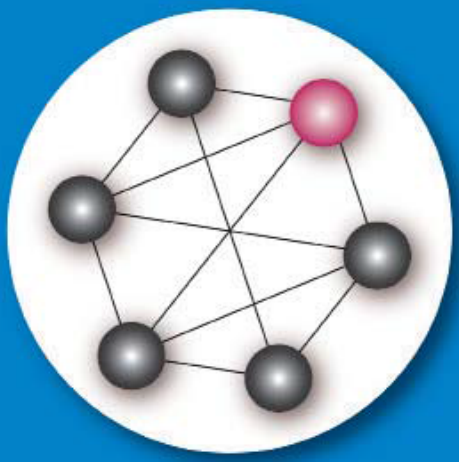


DBTechNet

DBTech VET

SQL Transactions

Teoriaa ja käytännönharjoituksia

Suomeksi

Lifelong Learning Programme

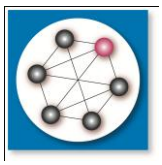
SQL-transaktioiden käytännön teoriaa ja harjoituksia

Versio 0.5, maaliskuu 2014

Tekijät: Martti Laiho ja Mika Wendelius

Tämä on vapaasti ja täydentäen käännetty suomenkielinen laitos EU LLP LdV projektin DBTech VET Teachers (DBTech VET, Code: 2012-1-FI1-LEO05-09365) kirjasta:
"SQL Transactions – Theory and Hands-on Exercises"
Version 1.3 of the first edition 2013
Authors: Martti Laiho, Dimitris A. Dervos, Kari Silpiö

DBTech VET is Leonardo da Vinci Multilateral Transfer of Innovation project,
funded by the European Commission and project partners.



www.DBTechNet.org
DBTech VET

SQL-transaktioiden käytännön teoriaa ja harjoituksia

Tarkoitus

Tietokantojen tietoihin tulee voida luottaa. Tämä edellyttää sovelluksilta luotettavaa tietokantakäsittelyä, mikä tulee toteuttaa oikeaoppisesti ohjelmoituina transaktioina ja nollatoleranssilla tietovirheille. Ohjelmoija, joka ei tunne tarvittavaa transaktioteknologiaa, vaarantaa tietokannan tietojen eheyden, hakee sovellukseen vääristyneitä tietoja ja voi hidastaa samanaikaisten muiden käyttäjien työtä. Liikennesääntöjen tapaan tietokantakäsittelyn säännöt on tunnettava ja niitä on noudatettava.

Tämän tutorialin tarkoitus on esittää transaktio-ohjelmoinnin perusteet keskeisten tietokantaohjelmistojen osalta. Tutorial kuvaa tyypilliset käsittelyn ongelmat ja näiden ratkaisemiseen tarvittavat transaktioiden säädöt. Tutorial esittelee käsitteet sovelluskehittäjän näkökulmasta. Tässä tutorialin suomenkielisessä versiossa terminologia on tarkoituksella lähellä englanninkielisiä termejä, koska alan kirjallisuus ja käsikirjat ovat pääasiassa englanninkielisiä ja ammattilaisen on tunnistettava termit tuotekohtaisissa käsikirjoissa ja osattava kommunikoida ulkomaalaisten kolleegojen kanssa.

Kohderyhmät

Tutorial on tarkoitettu ammatillisten oppilaitosten ja ammatillisesti suuntautuneiden korkeakoulujen opettajille, ohjaajille/valmentajille sekä opiskelijoille. Myös alan ammattilaisille tutorial voi tarjota hyödyllistä tietoa niiden johtavien tietokantatuotteiden ominaisuuksista, joita ei päivittäin ole itse käyttänyt.

Tarvittavat esitiedot

Lukijalla tulisi olla perustiedot SQL-kielestä ja kokemusta jonkin tietokantatuotteen peruskäytöstä.

Opiskelumenetelmät

Transaktio-ohjelmoinnin taitoja ei opita pelkästään kirjoista. Tutorialiin liittyy ilmainen virtuaalikone, jonka Linux-alustalle on asennettu valmiiksi ilmaiset kokeiluversiot DB2 Express-C, Oracle XE, MySQL, PostgreSQL ja Pyrrho-järjestelmistä. Lukijaa kehoitetaankin kokeilemaan itse tutorialin esimerkkejä ja harjoituksia, mielellään useampaakin virtuaalilaboratoriomme järjestelmää käyttäen. Järjestelmät käyttäytyvät eri tavoin. Ei kannata luottaa siihen, että aikaisemmin opittu pätee kaikissa järjestelmissä ja versioissa. Tulevan ammattilaisen opiskelumallin tuleekin olla ”Learning by Verifying”. Tutorialin esimerkkien skriptit löytyvät virtuaalilaboratorion ”Transactions”-hakemiston tiedostoista mahdollistaen nopean copy-paste -käytön.

Sisällysluettelo

1 SQL-transaktioista

1.1 Johdantoa

Jokapäiväisessä elämässämme teemme monenlaisia liiketoimintatransaktioita: ostamme tuotteita, tilaamme matkoja, muutamme tai peruutamme tilauksiamme, ostamme konserttilippuja, maksamme vuokria, sähkölaskuja ja vakuutusmaksuja, jne. Transaktio on looginen aktiviteettikokonaisuus, joka suoritetaan tai peruutetaan kerralla kokonaan. Kaikki transaktiot eivät edellytä tietokoneiden käyttöä, mutta tietotekniikan osuus kasvaa jatkuvasti.

Lähes kaikki tietojärjestelmät/sovellukset käyttävät tietokantajärjestelmiä (DBMS) tietojen talletukseen ja hakuun. Nykyajan tietokantatuotteet ovat teknisesti huippuun hiottuja, monimutkaisia ohjelmistoja, jotka suojaavat talletettujen tietojen eheyttä ja mahdollistavat tietojen nopean haun jopa monille samanaikaisille käyttäjille. Ne tarjoavat sovelluksille luotettavat tietojen talletuspalvelut, mutta vain jos sovellukset käyttävät niiden luotettavia palveluita oikeaoppisesti. Tämä toteutetaan ohjelmoimalla tietokantojen käsittely käyttäen tietokantatransaktioita. Tässä kirjasessa tarkastelemme tietokantatransaktioiden toteutusta SQL-kielellä eli SQL-transaktioita. Liitteessä 2 on esimerkki tietokantransaktion ohjelmoinnista Java-kielellä käyttäen JDBC API:a.

Puutteellinen transaktio-ohjelmointi voi johtaa tietojen vääristymisiin tai tietojen hukkaamisiin, esimerkiksi

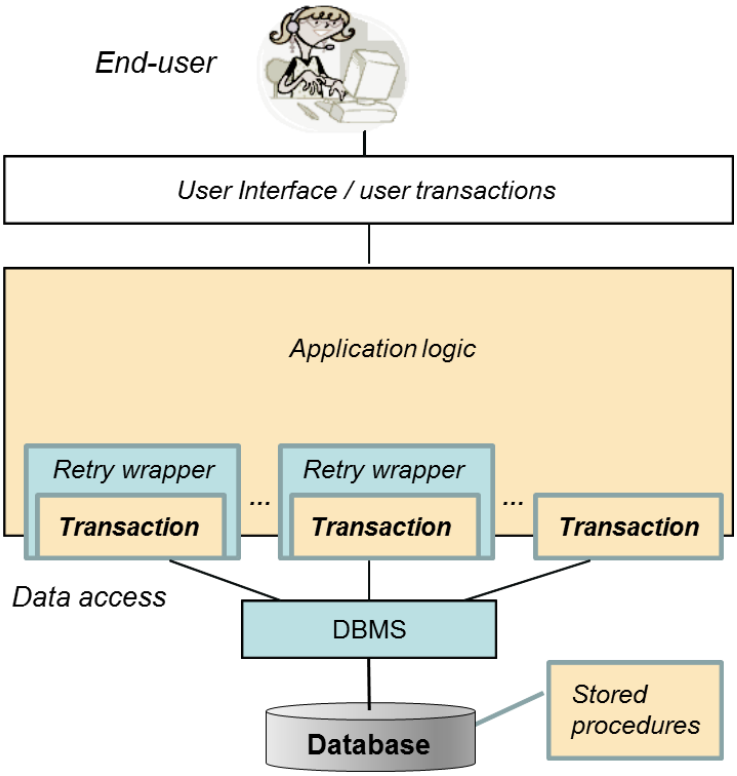
- verkkokauppaostoksen tilaus, maksu tai toimitus voi jäädä tekemättä tietokantaan
- juna-, bussi- tai lentomatkan paikanvarausjärjestelmässä istuinpaikalle voi tulla kaksoisvarauksia
- hälytyskeskuksen kirjaama hälytys voi hukkua tai vääristyä
- jne

Melkein jokainen on törmännyt joihinkin vastaaviin ongelmiin tosielämässä ja joskus todella vakavista ongelmista, varsinkin katastrofeista, kerrotaan myös uutisissa. Läheskään kaikista ei kuitenkaan haluta kertoa julkisuudessa.
DBTech VET projektin missio on saada tulevat ammattilaiset kiinnostumaan luotettavan transaktio-ohjelmoinnin tärkeydestä ja parhaista käytännöistä, jotta tällaisilta tietojen vääristymisiltä ja hukkaamisilta vältyttäisiin..

