236,271.67	559,097.00/ 887,012.08	887,012.08	1,247,795.50	4,200,018.00

Read Committed

Users	2	3	4	7	10	25
Pr. sek	5,750.66	8,236.80	5,433.79	10,189.73	8,898.53	
Pr. trans	1,204,284	1,336,420.00	1,243,392.39	5,845,086.38	1,422,875.10	

Forskjell: Read committed har en høyere verdi enn read committed.

- **Physical reads**

Leser på disk når den ikke finner de tabelldata den leter etter i cache(minne)

Serializable

Users	2	3	4	7	10	25
Pr. sek	124.20	188.30	158.09	122.61	168.04	317.17
Pr. trans	41,091.50	35,933.50	16,578.33	28,802.77	42,294.90	327,160.00

Read Committed

Users	2	3	4	7	10	25
Pr. sek	140.35	138.89	88.91	214.97	234.80	
Pr. trans	29,391.17	22,535.31	20,344.83	123,315.00	37,544.80	

Forskjell: Det ser ut til at de begge leser like mye på disk. Verdien er høyere annen hver gang på nivåene.

- **Physical writes**

Som physical reads, men at den skriver på disk

Serializable

Users	2	3	4	7	10	25
Pr. sek	365.25	562.04	493.41	358.70	551,91	642.48
Pr. trans	91,434.17	107,256.83	51,742.60	84,266.54	138,916.90	662,722.50

Read Committed

Users	2	3	4	7	10	25
Pr. sek	371.71	527.38	349.16	363.12	741,07	
Pr. trans	77,841.75	85,568.13	79,897.91	208,292.88	118,496.50	

Forskjell: Her varierer det på hvem som skriver mest. Kan ikke se at noen av dem peker seg veldig ut. Read committed har litt høyere verdier på ant. brukere på 7 og oppover. Den verdien som virkelig skiller seg ut er read committed for 7 brukere med en verdi på 208,292.88 writes pr. transaksjon. Det er vanskelig å se grunnen til at denne verdien skulle bli så høy.

- **Hard parses**

Vil ha få av dem pga. det er tidkrevende.

Users/ isolasjonN	2 pr.sek/pr.transaksjon	3	4	7	10	25
Serializable	0.67 / 166.83	0.16 / 31.17	0.62 / 64.93	0.69/ 162.08	0.59/ 148.80	0.69 / 716.00
Read Committed	0.11 / 22.92	0.16/ 25.88	0.49 / 111.09	0.71/ 407.38	1.08/ 172.40	

Forskjell: Read committed har en lavere verdi av hard parses opp til 3 brukere, men så tar serializable over og har den laveste verdien.

Instance Efficiency Percentages

Denne delen inneholder de mest kjente forholdene som er relatert til operasjoner til databaseinstans slik som ’Buffer Hit ratio’ (også kalt ’Cache Hit Ratio’) og ’Library Hit ratio’ (også kjent om ’Library Cache Hit ratio’). Som i Load Profile, så fins det ingen korrekte verdier, men kun det som er passende for din applikasjon og workload.

Sammenliges resultater over tid, så ser man trender, som man så kan sjekke ut, før de blir et altfor stort problem

- **Buffer hit ratio:**

IFIFORSK er en ganske liten database, slik at masse blokker blir bufret i minnet, som igjen gjør at det blir ofte høy CPU bruk. En ’buffer hit’ andel bør være så nær 100% som mulig. Som en tommelfingerregel bør Instance Efficiency Percentages ideelt være på over 90%.

Eks:
Instance Efficiency Percentages
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Buffer Nowait %:	100.00	Redo NoWait %:	99.99
Buffer Hit %:	99.71	In-memory Sort %:	100.00
Library Hit %:	88.52	Soft Parse %:	86.16
Execute to Parse %:	60.15	Latch Hit %:	100.00
Parse CPU to Parse Elapsd %:	19.10	% Non-Parse CPU:	98.16
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Users/ isolasjonN	2	3	4	7	10	25
Serializable	99.02 %	98.90	98.97	98.98	99.27	96.57
Read Committed	98.79	99.61	99.71	98.51	99.31	

Forskjell: Begge isolasjonsnivåene treffer med over 98% av alt den leter etter i bufferet. Hvilket vil si at minnet blir lest én gang, og så henter den det den trenger fra bufferet etter det. Dette gjør at de ikke trenger å lese så mye fra disk, noe som er bra.