Transaktiot ovat tietokantakäsittelyn kokonaisuuksia, joista vikatilanteissa voidaan toipua ja joiden kirjatun lokihistorian perusteella vioittunut tietokanta voidaan tarvittaessa palauttaa viimeisen onnistuneen transaktion jälkeiseen tilaan, kuten liitteessä 3 kuvataan. Transaktiot muodostavat myös perustan luotettavalle samanaikaisten käyttäjien palvelulle monen käyttäjän tietokantaympäristössä.

1.2 SQL-ympäristön Client/Server –käsitteistöä

Tässä tutorialissa käytämme tietokantatransaktioita vuorovaikutteisella SQL-kielellä, mutta on hyvä muistaa, että sovellusohjelmoinnissa tietokantoja käytetään API-liittymillä ja **transaktioprotokolla** on näissä hieman erilainen. Tästä esimerkkinä on liitteen 2 Java-ohjelma, jossa JDBC:n transaktioprotokolla osittain yhdenmukaistaa käytettävien DBMS-järjestelmien transaktioprotokollaa. SQL:n lisäksi tietokannan käsittelykieliä on muitakin, kuten esimerkiksi XQuery, ja tietokantatasolla nämä voivat käyttää samaa transaktioprotokollaa.

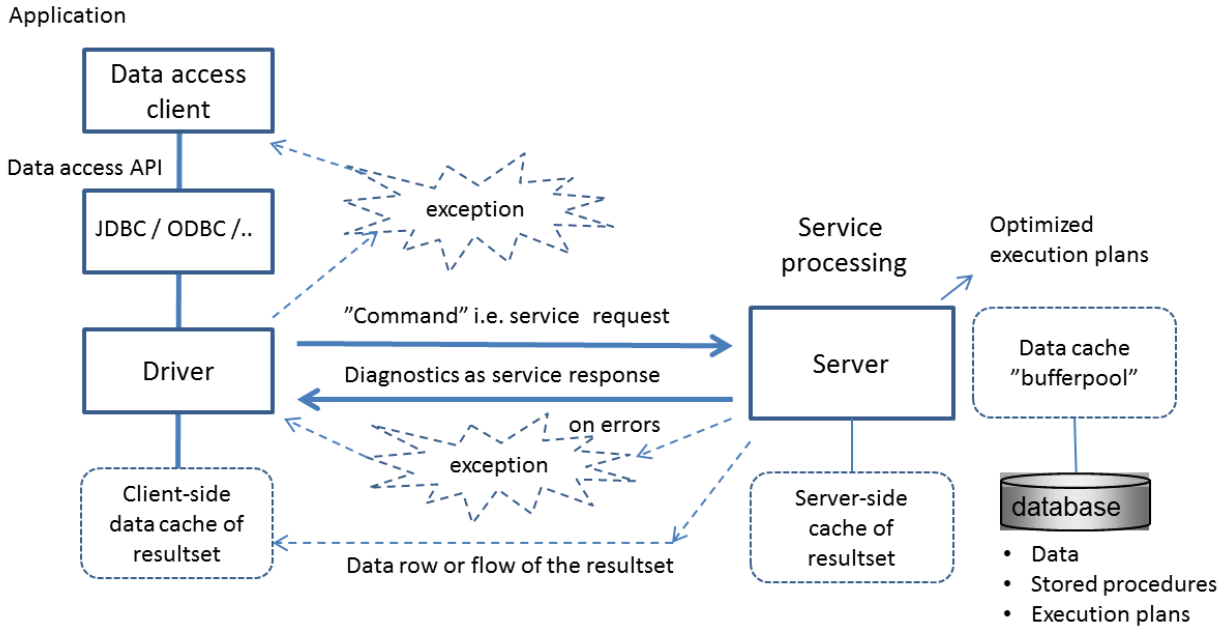


Kuva 1.1 SQL-transaktioiden sijainti sovelluskerroksissa

Kuva 1.1 esittää yksinkertaistettua sovellusarkkitehtuuria, missä tietokantakäsittely tapahtuu sovelluslogiikan kerroksessa erillään käyttöliittymäkerroksesta. Loppukäyttäjän näkökulmasta **liiketoimintatransaktio** (business transaction) toteutetaan sovellussuunnitelmassa yhdellä tai useammalla **käyttötapauksella** (use case), jotka toteutetaan käyttöliittymässä **käyttäjä-transaktioina** (user transaction). Käyttäjätransaktion sovelluslogiikassa käydään dialogia käyttäjän kanssa ja tietokannan kanssa. Tietokantadialogit (data access dialog) tulee toteuttaa käyttäjädialogeista erillisinä **tietokantatransaktioina**. Niillä haetaan aluksi tietoja käyttäjädialogeja varten ja tyypillisesti vasta käyttäjätransaktion viimeinen tietokantatransaktio kirjaa muokatut tiedot tietokantaan. Osa transaktiologiikasta voi olla toteutettuna tietokantaan talletettuina proseduureina (stored procedure), jotka voivat palvella myös useita transaktioita.

Kuten myöhemmin näemme, oikeinkin toimiva tietokantatransaktio saattaa törmätä muiden käyttäjien kilpaileviin transaktioihin ja tulla perutuksi. Yhteentörmäyksistä ei kannata heti valittaa käyttäjälle, sillä transaktion joku uusintayritys (retry) saattaa onnistua. Oppikirjoissa saatetaan väittää, että tietokantajärjestelmä käynnistäisi kaatuneen transaktion uudelleen. Tämä ei pidä paikkaansa, vaan uusintayritykset ovat sovelluksen (tai sovelluspalvelimen) vastuulla. Uusintayritysten käynnistyslogiikan ohjelmointimallista (data access pattern) käytetään nimitystä **Retry Wrapper**. Uusintayritykset kuluttavat resursseja ja hidastavat käyttäjän kokemaa **vastausaikaa**, joten niitä kannattaa tehdä vain äärellinen määrä, minkä jälkeen kannattaa luovuttaa ja kertoa käyttäjälle, ettei tietokantakäsittely ruuhkan vuoksi nyt onnistu.

Ymmärtääkseemme tietokantatransaktioita, meidän kannattaa sopia muutamista tietokanta-dialogien asiakas-palvelin -mallin (**Client-Server**) peruskäsitteistä. Tietokantakäsittelyä varten sovellus tarvitsee tietokantayhteyden (**database connection**). Avoin tietokantayhteys muodostaa kontekstin SQL-istunnolle (**SQL-session**). SQL-istunnon aloittanutta sovellusprosessia kutsutaan tässä yhteydessä **SQL-client**'iksi ja tietokantapalvelinta **SQL-server**'iksi[1]. SQL-istunnossa SQL-client käyttää SQL-server'in palveluja lähettämällä palvelimelle palvelupyyntöjä (service request). Tietokantapalvelimen kannalta palvelupyynnöt ovat SQL-komentoja (SQL command), jotka muodostuvat yhdestä tai useammasta SQL-kielen lauseesta (SQL statement) tai SQL-istunnon säädöstä. Sovelluksen käyttämän tietokantaliittymän API-tasolla (application programming interface)[2] palvelupyynnöt ovat funktio/metodi-kutsuja, joissa SQL-komennot välitetään palvelimelle parametreina. Matalammalla tasolla palvelupyynnöt ovat tyypillisesti TCP/IP –protokollalla välitettyjä RPC-kutsuja.



Kuva 1.2 SQL-komennon käsittelyvaiheet client-server dialogissa

Kuva 1.2 selittää SQL-komennon kierrosmatkaa (round trip) alkaen SQL-clientin palvelupyynnöstä API:n tietokanta-ajurin (driver) välityksellä ja verkkopalvelun läpi palvelimelle. Palvelin jäsentää komennon SQL-lauseet, analysoi ne systeemitaulujen tietoja vasten tarkistaen myös käsittelyoikeudet, laatii komennolle suoritussuunnitelman (execution plan), tai etsii valmiin suoritussuunnitelman, optimoi suoritussuunitelman ja käsittelee sen. Tietojen haku levyltä on hidasta verrattuna hakuihin keskusmuistissa, joten jo aikaisemminkin levyiltä haetut rivit pyritään säilyttämään palvelimen keskusmuistista käytössä olevassa puskurialtaassa ja käsiteltävät rivit haetaan levyltä vain jos niitä ei ole haettu jo puskurialtaaseen. SQL-komennon suoritus **atominen** siten, että koko komennon on onnistuttava tai koko komento peruutetaan.

Käsittelyn statistiikkatiedoista kootaan SQL-clientille palautettava diagnostiikkavastaus, kattaen käsiteltyjen rivien luumäärät, mahdolliset varoitukset (SQL warnings). Käsittelyvirheistä palautetaan SQL-poikkeukset (SQL exceptions), joista client-pään ajuri laukaisee sovelluksen käsiteltäväksi esimerkiksi Java-kielen tapauksessa javan exceptionit.

[1] Termi SQL-server ei tässä tarkoita vain Microsoftin SQL Server –tuotetta.
[2] API-liittymiä ovat mm. ODBC, JDBC, ADO.NET, LINQ, jne riippuen käytettävästä ohjelmointikielestä, kuten esimerkiksi C++, C#, Java, PHP, jne.

Matalan tason (CLI) käsittelystä on hyvä tietää, että esimerkiksi SELECT-komennon tulosjoukkoa (resultset) ei palauteta suoraan SQL-clientille, vaan esimerkiksi JDBC-ajuri hakee tulosjoukon rivit erillisillä palvelupyynnöillä kooten rivit SQLresultset-olioon. Myös käyttäessämme vuorovaikutteista SQL:ää käytämme itse asiassa jotakin SQL-editor –ohjelmaa, joka hakee ja esittää meille tulosjoukon rivit sekä statistiikan.