Verdier som er hentet fra memory tidligere, ligger ikke nødvendigvis der fortsatt. Og hvis dataene du leter etter ikke fins der, så må det leses fra disk i stedet. Disse nye dataene vil så legges inn i memory, og hvis det ikke lenger er ledig plass i memory, så lages det plass ved at de eldste dataene skyves ut.

Top 5 Timed Events

Hvilke 5 events i databasen som bruker mest tid for denne rapporten akkurat nå – av alle events. For de fleste databaser som ikke har problemer, så vil "CPU Time" wait være blant top 5.

Eks: isolasjonsnivå: read committed, ant. virtuell brukere: 2

Top 5 Timed Events ~~~~~~~~~~~~~~~~~~~ *Event*	*Waits*	*Time (s)*	*Avg % wait (ms)*	*Total Call Time*
enq: TX - row lock contention	*294*	*861*	*2930*	*33.4*
db file sequential read	*83,161*	*529*	*6*	*20.5*
CPU time		*424*		*16.4*
log file parallel write	*3,728*	*389*	*104*	*15.1*
control file parallel write	*2,542*	*148*	*58*	*5.7*

Top 5 events listes opp i synkende rekkefølge for wait time for når timed_statisitcs (parameter som kan settes i databasen og brukes til ytelses-tuning) er satt som true i databasen, som så viser en %wait time for hvert attributt.
I eksemplet over så ser det ut til at 33% av all wait time ble brukt til å vente for *'enq: TX - row lock contention'* , mens 21% av wait time ble brukt på å vente for *' db file sequential read'*.
I en tuningssituasjon kan man så se nærmere på hva som bruker så lang tid.

- **enq: TX - row lock contention - waits**

Enq = Enqueues er låser som koordinerer aksess til databaseressurser,
Enq: wait event indikerer at sesjonen (session) venter på en lås som holdes av en annen

sesjon. TX låser må være satt som exclusive mens transaksjonen gjør endringer, og låses opp først når transaksjonen gjør en COMMIT eller ROLLBACK