Tärkeää on ymmärtää, että UPDATE ja DELETE –komennot onnistuvat palvelimen kannalta, vaikka eivät löytäisi yhtään riviä, mikä sovelluksen kannalta voi olla virhe. Käsitellyistä riveistä palautetaan clientille vain rivien lukumäärät. Tämän vuoksi sovelluskoodin tulee tarkastaa käsittelyn diagnostiikka mukaan lukien rivimäärä aina komennon suorituksen jälkeen.

1.3 SQL-transaktion rajoista

Tietokantadialogi muodostuu sovelluslogiikan osasta, johon kuuluu yksi tai useampi SQL-komento. Sovelluslogiikan kannalta dialogia sanotaan loogiseksi työyksiköksi (logical unit of workLUW). Jos tämän käsittelyn on oltava atominen, muodostetaan dialogista transaktio. Ideaalinen transaktio noudattaa ACID-periaatetta, jota tarkastelemme myöhemmin. Sen mukaan transaktio on riippumaton samanaikaisista muista transaktioista ja siirtää osaltaan tietokannan yhdestä ehyestä tilasta (consistent state) toiseen ehyeen tilaan. Tässä mielessä transaktion sanotaan muodostavan tietokantakäsittelyn eheysyksikön (unit of consistency). Onnistunut transaktio päätetään **COMMIT**-komennolla. Sen sijaan transaktio, jota ei voida päättää onnistuneesti, päätetään **ROLLBACK**-komennolla, joka automaattisesti peruuttaa tietokannasta kaikki transaktion tekemät muutokset. Tämä yksinkertaistaa merkittävästi sovellusohjelmointia, vapauttaen ohjelmoijan virhealttiista sarjasta käsittelyvaiheiden käänteisiä operaatioita, joiden onnistuminen monen käyttäjän ympäristössä ei aina ole edes mahdollista. Tässä mielessä transaktion sanotaan olevan tietokantakäsittelyn palautusyksikkö (unit of recovery). ROLLBACK-komento onnistuu aina. Myös transaktio, jota ei saada commitoitua, esimerkiksi tietokantayhteyden katkettua tai palvelimen kaaduttua (kuten esitetään liitteessä 3), peruuntuu automaattisesti. Useimmat DBMS-tuotteet peruuttavat transaktion automaattisesti transaktion jouduttua sarjallistuvuuskonfliktiin deadlock-uhriksi. Käsittelemme tätäkin jatkossa.

Useimmat DBMS-tuotteet, esimerkiksi SQL Server, MySQL/InnoDB, PostgreSQL ja Pyrrho toimivat oletuksena **AUTOCOMMIT**-moodissa. Tämä tarkoittaa, että jokainen SQL-komento commitoituu automaattisesti ja niitä ei voida peruuttaa. Ohjelmoija ei siis voi virhetilanteissa turvautua ROLLBACK-palveluun. Käsittely on tällöin virhealtista ja esimerkiksi tietokantayhteyden katketessa tietokanta voi jäädä epäehyeen tilaan.

> SQL-standardin mukaan SQL-istunto toimii **transaktionaalisessa moodissa** siten, että jos transaktio ei ole käynnissä, niin ensimmäinen transaktiossa mahdollinen SQL-komento aloittaa uuden transaction **implisiittisesti**. Näin toimivat esimerkiksi DB2 ja Oracle -palvelimet, joskin niiden SQL-client –ohjelmat voivat toimia autocommit-moodissa.

Vaikka SQL-istunto toimisi autocommit-moodissa, voidaan kuitenkin suorittaa myös transaktioita aloittamalla transaktio eksplisiittisesti järjestelmästä riippuen jollakin seuraavista asetuksista: BEGIN WORK, BEGIN TRANSACTION tai START TRANSACTION tai näiden lyhenteillä. Eksplisiittisesti aloitetun transaktion päätyttyä istunto palaa autocommit-moodiin.

MySQL ei itse ole tietokantajärjestelmä, vaan se tarvitsee erillisen tietokantamoottorin (storage engine). Moottori voidaan valita taulukohtaisesti optiolla ENGINE=<moottori>. Varhainen MyISAM-moottori ei tukenut transaktioita lainkaan, mutta versiosta 5.1 lähtien MySQL:n oletusmoottori on transaktioita tukeva InnoDB. Jatkossa tarkoitamme MySQL-järjestelmällä MySQL/InnoDB-kokonaisuutta.

MySQL/InnoDB istunto voidaan vaihtaa autocommit-moodista transaktionaaliseen moodiin asetuksella

```
SET AUTOCOMMIT = 0;
```

jolloin transaktiot alkavat implisiittisesti. Vastaavasti transaktionaalisesta moodista voidaan palata takaisin autocommit-moodiin asetuksella

```
SET AUTOCOMMIT = 1;
```

Microsoftin SQL Server toimii oletuksena autocommit-moodissa, mutta koko instanssi voidaan konfiguroida toimimaan transaktionaalisessa moodissa, ja yksittäinen SQL-istunto voidaan asettaa toimimaan transaktionaalisesti asetuksella

```
SET IMPLICIT TRANSACTIONS ON
```

ja vastaavasti palauttaa autocommit-moodiin asetuksella

```
SET IMPLICIT TRANSACTIONS OFF
```

Riippumatta käytettävästä DBMS-järjestelmästä, jotkut tietokantaliittymien API:t, kuten esimerkiksi ODBC ja JDBC, toimivat oletuksena autocommit-moodissa ja transaktioprotokollaan ei käytetä SQL:n lauseita. Esimerkiksi JDBC API:ssa transaktiot aloitetaan eksplisiittisesti connection-olion metodikutsulla

```
<connection>.setAutoCommit(false);
```

Myös jotkut utility-ohjelmat, kuten esimerkiksi DB2:n merkkipohjainen SQL-editori CLP toimivat oletuksena autocommit-moodissa.

Tyypillisesti lähes kaikki SQL-komennot voivat esiintyä transaktiossa, myös DDL-komennot. DDL-komentoja ovat taulujen, näkymien, indeksien, jne CREATE, ALTER tai DROP komennot, mutta ei CREATE DATABASE. Täten esimerkiksi transaktiossa luotu taulu häviää, jos transaktio perutaan. Näin ei kuitenkaan käy Oraclessa ja MySQL/InnoDB:ssä. Näissä DDL-komento aiheuttaa implisiittisen COMMIT-operaation, mikä tarkoittaa että mahdollisesti käynnissä olleen transaktion kaikki aikaisemmat komennot commitoituvat myös.

1.4 Transaktiologiikasta

Transaktio voi muodostua yhdestä tai muutamasta peräkkäisestä SQL-komennosta, mutta siihen voi kuulua myös monimutkaisempaa ohjelmalogiikkaa. Transaktion aikaisempien komentojen havaitsema tilanne tietokannan tietosisällössä voi vaikuttaa siihen miten transaktion kulku jatkossa haarautuu eri komentosarjoihin. Jotkut SQL-komennot saattavat myös epäonnistua. Jotkut oppikirjat voivat antaa mielikuvan, että transaktio peruuntuu automaattisesti virhetilanteessa, mutta näin ei asianlaita yleensä ole. Jokaisen SQL-komennon jälkeen komennon onnistuminen tulee diagnosoida ja huomioida transaktiologiikassa. Sovelluksen ohjelmalogiikka on tästä vastuussa ja sen tulee päättää milloin transaktio peruutetaan.

Tarkastellaan seuraava yksinkertaistettua pankkitilien taulua (jonka tunnusten nimiä emme ole lähteneet suomentamaan)

```
CREATE TABLE Accounts (
acctId  INTEGER NOT NULL PRIMARY KEY,
balance DECIMAL(11,2) CHECK (balance >= 0.00)
);
```

Tässä accounts tarkoittaa tilejä, acctId on tilin perusavain (primary key) ja balance on tilin saldo, minkä arvot eivät CHECK-rajoitteen vuoksi voi olla negatiivisia. Tyypillinen oppikirjojen transaktioesimerkki on tilisiirto yhdeltä tililtä toiselle. Seuraavassa esimerkissä 100 euroa tililtä 101 tilille 202:

```
BEGIN TRANSACTION;
UPDATE Accounts SET balance = balance - 100 WHERE acctId = 101;
UPDATE Accounts SET balance = balance + 100 WHERE acctId = 202;
COMMIT;
```

Jos tietokantapalvelin kaatuu tai tietokantayhteys yllättäen katkeaa ensimmäisen UPDATE-komennon jälkeen, transaktio jää commitoimatta ja se peruuntuu automaattisesti. Ilman transaktioprotokollaa tietokanta olisi jäänyt rikkonaiseen tilaan. Transaktioprotokollasta huolimatta esimerkkimme transaktiota ei vielä voi väittää luotettavaksi, sillä:

a) Vaikka jompaa kumpaa tilinumeroa ei ole olemassakaan, vastaava UPDATE-komento on onnistunut SQL-kielen kannalta. Siksi kunkin SQL-komennon jälkeen on SQL-diagnostiikasta selvitettävä käsiteltyjen rivien lukumäärä ja jos selviää ettei jompaa kumpaa tiliä löytynyt, tulee koko transaktio perua.

b) Jos tilin 101 saldo UPDATE-komennon jälkeen tulisi negatiiviseksi rikkoen CHECK-rajoitteen eheysehtoa, virittäisi rajoite virhetilanteen jossa UPDATE-komento peruuntuisi. Jos sovellus ei tässä tapauksessa peruisi transaktiota ROLLBACK-komennolla, jäisi tietokanta rikkonaiseen tilaan.