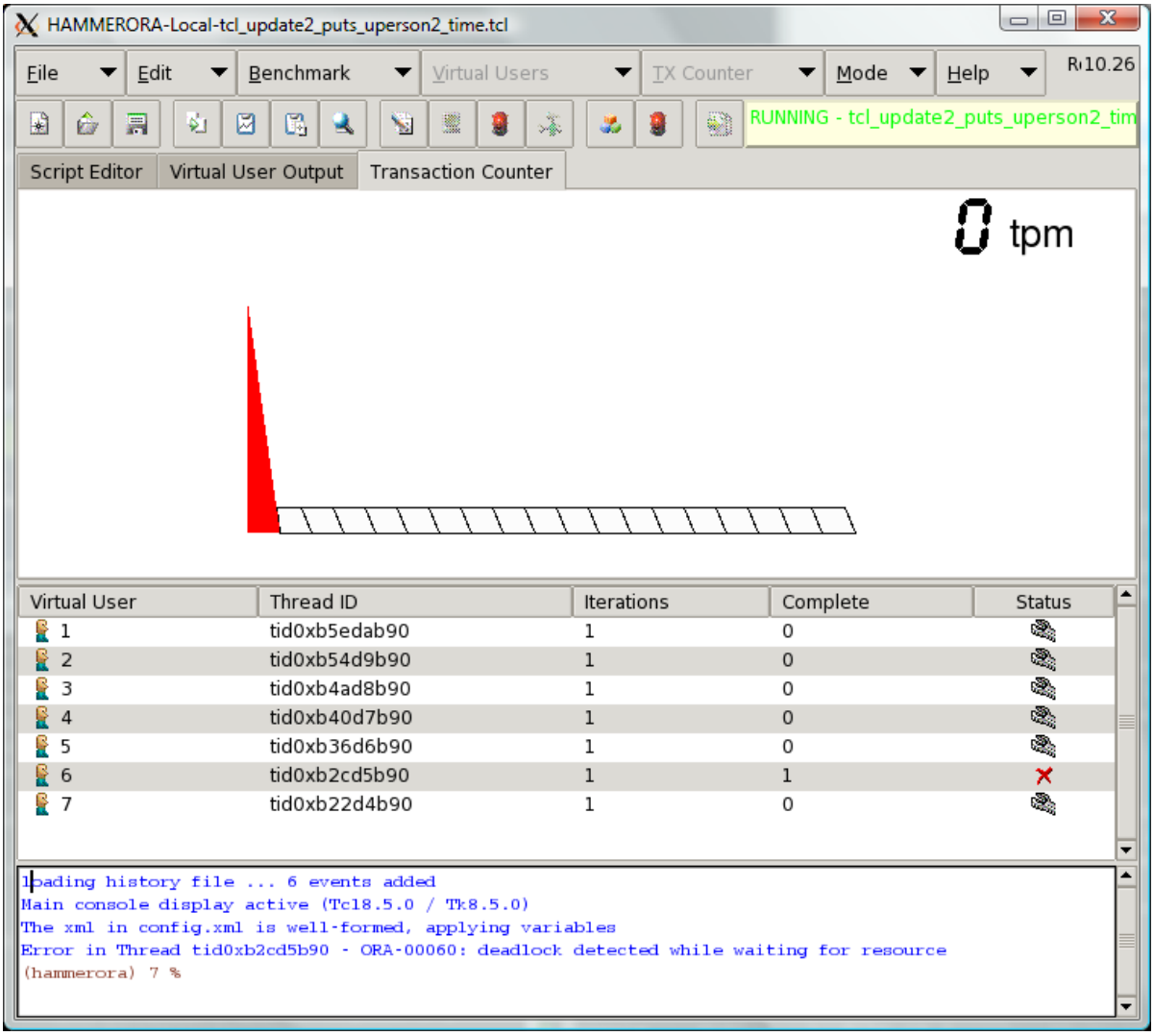
Det kan være forskjellige grunner til at at TX-låser oppstår, men jeg trekker frem den mest sannsynlige grunnen her, som er at en sesjon venter på en rad nivå lås som allerede holdes av en annen sesjon. Denne situasjonen oppstår når en bruker oppdaterer eller sletter en rad som er den samme som den andre sesjonen ønsker å oppdatere eller slette. Denne typen av TX-wait-låser korrespenderer til wait eventen enq: TX –row lock contention.

Users/ isolasjonN	2 Waits	3	4	7	10	25
Serializable	266	556	586	1,145	2,302	-
Read Committed	294	802	2,892	7,617	2,476	

Forskjell: Read committed har de høyeste wait-tidene.

Viser under et eksempel fra en av Hammerora-kjøringene:

Isolasjonsnivå: read commited, Ant virtuelle brukere: 7
Her ser du at første deadlock er kommet i Hammerora:




```
SQL> SELECT substr(DECODE(request,0,'Holder: ','Waiter: ')||sid,1,12) sess,
      id1, id2, lmode, request, type, inst_id
 FROM GV$LOCK
WHERE (id1, id2, type) IN
  (SELECT id1, id2, type FROM GV$LOCK WHERE request>0)
    ORDER BY id1, request;

SESS                ID1      ID2     LMODE   REQUEST TY    INST_ID
------------------- ---------- ---------- ---------- ---------- -- ----------
Holder: 140         393241   121251        6          0 TX          1
Waiter: 143         393241   121251        0          6 TX          1
Waiter: 147         393241   121251        0          6 TX          1
Holder: 132         524312   121072        6          0 TX          1
Waiter: 140         524312   121072        0          6 TX          1
Waiter: 134         524312   121072        0          6 TX          1
Waiter: 130         524312   121072        0          6 TX          1
Waiter: 137         524312   121072        0          6 TX          1
```

8 rows selected.

Ser at sesjon 140 blokkerer for sesjon 143 og 147, mens 132 blokkerer for 134, 130 og 137.

Denne selecten vider hvordan finne frem til hvilken rad som blokkeres:

SQL> select * from v$lock ;

ADDR	KADDR	SID	TY	ID1	ID2	LMODE	REQUEST	CTIME	BLOCK
51434398	514343AC	140	TX	524312	121072	0	6	430	0
51434450	51434464	143	TX	393241	121251	0	6	948	0
51434508	5143451C	147	TX	393241	121251	0	6	948	0
51434958	5143496C	130	TX	524312	121072	0	6	674	0
514349B4	514349C8	137	TX	524312	121072	0	6	674	0
51434A10	51434A24	134	TX	524312	121072	0	6	674	0
50F5184C	50F51864	143	TM	91684	0	3	0	1226	0
50F518F8	50F51910	147	TM	91684	0	3	0	1226	0
50F519A4	50F519BC	140	TM	91684	0	3	0	1226	0
50F51A50	50F51A68	137	TM	91684	0	3	0	1223	0
50F51AFC	50F51B14	134	TM	91684	0	3	0	1223	0
50F51C54	50F51C6C	130	TM	91684	0	3	0	1223	0
50FA1DB4	50FA1DD8	132	TX	524312	121072	6	0	674	1
50FAD994	50FAD9B8	140	TX	393241	121251	6	0	954	1

Ser at to sesjoner holder låser som blokkerer andre sesjoner.
Henter informasjon om BLOCK=1, og sammenligner ID1 og ID2 for å se hvem som blokkerer hvem.

140:

50F519A4	50F519BC	140	TM	91684	0	3	0

Identifiserer det låste objektet:

SQL> select object_name from dba_objects where object_id=91684 ;

OBJECT_NAME

FILMPARTICIPATION

Identifiserer den låste raden:

```
SQL> select do.object_name,
row_wait_obj#, row_wait_file#, row_wait_block#, row_wait_row#,
dbms_rowid.rowid_create ( 1, ROW_WAIT_OBJ#, ROW_WAIT_FILE#, ROW_WAIT_BLOCK#,
ROW_WAIT_ROW# )
from v$session s, dba_objects do
```

```
where sid=140 and s.ROW_WAIT_OBJ# = do.OBJECT_ID ;

OBJECT_NAME
--------------------------------------------------------------------------------
ROW_WAIT_OBJ# ROW_WAIT_FILE# ROW_WAIT_BLOCK# ROW_WAIT_ROW#
DBMS_ROWID.ROWID_C
------------- -------------- --------------- ------------- ------------------
FILMPARTICIPATION
      91684              6         165895           254 AAAWYkAAGAAAogHAD+
```

Deretter se på selve raden:

```
SQL> select * from FILMPARTICIPATION where rowid = 'AAAWYkAAGAAAogHAD+';

    PARTID     FILMID  PARTTYPE        PERSONID
---------- ---------- ---------- ----------
  26363826    1514662 cast            3039401
```