Harjoittelemme myöhemmin vastaavan esimerkin kanssa ja tulemme huomaamaan, että sovellusohjelmoijien on tunnettava käytössä olevan DBMS-järjestelmän käyttäytyminen. Tärkeää on myös osata SQL-diagnostiikan tutkiminen DBMS-järjestelmästä ja käytettävästä tietokantaliittymän APIsta.

Transaktiologiikan osalta protokollaan kuuluu myös **SAVEPOINT-käsittely**. Transaktion komentojen suoritusvirtaan voidaan määrittää nimettyjä savepoint-pisteitä ja transaktion jossakin myöhemmässä vaiheessa voidaan tehdä transaktion osittainen rollback johonkin jo nimettyyn savepointtiin, mikä tarkoittaa että kaikki päivitykset mitä transaktio on kyseisen savepointin jälkeen tehnyt peruutetaan, mutta kaikki sitä aikaisemmat päivitykset säilyvä edelleen ja transaktion suoritus jatkuu nykyisestä kohdasta (ei siis hypätä savepoint-pisteeseen). Savepoint-käsittely on specifioitu ISO SQL-standardissa ja se on toteutettu keskeisissä DBMS-järjestelmissä, mutta näiden syntakseissa on eroja. Savepoint-käsittely ei mielestämme kuulu transaktio-ohjelmoinnin perusasioihin ja on jätetty pois tästä tutorialista. Tätä on käsitelty tutorialissamme ”SQL Concurrency Technologies”.

1.5 SQL-diagnostiikan tutkiminen

Jonkun SQL-komennon peruuntuminen virheen vuoksi ei yleensä johda koko transaktion automaattiseen peruuntumiseen[3]. Sovellusohjelman on jokaisen SQL-komennon jälkeen tutkittava palvelimelta saatu SQL-diagnostiikka virheiden, varoitusten ja käsiteltyjen rivimäärien osalta ja päätettävä onko transaktio mahdollisesti peruttava ROLLBACK-komennolla.

Virhediagnostiikkaa varten ISO SQL-89 -standardi määritteli integer-tyyppisen indikaattorin **SQLCODE** joka jokaisen SQL-komennon jälkeen kertoo komennon onnistumisesta seuraavasti: 0 ilmaisee komennon onnistuneen, 100 ilmaisee ettei yhtään käsiteltävää riviä löytynyt, ja kaikki muut arvot ovat tuotekohtaisia, mutta rajattu seuraavasti: positiiviset arvot ovat varoituksia ja negatiiviset arvot virheitä.

ISO SQL-92 –standardissa diagnostiikkaa pyrittiin yhtenäistämään. SQLCODE:n jatkosta luovuttiin ja sen tilalle määriteltiin 5-merkkinen **SQLSTATE**, jonka kaksi ensimmäistä merkkiä määrittävät virheiden sekä varoitusten luokat ja kolme viimeistä merkkiä tarkentavat syykoodin. Merkkiparilla ”40” alkavat koodit kertovat automaattisesti peruuntuneesta transaktiosta, missä tarkempana syynä voi olla sarjallistuvuusongelma (tyypillisesti ”deadlock”, johon palaamme myöhemmin) tai mahdollisesti katkenneesta yhteydestä (ajurin kertomasta) tai jostakin palvelimen ongelmasta. Merkkijono ”00000” kertoo SQL:n kannalta täysin onnistuneesta suorituksesta. Kaikkiaan standardi kattaa satoja koodeja, mukaan lukien koodit SQL-constraint määritysten loukkauksille. Lisäksi DBMS-tuotteiden valmistajat voivat määrittää omia tuotekohtaisia koodeja. Riippumatta SQL-standardista DBMS-tuotteet jatkavat myös SQLCODE:n käyttöä.

X/Open Group kehittää DBMS-tuotteita lähempänä olevaa erillistä standardia. Tässä ja useissa DBMS-tuotteissa upotetun SQL:n (Embedded SQL, ESQL) SQL-diagnostiikka palautuu sovellukselle **SQLCA**-struktuurina[4], jonka kentistä on luettavissa mm. SQLCODE, varoitusilmaisimia ja käsiteltyjen rivien määrä. X/Open Group laajensi diagnostiikkaa itse SQL-kieleen **GET DIAGNOSTICS** –lauseella, jonka muodoilla saadaan diagnostiikkatietoa paitsi koko SQL-komennosta, myös sen osien diagnostiikaketjusta. Tämä tullut mukaan myös ISO SQL standardiin alkaen versiosta **SQL:1999**. Toistaiseksi tämä on tuettu vain joissakin DBMS-tuotteissa, kuten esimerkiksi DB2 V9, MySQL 5.6, MariaDB 10 ja PostgreSQL 9, mutta vain osia standardin määrittämistä nimetyistä diagnostiikka-itemeistä, ehkä täydellisimmin Mimer:issä. Seuraava esimerkki näyttää kuinka diagnostiikkaa luetaan MySQL 5.6:n ja MariaDB:n SQL-kielessä käyttäen kielen @-alkuisia muuttujia:

```
GET DIAGNOSTICS @rowcount = ROW_COUNT;
GET DIAGNOSTICS CONDITION 1 @sqlstate = RETURNED_SQLSTATE,
                            @sqlcode  = MYSQL_ERRNO ;
SELECT @sqlstate, @sqlcode, @rowcount;
```

GET DIAGNOSTICS –lausetta ei ainakaan toistaiseksi ole toteutettu lainkaan Oraclessa eikä Microsoftin SQL Serverin Transact-SQL (T-SQL) –kielessä, joissa molemmissa on omat proseduraaliset laajennukset.

Oracle:n SQL-kieltä on laajennettu erillisellä lohkorakenteisella PL/SQL-kielellä, jossa SQLCODE on kieleen sisältyvä virheindikaatori ja käsitellyn rivimäärän indikaattori on SQL%ROWCOUNT.

---

[3] PostgreSQL jättää virheen jälkeen kaikki transaction komennot palvelematta ja vaihtaa mahdollisen COMMIT-komennon automaattisesti ROLLBACK-komenoksi.

[4] SQLCA:n tarkempi esittely ei kuulu tämän tutorialin aiheisiin. Upotettu SQL oli ensimmäisiä SQL-tietokantaliittymiä ja on edelleen käytettävissä, mutta se on jätetty jo pois ISO SQL –sandardista.

Transact-SQL (josta käytetään myös lyhennettä T-SQL) itse on proseduraalinen SQL-toteutus, jossa diagnostiikkaa varten käytettävissä @@-alkuiset virheindikaattori @@ERROR ja käsiteltyjen rivien määräindikaattori @@ROWCOUNT.

DB2:n proseduraalinen laajennus SQL PL on käytettävissä vain talletetuissa proseduureissa ja siinä sekä SQLCODE että SQLSTATE ovat käytettävissä kieleen sisältyvinä indikaatttoreina seuraavan esimerkin tapaan

```
<SQL statement>
IF (SQLSTATE <> '00000') THEN
     <error handling>
END IF;
```

Virhetilanteiden selvittely ohjelmointikielissä tehdään poikkeuskäsittelylohkoissa (exception handling), joista ensimmäisiä oli ADA-kielen lohkojen EXCEPTION-osa. Oraclen PL/SQL noudattaa pitkälti ADA-kieltä ja PL/SQL-lohkojen EXCEPTION-osa on tätä perua.

Tätä perua on myös Java- ja C# -kielten try-catch –rakenne. Javan JDBC API:ssa virheilmaisimet on paketoitu SQLException-olioiksi, joiden käsittely tapahtuu seuraavan rakenteen mukaan

```
... throws SQLexception {
...
try {
     ...
     <JDBC statement(s)>
}
catch (SQLException ex) {
     <exception handling>
}
```

Catch-haaran poikkeuskäsittelyosassa tutkitaan virhettä tarkemmin ex-olion metodeilla, joilla saadaan selville käytetyn DBMS-järjestelmän tukemat SQLCODE ja SQLSTATE. Näistä meillä on esimerkki liitteen 2 Java-ohjelmassa. Samoin käsitellyn rivimäärän ohjelmallisesta lukemisesta SQLStatement-olioiden execute-metodien paluukoodina.

1.6 Yksittäisten transaktioiden “Hands-on” -kokeilut

> Ei kannata uskoa kaikkea mitä lukee tai kuulee tietokantajärjestelmistä! Luotettavasti toimivien sovellusten rakentamista varten sinun on kokeiltava ja todennettava se miten DBMS-järjestelmäsi käyttäytyy palvelujen toteutuksessa. Tietokantajärjestelmät käyttäytyvät eri tavoin jopa transaktiopalveluiden perusasioissa. Oppikirjojen kirjoittajat eivät välttämättä itse ole testanneet kaikkea, vaan kertovat mitä ovat muista kirjoista lukeneet.

Liite 1 esittää kokeiluja eksplisiittisillä ja implisiittisillä transaktioilla, COMMIT ja ROLLBACK komennoilla sekä transaktiologiikasta käyttäen MS SQL Server –järjestelmää. Vaikka ne ovat listauksia oikeasti suoritetuista testeistä, sinun on parempi testata niitä itse SQL Serverillä todentaaksesi SQL Serverin todellisen käyttäytymisen. SQL Server toimii vain Windows-alustalla, joten sitä ei ole voitu asentaa virtuaalilaboratorioomme, mutta kuka tahansa saa ladata sen ilmaisversion SQL Server Express Microsoftin websivustosta.