Med SID kan man finne SQL som kjøres:

```
SQL> select s.sid, q.sql_text from v$sqltext q, v$session s
where q.address = s.sql_address
and s.sid = &sid
order by piece;

Enter value for sid: 140
old   3: and s.sid = &sid
new   3: and s.sid = 140

      SID SQL_TEXT
---------- ----------------------------------------------------------------
      140 update filmparticipation set parttype = 'producer' where personi
      140 d IN (select p.personid from person p, filmparticipation fp wher
      140 e p.personid = fp.personid and fp.parttype like '%' and p.person
      140 id IN (select bi.personid from biographyitem bi, person p where
      140 bi.personid = p.personid ))

SQL> select s.sid, q.sql_text from v$sqltext q, v$session s
where q.address = s.sql_address
and s.sid = &sid
order by piece;

Enter value for sid: 132
old   3: and s.sid = &sid
new   3: and s.sid = 132

      SID SQL_TEXT
---------- ----------------------------------------------------------------
      132 update filmparticipation set parttype = 'producer' where personi
      132 d IN (select p.personid from person p, filmparticipation fp wher
```

```
132 e p.personid = fp.personid and fp.parttype like '%' and p.person
132 id IN (select bi.personid from biographyitem bi, person p where
132 bi.personid = p.personid ))
```

- **db file sequential read - waits**

Sesjonen venter mens en sekvensiell lesing (lese en enkelt blokk i en tabell/index) fra databases utføres.
Wait time er den tiden det tar å gjøre I/O.

Users/ isolasjonN	2	3	4	7	10	25
Serializable	17,272	22,129	29,579	48,003	35,564	70,411
Read Committed	83,161	25,389	-	413,952	40,384	

Forskjell: Serializable har betraktelig mye lavere wait time enn det read committed har.

- **log file parallel write – waits**

Denne eventen skjer i DBWR (DataBase WRiter), og indikerer at DBWR utfører en parallell skriving til filer og blokker. Når siste I/O har gått til disk, så avslutter waits. DBWR er en bakgrunnsprosess i Oracle, som skriver data fra SGA til Oracle databasefiler.

Wait time er wait inntil alle I/O er fullført.

Users/ isolasjonN	2	3	4	7	10	25
Serializable	1,960	1,885	1,927	1,812	1,940	1,948
Read Committed	3,728	5,511	6,884	3,702	767	

Forskjell: Read committed har jevnt over mer wait time enn det serializable har.

- **CPU time**

Ant CPU brukt av denne sesjonen.
Call time er den totale tiden brukt på database kall.
Hensikten er å identifisere hva som bruker mest tid, for så å forbedre sql-en som er brukt.

Users/ isolasjonN	2 Time/Total Call Time	3	4	7	10	25
Serializable	289 /17.1	333/ 12.5	404 /13.6	678/11.3	656/6.8	712 / 22.8
Read Committed	424 /16.4	605/12.3	-	635/2.4	219/2.2	

Det er en fordel om at det ventes på CPU og ikke på disk, da dette kan tyde på at databasen trenger mer minne og derfor benytter CPU'en mer effektivt.
Dette må ses opp mot faktisk tid ventet.

Forskjell: Read committed har brukt lengre tid opp til 3 brukere. Fra og med 4 brukere er det serializable som har brukt mest tid.

Host CPU - Load Average

Users/ isolasjonN	2 User/system/ WIO	3	4	7	10	25
Serializable	9.69/3.33/30.54	14.13 / 4.30 / 41.50	12.44 / 3.96 / 33.81	11.0/3.85/28.41	14.18/3.45/29.33	15.68 / 4.14 / 41.40
Read Committed	8.61/3.44/36.08	11.73 / 4.65/ 51.03	20.30 / 4.20/ 49.46	9.89/3.11/27.64	20.99/4.25/32.68	

WIO – wait for I/O. Transaksjonen venter på å lese/skrive pga. at andre brukere/systemet henter noe fra disk

Forskjell:

User: Bortsett fra for 4 brukere, så var det omtrent like mange høye/lave verdier.
Read committed har en topp på 4 brukere.
System: Ganske like resultater
WIO: Generelt høyere verdier for read committed, bortsett fra for 7 brukere der serializable hadde en høyere verdi

Statistic

Eks:

Statistic	Time (s)	% of DB time
sql execute elapsed time	14,214.2	99.7
RMAN cpu time (backup/restore)	1,337.8	9.4
DB CPU	828.3	5.8
parse time elapsed	120.4	.8
hard parse elapsed time	115.0	.8
PL/SQL compilation elapsed time	23.4	.2
PL/SQL execution elapsed time	12.4	.1
connection management call elapsed	7.8	.1
inbound PL/SQL rpc elapsed time	1.9	.0
hard parse (sharing criteria) elaps	0.9	.0

repeated bind elapsed time	0.1	.0
hard parse (bind mismatch) elapsed	0.1	.0
sequence load elapsed time	0.0	.0
failed parse elapsed time	0.0	.0
DB time	14,255.1	
background elapsed time	13,325.2	
background cpu time	1,594.9	

- **parse time elapsed**

Users/ isolasjonN	2 Time/ DB time	3	4	7	10	25
Serializable	28.1 / 1.5	3.7 / .1	30.5 / .9	90.3/1.2	45.4/.3	84.2 / .6
Read Committed	8 /.3	39.2 / .8	120.4 / .8	111.0/.4	79.4/.5	

Forskjell: Read Committed hadde de fleste høyere verdier.

- **DB CPU**

Users/ isolasjonN	2	3	4	7	10	25
Serializable	282,5/15.2	329.3 / 11.3	404.6 / 11.7	676.4/8.8	775.9/5.6	707.0 / 5.0
Read Committed	420.8/14.8	39.2 / .8	828.3 / 5.8	943.5/3.3	724.0/4.9	

2 users.
Read committed har en mye høyere verdi enn serializable.
Muligens er grunnen fordi serializable kjører med kun en user, siden den andre feilet.
Mens read committed kjører med 2 users inntil det stopper opp pga deadlock. Siden begge users kjører, så er også verdien nesten doblet.

- **DB time**

Total tid brukt for sessions

Users/ isolasjonN	2 time/DB time	3	4	7	10	25
Serializable	1,860.8	2,903.8	3,459.3	7,649.9	13,765.7	14,258.4
Read Committed	2,838.0	5,139.4	14,255.1	28,523.4	14,724.0	

Forskjell: Read committed er den som har brukt mest tid.

Eksempler på statspack-resultater fins her:
http://folk.uio.no/vibekes/read_comm_4users_nyest.log
http://folk.uio.no/vibekes/read_comm_7users_hang.log

6.4 Problemer underveis

Problemer som oppsto under prosjektet:

Import problem: Jeg fikk problemer under import fra Postgres-databasen til min bruker i Oracle-databasen. Enkelte tabellfelt i Postgres var ikke satt opp med en størrelse, slik at når mine tabeller ble opprettet i Oracle med en antatt størrelse, så var det en sjanse for at størrelsen på feltet ikke var stor nok. Et eksempel er tabellen `biographyitem` og feltet `bdesc` som tilslutt måtte settes til `VARCHAR2(600)` for å få importen til å gå i orden. Grunnen til at feltet måtte settes så stort, var fordi det gjerne lå hele fortellinger i dette feltet.

Diskplass: Diskplassen måtte økes flere ganger, da det ble for lite både under import og under testkjøringer

Databasetilgang: Min databasebruker mistet tilgang til databasen. Jeg fikk nye rettigheter slik at jeg igjen fikk logget meg på databasen

Tablespace'ene gikk fulle og måtte utvides

Alert-logg: Jeg måtte ha tilgang til alert-loggen for å kunne se feilmeldinger som oppsto i databasen

Trace-filer: Jegmanglet tilgang til trace filer som trengtes til kjøring i Hammerora

Å finne den rette update-setningen som hadde en passende lengde på tid var heller ikke likeså greit

7. Resultat

7.1 Hammerora resultater

Dette er resultatene etter Hammerora-kjøring:
Tabellene under viser hvilket resultat de enkelte transaksjons-kjøringene fikk i Hammerora.