Hands-on –harjoittelua varten tarjoamme opiskelukäyttöön tarkoitun ilmaisen virtuaali-laboratoriomme DBTechLab (josta käytämme myös nimeä DebianDB). Se perustuu virtuaalikoneeseen, jonka käyttöjärjestelmänä on Debian Linux ja johon on valmiiksi asennettu SQL serveriä lukuun ottamatta yleisimmät DBMS-järjestelmä: IBM DB2 Express-C, Oracle XE, MySQL GA ja PostgreSQL sekä edistyksellinen Pyrrho DBMS.

Jatkossa esittämämme harjoitukset on tarkoitus tehdä DBTechLabin MySQL GA palvelimella, perustuen siihen, että MySQL on tiettävästi kouluissa eniten käytetty DBMS. MySQL:n alkuperäisten kehittäjien MariaDB näyttäisi käyttäjälle samanlaiselta. Harjoitusten skriptit löytyvät DBTechLabin student-käyttäjän hakemiston ”Transactions” tiedostoista Appendix1_<dbms>.txt, missä <dbms>-osa kertoo mille järjestelmälle ne on sovitettu. Jotkut harjoitukset näyttävät yllättäviä tuloksia MySQL:n osalta, mikä ei ole ollut tavoitteena. Harjoitukset on sovitettu samalta pohjalta kaikille järjestelmille. Ongelmatapauksissa pyrimme myös esittämään ongelmaan kiertoratkaisun.

MySQL:n käyttäytyminen riippuu siitä mikä tietokantamoottori on käytössä. Käytettävä moottori määritetään taulukohtaisesti **ENGINE**-määrellä, esimerkiksi

```
CREATE TABLE (…  ) ENGINE=InnoDB ;
```

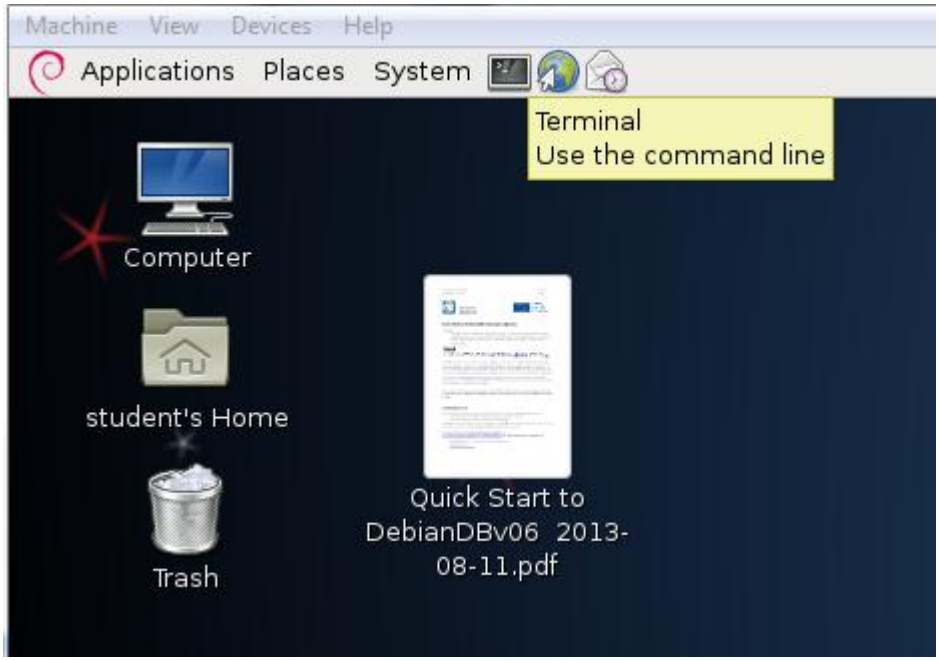
Varhaisemmat MySQL-versiot eivät tukeneet transaktioita lainkaan, mutta alkaen versiosta 5.1 oletusmoottorina on transaktioita tukeva InnoDB moottori.

Harjoitusten aluksi kannattaa tutustua ohjeeseen "Quick Start to the DebianDB Database Laboratory" (Quick Start Guide). Tässä opastetaan DBTechLabin asennus ja kuinka tietoja voidaan siirtää virtuaalikoneen ja isäntäkoneen välillä, mitä käyttäjätunnuksia laboratorioon on asennettu, miten virtuaalikonetta ylläpidetään ja suljetaan, miten sen tietokantapalvelimet otetaan käyttöön, jne..

Kun DebianDB käynnistetään, oletuksena sen istunto alkaa student-käyttäjän nimissä. Student-käyttäjän salasana on “**password**”. Tätä salasanaa tarvitaan toisinaan, esimerkiksi kun DebianDB on ollut käyttämättömänä jonkin aikaa ja sen ruutu pimenee.

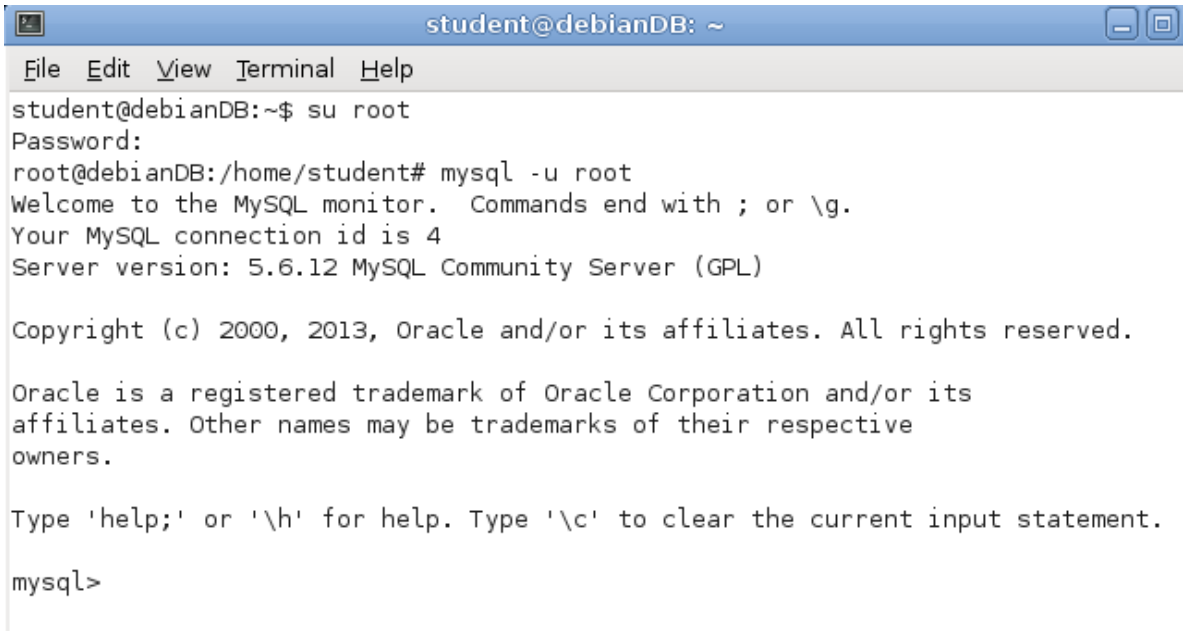
Tässä tutorialissa käytämme MySQL-järjestelmää Linuxin pääte-emulaattorilla ”Terminal”, joka käynnistetään kuvan 1.4 mukaisesti Linux Gnomen valikkorivin ikonista "Terminal/Use the

command line". Ensimmäisen MySQL-tietokannan luontia varten tulee istunto vaihtaa root-käyttäjän nimiin Linux-komennolla ”su root”. Rootin salasana DebianDB:ssä on “**P4ssw0rd”**.



Kuva 1.4 Pääteistunnon käynnistys ikonista "Terminal / Use the command line"

Seuraavaksi käynnistetään kuvan 1.5 mukaisesti MySQL:n client-ohjelmalla MySQL-istunto:



Kuva 1.5 MySQL-istunnon aloitus root-käyttäjänä

Luodaan aluksi uusi tietokanta ”testdb” seuraavalla komennolla:

```
CREATE DATABASE testdb;
```

Seuraavalla komennolla root-käyttäjä myöntää student-käyttäjälle kaikki oikeudet testdb-kannan

käyttöön:

```
GRANT ALL ON testdb.* TO 'student'@'localhost';
```

Tämän jälkeen root-käyttäjän mysql-istunto voidaankin lopettaa EXIT-komennnolla ja palata Linuxin exit-komennolla jatkamaan student-käyttäjän istuntoa seuraavasti:

```
EXIT;
exit
```

Nyt student-käyttäjä voi aloittaa testdb-kannan käytön mysql-clientilla:

```
mysql
use testdb;
```

Kannattaa muistaa, että Linux- ja Unix-alustoilla MySQL:ssä tietokannan nimi on **case-sensitiivinen** eli isot ja pienet kirjaimet tulkitaan tässä eri merkeiksi. Kannan nimen voi antaa myös suoraan clientin komentorivillä seuraavasti:

mysql –D testdb

HARJOITUS 1.1

Aloitamme luomalla ensimmäisen taulun, jonka nimenä voisi olla vaikka “T” ja määritämme siihen kolme saraketta:

id tyyppinä integer ja tehdään tästä perusavain (primary key)
s tyyppinä vaihtuvanmittainen merkkijono, kuitenkin korkeintaan 40-merkkiä pitkä
si tyyppinä smallint arvoalueena -32768 . . +32767

missä sarakkeille s ja si sallitaan myös puuttuvat eli NULL-arvot, seuraavasti:

```
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(40), si SMALLINT)
ENGINE=InnoDB;
```

Kannattaa muistaa, että Linux- ja Unix-alustoilla MySQL:ssä myös taulujen nimet ovat case-sensitiivisiä eli isot ja pienet kirjaimet tulkitaan eri merkeiksi.
MySQL-clientin komennot tulee aina päätttää puolipisteeseen (paitsi EXIT-komento).
Tässä ENGINE-määreen voisi jättää myös pois, koska InnoDB on oletusmoottori.

```
mysql> CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(40), si SMALLINT)
    -> ENGINE=InnoDB;
Query OK, 0 rows affected (0.27 sec)

mysql>
```

Jokaisen komennon jälkeen mysql näyttää automaattisesti keskeiset diagnostiikkatiedot komennon suorituksesta.

MySQL-clientilla on käytettävissä myös joitakin omia komentoja, jotka eivät kuulu itse SQL-kieleen. DESCRIBE-komennolla voidaan tarkistaa jo olemassa olevan taulun rakenne, esimerkiksi seuraavasti:

```
DESCRIBE T;
```

Jatketaan harjoitusta ja lisätään tauluumme seuraavat rivit:

```
-- tämä rivi on vain kommenttirivi ---------
INSERT INTO T (id, s) VALUES (1, 'first');
INSERT INTO T (id, s) VALUES (2, 'second');
INSERT INTO T (id, s) VALUES (3, 'third');
SELECT * FROM T ;
----- tämäkin rivi voisi olla kommenttirivi muissa järjestelmissä ----
```

Jos kopioimme suoritukseen myös alussa ja lopussa olevat kommenttirivit, niin voimme havaita seuraavaa: Jotta MySQL tulkitsisi peräkkäisten tavuviivojen aloittavan loppurivin kattavan kommentin, on tavuviivoja oltava vain kaksi ja niitä on seurattava ainakin yksi blanko.

Muistellaanpa mitä aikaisemmin kerroimme SQL-transaktioista ja kokeillaan peruuntuvatko edellä lisäämämme rivit ROLLBACK-komennolla:

```
ROLLBACK;
SELECT * FROM T ;
```

Ainakaan mysql ei anna tässä virheilmoitusta ROLLBACK-komennosta, mutta "SELECT * FROM T" komento näyttää että rivit ovat edelleen tallessa taulussa T.

- Mikä mahtaa olla selitys?
No, selityshän on, että mysql-istunto alkaa oletuksena AUTOCOMMIT-moodissa. Jos tässä moodissa halutaan tehdä transaktio, on ensin annettava komento ”START TRANSACTION”. Kokeillaanpa seuraavaa:

```
START TRANSACTION;
INSERT INTO T (id, s) VALUES (4, 'fourth');
SELECT * FROM T ;
ROLLBACK;
SELECT * FROM T;
```

Havaitaan, että transaktion päätyttyä mysql-istunto on edelleen autocommit-moodissa.

Kysymys

- Vertaa yllä annettujen kahden “SELECT * FROM T” komennon tuloksia. Miten selität tulosten eron?

HARJOITUS 1.2

Jatketaan kokeilua seuraavasti:

```
START TRANSACTION;
INSERT INTO T (id, s) VALUES (5, 'fifth');
SELECT * FROM T ;
SELECT @@autocommit;
ROLLBACK;
SELECT @@autocommit;
INSERT INTO T (id, s) VALUES (6, 'sixth');
SELECT * FROM T;
ROLLBACK;
SELECT * FROM T;
```

Kysymyksiä

- Mikä on viimeksi annetun “SELECT * FROM T” komennon tulosjoukko?
- Miksi @@autocommit-rekisterin arvo on edellä sama transaktion aikana ja sen jälkeen?

HARJOITUS 1.3

Käännetään nyt autocommit-moodi pois päältä "SET AUTOCOMMIT" komennolla:

```
SET AUTOCOMMIT = 0;
```

ja poistetaan kaikki muut lisätyt rivit paitsi ei ensimmäistä

```
DELETE FROM T WHERE id > 1;
COMMIT;
```

lisäten sitten uusia rivejä

```
START TRANSACTION;
INSERT INTO T (id, s) VALUES (7, 'seventh');
COMMIT;
INSERT INTO T (id, s) VALUES (8, 'eigth');
SELECT * FROM T;
```

… ja peruuttaen transaktion:

```
ROLLBACK;
SELECT * FROM T;
```

Kysymys

- Mitä eroa on "START TRANSACTION" ja "SET AUTOCOMMIT" -komennoilla ?

Huomio:
Linuxin terminal-ikkunassa MySQL client tukee näppäimistön nuolia siten että niiden avulla voidaan palata aikaisempiin komentoihin ja korjailla niiden muotoa poistaen merkkejä DEL-nppäimellä tai kirjoittaen uusia merkkijonoja. Tämä ei ole mahdollista kaikissa muissa DBMS-järjestelmissä.

HARJOITUS 1.4

```
-- paluu kannan aikaisempaan tilaan, jos halutaan toistaa Harjoitus 1.4
SET AUTOCOMMIT=0;
DELETE FROM T WHERE id > 1;
DROP TABLE T2;
COMMIT;
```

SQL-komennot, joilla luodaan, muutetaan tai poistetaan tietokannan rakenteita: tauluja, näkymiä, indeksejä jne, eli esimerkiksi CREATE TABLE, ALTER TABLE, CREATE VIEW, CREATE INDEX ja DROP TABLE, kuuluvat SQL-kielen osaan “Data Definition Language” eli DDL, kun taas komennot INSERT, SELECT, UPDATE ja DELETE, joilla käsitellään talletettuja tietoja kuuluvat SQL-kielen osaan ”Data Manipulation Language” eli DML. DML-kielen komennot voivat kuulua normaalisti transaktioon, mutta DDL-komentojen käsittely transaktioissa vaihtelee järjestelmien kesken. Kokeillaan miten nämä hoituvat MySQL:ssä:

```
SET AUTOCOMMIT=0;
INSERT INTO T (id, s) VALUES (9, 'will this be commited?');
CREATE TABLE T2 (id INT);
INSERT INTO T2 (id) VALUES (1);
SELECT * FROM T2;
ROLLBACK;

SELECT * FROM T;  -- Mitä tapahtui taululle T ?
SELECT * FROM T2; -- Mitä tapahtui taululle T2 ?
-- Vertaa seuraavana kyselyä olemattomasta taulusta:
SELECT * FROM T3; -- siis olettaen, ettei meillä ole taulua T3
SHOW TABLES;      -- tämä on MySQL:n oma komento
DROP TABLE T2;
COMMIT;
```

Kysymys

- Mitä saimme selville DDL-komentojen vaikutuksesta transaktiokäsittelyssä?

HARJOITUS 1.5

Palautetaan taulun T sisältö alkutilaan:

```
SET AUTOCOMMIT=0;
DELETE FROM T WHERE id > 1;
COMMIT;
SELECT * FROM T;
COMMIT;
```

Testaamme seuraavalla komentosarjalla laukaiseeko SQL-virhe MySQL:ssä transaktiolle automaattisen ROLLBACKin:

```
SET AUTOCOMMIT=0;
INSERT INTO T (id, s) VALUES (2, 'Error test starts here');
-- Testataan onko nollalla jako MySQL:ssä virhe:
SELECT (1/0) AS dummy FROM DUAL;
-- Kuinka käy, kun päivitämme riviä, jota ei ole?
UPDATE T SET s = 'foo' WHERE id = 9999 ;
-- Kuinka käy, kun poistamme rivin, jota ei ole?
DELETE FROM T WHERE id = 7777 ;
-- Mitä tapahtuu tuplariville?
INSERT INTO T (id, s) VALUES (2, 'Hi, I am a duplicate');
-- Mitä tapahtuu liian pitkälle merkkijonoarvolle
INSERT INTO T (id, s)
       VALUES (3, 'How about inserting too long of a string value?');
-- Mitä tapahtuu SMALLINT:in ylivuodolle?
INSERT INTO T (id, s, si) VALUES (4, 'Smallint overflow for 32769?', 32769);
-- Näitä pitäisi kokeilla jokaisen komennon jälkeen
-- sille kertovat tilanteen vain viimeksi suoritetun komennon osalta!
SHOW ERRORS;
SHOW WARNINGS;
--
INSERT INTO T (id, s) VALUES (5, 'Is the transaction still active?');
SELECT * FROM T;
COMMIT;
DELETE FROM T WHERE id > 1;
SELECT * FROM T;
COMMIT;
```

Kysymyksiä

a) Laukaisiko mikään virhe MySQL:ssä automaattisen rollbackin?
b) Onko nollalla jako virhe MySQL:ssä?
c) Reagoigo MySQL arvojen ylivuotoihin?
d) Mitä opimme seuraavista tuloksista?

```
mysql> UPDATE T SET s = 'foo' WHERE id = 9999 ;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0

mysql> INSERT INTO T (id, s) VALUES (2, 'Hi, I am a duplicate');
ERROR 1062 (23000): Duplicate entry '2' for key 'PRIMARY'
```