Følgende intervaller er kjørt:
2 virtuelle brukere for isolasjonsnivå read committed (read comm)
2 virtuelle brukere for isolasjonsnivå serializable (seriali)
3 virtuelle brukere for isolasjonsnivå read committed (read comm)
3 virtuelle brukere for isolasjonsnivå serializable (seriali)
4 virtuelle brukere for isolasjonsnivå read committed (read comm)
4 virtuelle brukere for isolasjonsnivå serializable (seriali)
7 virtuelle brukere for isolasjonsnivå read committed (read comm)
7 virtuelle brukere for isolasjonsnivå serializable (seriali)
10 virtuelle brukere for isolasjonsnivå serializable (seriali)
10 virtuelle brukere for isolasjonsnivå read committed (read comm)
25 virtuelle brukere for isolasjonsnivå serializable (seriali)

Isoleringsnivå	2 read com	2 seriali	3 read comm	3 seriali	4 read comm	4 read comm	4 seriali
Fullført	2stk OK tot. 30min	2: OK etter 15m. 1 fikk ORA-08177: can't serialize acces or this transaction	Alle 3 OK. 1: 29min, 2: 15min, 3: 41min	3:OK etter 17min, rest: ORA-08177: can't serialize acces or this transaction	2 etter 7m: ORA-00060: deadlock detected while waiting for resurces	4stk OK 1: 1t20m 2; 39min 3:54min 4: 1t7min	2: OK 19m rest: ORA-08177: can't serialize

Isoleringsnivå	7 read comm	7 seriali	10 read comm	10 seriali	25 seriali
Fullført	6: etter 15m ORA-00060: deadlock	7: OK etter 24min,rest; ORA-08177: can't serialize	2 etter 15m ORA-08177: can't serialize	3: OK etter 27min, 7 stk ORA-08177: can't serialize, 2 manglet	12: OK etter 28min, resten ORA-08177: can't serialize,

Første nummeret hentyder til hvilken virtuell bruker som det snakkes om.
Eks. nivå serializable 7 users:
Virtuell user 7 fullførte etter 24 min, og de andre virtuelle brukerne (vusers) feilet med feilmeldingen "ORA-08177: can't serialize"

Read Committed:

Opp til 4 samtidige brukere gikk det som regel greit å kjøre.Fra 7 og oppover, kom det feilmeldinger om deadlock etter ca. 15min.

Jeg prøvde å la kjøringen stå og gå inntil den fullførte, men ga opp etter at første deadlock ble avløst med en ny deadlock etter 8 timer. Jeg hadde ikke tid til å la kjøringen stå og gå inntil den fullførte eller feilet, og avsluttet kjøringen en liten stund etter at første deadlock feilmelding kom.

Hvis man har muligheten for å la kjøringen gå til den avslutter av seg selv, så vil nok databasen selv løse opp i deadlock'ene med enten å avbryte den ene sesjonen, eller rulle den tilbake for å kjøre den på nytt senere.

Serializable:

På dette isoleringsnivået gikk det kun én bruker igjennom og fullførte. Uansett hvor mange andre samtidige brukere som ble startet (jeg testetet opp til 25 brukere), så fikk jeg feilmeldingen "ORA-08177: can't serialize acces for this transaction" for alle bortsett fra den ene som fullfører.

Hvis jeg kun skal tenke på resultatene jeg fikk fra Hammerora, og hvilket isolasjonsnivå som fikk igjennom flest transaksjoner, så er det isolasjonsnivå read committed som var beste alternativ opp til 4 samtidige brukere, da alle brukerne fullfører. Isolasjonsnivå serializable hadde kun én bruker som fullførte. Hvis man derimot ser på fra 7 samtidige brukere, så var serializable, som fullførte med en bruker, et bedre alternativ enn read committed som får deadlock og stopper helt opp. Ulempe: kun én transaksjon går igjennom på serializable

7.2 Statspack resultater

Hvis jeg ser på alle resultatene fra seksjon 6.3 for statspack-kjøringene, så har read committed de fleste høye verdiene som skulle tilsi at dette er et dårligere isolasjonsnivå for samtidige kjøringer. Men før man kan trekke noen konklusjon på det, så må man også se på forholdene rundt kjøringene. Jeg fikk dessverre ikke testet skikkelig siden transaksjonene med isolasjonsnivå read committed ble avbrutt ved første deadlock i Hammerora da transaksjonskjøringen tok for lang tid, og at min putty-sesjon ikke kunne opprettholde forbindelsen. Dette har nok hatt ganske mye å si på testresultatene.