Tässä mysql-clientin tulostamassa diagnostiikassa epäonnistuneen INSERT-komennon virheilmoituksen koodi "23000" on SQL-standardin mukainen SQLSTATE:n koodi perusavainrikkeestä eli yhteentörmäyksestä jo varattuun perusavaimen arvoon, kuten MySQL:n virheilmoitustekstikin kertoo. Luku 1062 on MySQL:n oma SQLCODE:n koodi tälle rikkeelle. Sovellusohjelmaan diagnostiikka saataisiin MySQL 5.6:n SQL:n keinoin GET DIAGNOSTICS komennoilla seuraavasti (tosin tässä mysql-clientilla katsottuna):

```
mysql> GET DIAGNOSTICS @rowcount = ROW_COUNT;
Query OK, 0 rows affected (0.00 sec)

mysql> GET DIAGNOSTICS CONDITION 1 @sqlstate = RETURNED_SQLSTATE,
    ->                             @sqlcode = MYSQL_ERRNO ;
Query OK, 0 rows affected (0.00 sec)
```

Merkillä “@” alkavat tunnukset ovat MySQL:n tyypittömiä paikallisia muuttujia. Sovellusohjelmien API-liittymissä näiden paikalla voidaan käyttää sovellusohjelman omia muuttujia, mutta koska näissä harjoituksissa käytämme vain SQL-kieltä, simuloimme niiden käyttöä näillä MySQL:n muuttujilla. Käytämme tätä tekniikkaa myös myöhemmissä harjoituksissa. Seuraava esimerkki näyttää kuinka voimme käyttää näiden muuttujien arvoja:

```
mysql> SELECT @sqlstate, @sqlcode, @rowcount;
+-----------+----------+-----------+
| @sqlstate | @sqlcode | @rowcount |
+-----------+----------+-----------+
| 23000     |     1062 |        -1 |
+-----------+----------+-----------+
1 row in set (0.00 sec)
```

HARJOITUS 1.6

```
DROP TABLE Accounts;
SET AUTOCOMMIT=0;
```

Tunnettu MySQL:n puute on etteivät **saraketason CHECK**-rajoitteet ole tuettuja, eli seuraava muille järjestelmille tässä harjoituksessa käyttämämme taulun luontikomento ei onnistu MySQL:ssä

```
CREATE TABLE Accounts (
acctID  INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL CONSTRAINT unloanable_account CHECK (balance >= 0)
);
```

MySQL:n SQL-jäsentäjä (parser) hyväksyy kyllä seuraavan komentomuodon **rivitason CHECK**-rajoitteen

```
CREATE TABLE Accounts (
acctID  INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL ,
CONSTRAINT unloanable_account CHECK (balance >= 0)
) ENGINE = InnoDB;
```

mutta vaikka syntaksi kelpaa, niin CHECK-rajoitteella ei ole vaikutusta ainakaan vielä versiossa 5.6, mikä voidaan todeta seuraavalla kokeilulla, missä INSERT-komennon kuuluisi kaatua sillä sen ei pitäisi läpäistä CHECK-rajoitettamme:

```
INSERT INTO Accounts (acctID, balance) VALUES (100,-1000);
SELECT * FROM Accounts;
ROLLBACK;
```

Huom:

> Emme ole jättäneet tätä CHECK-rajoitetestiä kokeiluista pois, koska se kuuluu testisarjaamme ja toisaalta sovelluskehittäjien on tunnettava tämä ongelma. Kaikissa tietokantohjelmistoissa on virheitä. Sovelluskehittäjän ammattitaitoa on tuntea nämä ongelmat ja niiden kiertoratkaisut (workaround). **SQL-triggerit** eivät varsinaisesti kuulu kirjamme aiheisiin, mutta asiasta kiinnostuneen lukijan kannattaa tutustua tämän MySQL:n CHECK-ongelman kiertoratkaisuun triggerillä, joka löytyy virtuaalilaboratoriomme hakemiston ”Transactions” tiedostosta `AdvTopics_MySQL.txt`.

Käytämme luotua taulua jatkotesteissämme ja lataamme siihen nyt testiaineistomme ja teemme muutaman tilisiirtokokeilun peruuttaen kunkin kokeilutransaktion siten että kokeilut ovat toisistaan riippumattomia:

```
-- Let’s load also contents for our test:
SET AUTOCOMMIT=0;
INSERT INTO Accounts (acctID, balance) VALUES (101, 1000);
INSERT INTO Accounts (acctID, balance) VALUES (202, 2000);
SELECT * FROM Accounts;
COMMIT;

-- A. Let's try the bank transfer
UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;
SELECT * FROM Accounts;
ROLLBACK;

-- B. Let's test that the CHECK constraint actually works:
UPDATE Accounts SET balance = balance - 2000 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 2000 WHERE acctID = 202;
SELECT * FROM Accounts ;
ROLLBACK;
```

Seuraava tilisiirto yrittää siirtää 500 euroa tililtä 101 sellaiselle tilille, jota ei ole aineistossamme:

```
-- C. Updating a non-existent bank account 777:
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 777;
SELECT * FROM Accounts ;
ROLLBACK;
-----------------------------------
```

Kysymyksiä

a) Mitä voidaan sanoa tilisiirtotransaktiosta, jossa toinen UPDATE-komennoista ei sovelluksen kannalta onnistu?

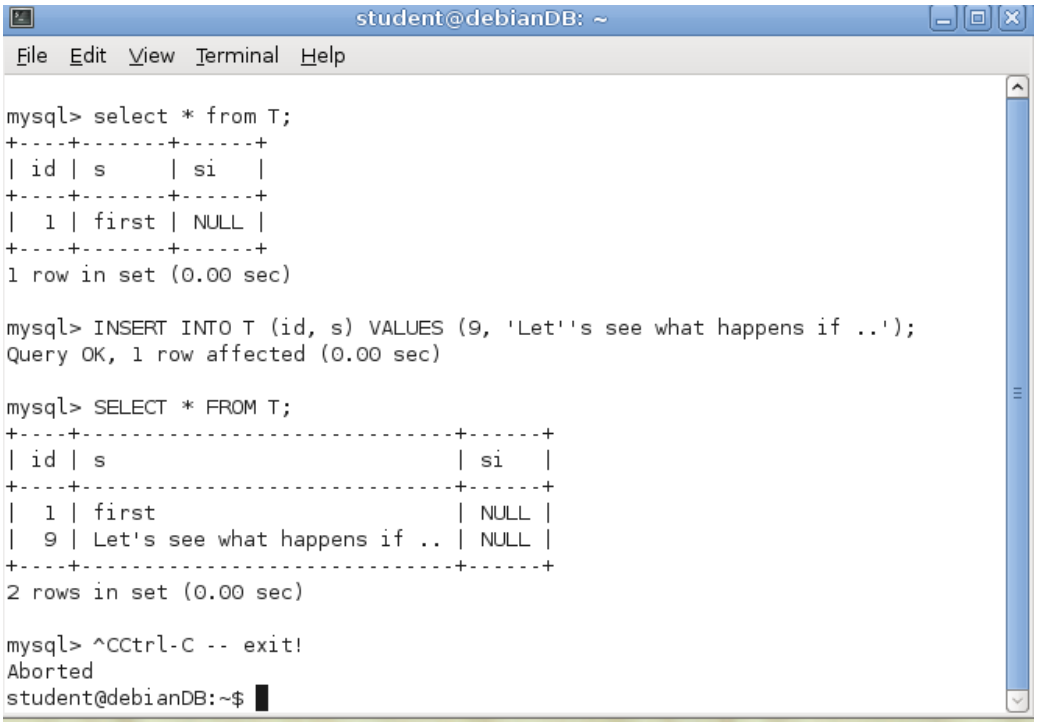
b) Jos transaktioiden B tai C ROLLBACK-komennot olisi korvattu COMMIT-komennolla, niin olisiko tietokannan tietosisältö kunnossa kokeilun jälkeen?
c) Mitä MySQL:n diagnostiikkaindikaattoreita sovellus voisi hyödyntää edellä kuvatuissa transaktioissa tietovirheiden estämiseksi?

HARJOITUS 1.7 – SQL-transaktio toipumisyksikkönä (Unit of Recovery)

Kokeilemme nyt tietokannan toipumista ongelmasta, jossa transaktio on jäänyt commitoimatta. Tämähän ei välttämättä ole kovin harvinaista Internet-sovelluksissa. Koetta varten suoritamme seuraavassa pienen transaktion, jossa lisäämme kantaan INSERT-komennolla yhden rivin ja SELECT-komennolla toteamme rivin löytyvän kannasta, mutta COMMIT-komennon asemesta katkaisemme SQL-istunnon kaatamalla client-ohjelman. Tämä simuloi tapausta, jossa yhteys palvelimeen menetetään tai sovellus taikka itse tietokantapalvelin kaatuu. Tämän jälkeen aloitamme uuden SQL-istunnon ja tarkastamme löytyykö rivi vielä kannasta:

```
SET AUTOCOMMIT = 0;
INSERT INTO T (id, s) VALUES (9, 'Let''s see what happens if ..');
SELECT * FROM T;
```

Seuraavaksi katkaisemme mysql-ohjelman "Control C" (Ctrl-C) painalluksella (kuvassa 1.6):



Kuva 1.6 Simuloidaan tietokantayhteyden katkeamista, jolloin transaktio jää kesken

Jatketaan koetta avaamalla uusi mysql client-istunto tietokantaan:

```
mysql
USE testdb;
SET AUTOCOMMIT = 0;
SELECT * FROM T;
COMMIT;
EXIT;
```

Kysymys

- Mitä voimme päätellä keskenjäävän transaktion vaikutuksesta tietokannan sisältöön?