I tillegg så ville ikke isolasjonsnivå serializable kjøre med mer enn én bruker.

Å kode transaksjoner som skal kjøre med isolasjonsnivå serializable krever ekstra arbeid, for å sikre at feilmeldingen " ORA-08177: Cannot serialize access" ikke oppstår. [18] Dette ble ikke tatt høyde for under min testing, og er grunnen til at denne feilmeldingen kom.

Isolasjonsnivå serializable, slik den er implementer i oracle ved snapshot isolation, er mest passende for miljøer der det er relativt liten sjanse for at to samtidige transaksjoner vil endre samme rad. For min testsituasjon er det jo dette jeg faktisk tester, og vi kan vel si det slik etter resultatene jeg fikk, så er det ikke en bra løsning.

7.3 Konklusjon

Min testsituasjon var ikke optimal for å gi et godt svar på det jeg egentlig ønsket svar på. Men det jeg fant ut var at isolasjonsnivå read committed er det nivået som må brukes for å få flest mulige samtidige transaksjoner igjennom. Brukes isolasjonsnivået serializable vil man få store problemer med samtidigheten. Med miljøer som har høy fokus på ytelse og har en høy andel av transaksjonsgjennomstrømming, så kreves det en kjappere responstid enn det som kan fås av serializable.

8. Videre arbeid

Hva jeg ikke rakk

På grunn av tidsnød fikk jeg ikke testet med stor variasjon i antall virtuelle brukere. Planen var å teste opp til i hvert fall 100 samtidige brukere.

Det betyr at mitt testutvalg var litt mindre enn jeg håpet på, da jeg startet å skrive.

Videre arbeid

Kjøre flere tester med flere brukere, og gjerne kjøre om igjen de andre testene.

Dette for å sikre at jeg har luket unna andre faktorer som muligens har spilt en rolle vedr. ytelsen på målingene. Faktorer kan være RMAN backup, andre bakgrunnsprosesser som stjeler CPU fra meg.

Omprogrammere for å forhindre feilmeldingen ’ORA-08177: Cannot serialize access’.

9. Bibliografi

[1] http://hammerora.sourceforge.net/

[2] http://www.tpc.org/

[3] https://www.usit.uio.no/it/programvare/base/produkt.html?id=2181

[4] http://download.oracle.com/docs/cd/B10501_01/server.920/a96533/statspac.htm

[5] http://www.oracle.com/technology/oramag/oracle/05-nov/o65asktom.html

[6] http://www.akadia.com/services/ora_statspack_survival_guide.html

[7] A critique of ANSI SQL Isolation levels, s,1-10, 1995, Hal Berenson, Phil Bernstein, Jim grey, Jim melton, Elizabetg O'Neill, Patrick O'Neill

[8] En teoretisk studie av "Snapshot Isolation", Lene T. Østby, 2006

[9] http://en.wikipedia.org/wiki/Isolation_(database_systems)

[10] http://www.postgresql.org/about/

[11] http://www.tcl.tk/

[12] http://www.tpc.org/tpce/default.asp

[13] http://www.oracle.com/technology/deploy/performance/pdf/statspack.pdf

[14] http://en.wikipedia.org/wiki/Oracle_database

[15] http://en.wikipedia.org/wiki/Snapshot_isolation#Making_Snapshot_Isolation_Serializable

[16] Mihalis Yannakakis. Serializability by Locking. Journal of the ACM, 31(2):227–244, apr 1984.

[17] http://www.oracle.com/technology/oramag/oracle/05-nov/o65asktom.html

[18] http://download.oracle.com/docs/cd/B13789_01/server.101/b10743/consist.htm