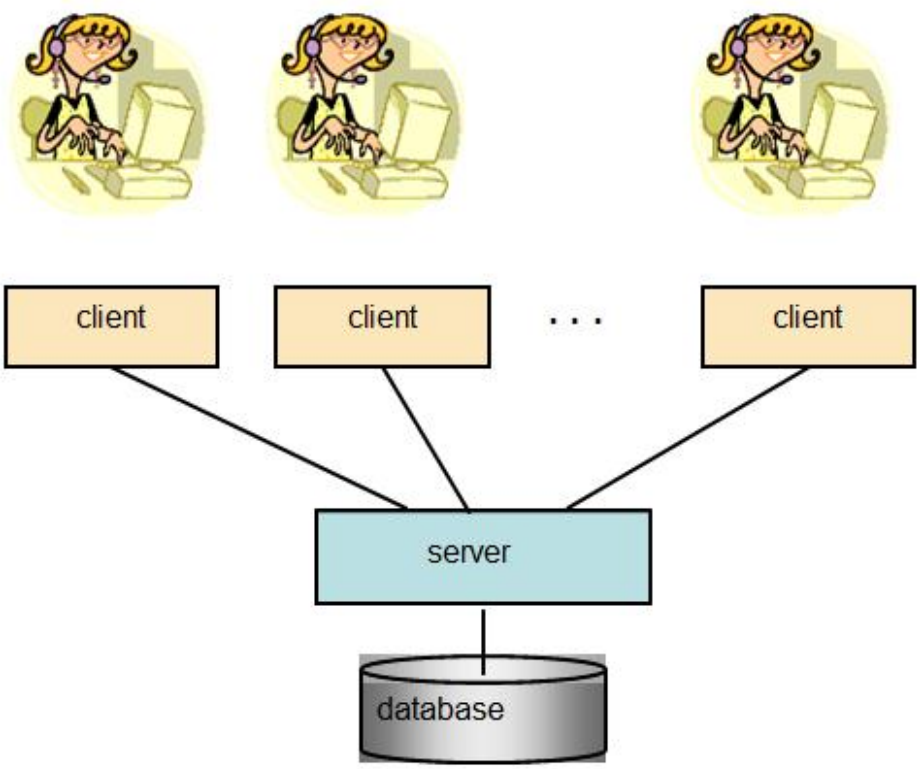
Huom:

Kaikki transaktioiden tekemät päivitykset tietokannan sisältöön kirjataan tietokannan transaktiolokiin. Liite 3 selittää miten koko tietokanta toipuu transaktiolokin avulla viimeisen commitoidun transaktion tasolle, jos palvelin sattuisi kaatumaan. Harjoitustamme voisi laajentaa myös tämän kokeiluun, jos clientin asemesta kaadamme MySQL:n palvelinprosessin **mysqld** seuraavasti:

```
student@debianDB:~$ su root
Password:
root@debianDB:/home/student# ps -e | grep mysqld
 1092 ?        00:00:00 mysqld_debianDB
 1095 ?        00:00:00 mysqld_safe
 1465 ?        00:00:01 mysqld
root@debianDB:/home/student# kill 1465
```

2 Samanaikaiset transaktiot

Sovellusohjelma, joka on testattu oikein toimivaksi yhden käyttäjän ympäristössä, voi satunnaisesti toimia väärin kun samaa tietokantaa käyttää samanaikaisesti useampi käyttäjä joko samalla tai eri ohjelmilla. Kuva 2.1. esittää tilannetta, jossa useampi käyttäjä käyttää samaa tietokantaa samanaikaisesti.



Kuva 2.1 Useampi käyttäjä käyttää samaa tietokantaa samanaikaisesti.

Varoituksen sana!

Älä usko kaikkea mitä luet tai kuulet DBMS-tuotteiden transaktiopalveluista! Jotta osaisit rakentaa luotettavia, sinun tulee kokeilla käyttämäsi DBMS-järjestelmän käyttäytymistä myös samanaikaisuuden hallintapalveluiden osalta. Järjestelmien transaktiopalveluissa on merkitseviä eroja. Vaikka samanaikaisuusongelmat tuotannossa ovat satunnaisia ja jotkut jopa harvinaisia, niitä voidaan sopivilla koejärjestelyillä testata jo kehitysvaiheessa transaktioiden yksikkötestein.

2.1 Samanaikaisuusongelmia

Jos tietokantapalvelimessa ei olisi kunnon samanaikaisuuspalveluita tai sovelluskehittäjä ei tunne kunnolla palvelimen palveluiden oikeaoppista käyttöä, tietokannan tietosisältö tai tietokannasta luetut tiedot voisivat korruptoitua eli tietokanta tai sovellus olisi **epäluotettava**.

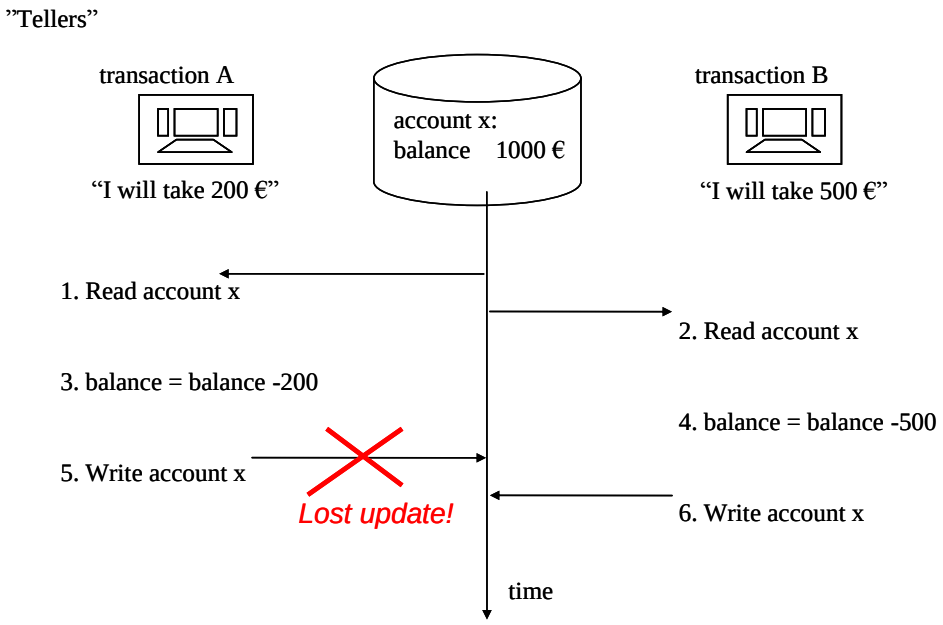
Tarkastelemme seuraavaksi tyypillisiä samanaikaisuusongelmia:

- Hukatun päivityksen ongelma (lost update problem)
- Likaisen lukemisen ongelma eli commitoimattoman tiedon lukemisen ongelma (dirty read problem)
- Ei-toistettavissa olevan lukemisen ongelma (non-repeatable read), missä samoja rivejä ei enää saada luettua saman transaktion aikana (ongelmaan eivät kuulu ne luetut rivit, joita transaktio itse on päivittänyt)
- Havaitsemattomien rivien ongelma (The phantom read problem), missä transaktion aikana tietokantaan putkahtaa uusia rivejä, jotka transaktion SQL-lauseiden hakuehtojen kannalta olisivat voineet tulla käsitellyiksi transaktiossa, mutta jäävät transaktiolta huomaamatta.

Ongelmien tarkemman kuvauksen jälkeen katsomme miten nämä ratkaistaisiin ISO SQL standardin mukaan ja miten ne ratkaistaan oikeilla DBMS-järjestelmillä.

2.1.1 Hukatun päivityksen ongelma (The Lost Update Problem)

Hukatun päivityksen ongelma voisi syntyä, jos useampi client ensin lukee (esimerkiksi tiedostosta) saman tietueen, muokkaa sitä ja päivittää takaisin samalle paikalle ilman samanaikaisuuden hallintapalvelua. Kuva 2.2 havainnollistaa ongelmaa pankkisovelluksella missä esimerkiksi kaksi saman perheen jäsentä sattuu eri pankkiautomaateilla käyttämään samaan aikaan samaa tiliä, jolla alun perin on 1000 euroa, toisen (nostotransaktio A) nostaessa tililtä 200 euroa ja toisen (nostotransaktio B) nostaessa 500 euroa.



Kuva 2.2 Hukatun päivityksen ongelma

Oletetaan, että A:n automaatti kirjoittaa vaiheessa 5 uudeksi saldoksi 800 euroa ja hetkeä myöhemmin vaiheessa 6 B:n automaatti kirjoittaa uudeksi saldoksi 500 euroa tutkimatta tilin senhetkistä saldoa. Tällöin transaktion A kirjoittama tieto tulee hukatuksi. Minkään nykyaikaisen relaatiotietokannan transaktioiden tapauksessa siten että B kirjoittaisi A:n tekemä päivityksen päälle ennen kuin A:n transaktio päättyy, sillä relaatiotietokantajärjestelmissä kirjoitus suojataan aina ns. **kirjoituslukolla, joka vapautetaan vasta transaktion päättyessä.**

Tässä mielessä hukattu päivitys (lost update problem) ei ole mahdollinen transaktion aikana, mutta transaktion A päätyttyä tilanne muuttuu:

Transaktio B jäisi odottamaan lukon vapautumista ja kun lukko vapautuu, se saisi oman kirjoituslukon ja voisi päällekirjoittaa tilille oman saldonsa. Lopputulos olisi siis kuitenkin hukatun päivityksen kaltainen, mitä oppikirjoissa ei mainosteta. Jotkut pitävät myös tätä hukatun päivityksen ongelmana. Meille tämä on esimerkki **sokean päällekirjoituksen** ongelmasta (**blind overwriting** problem). Seuraavassa tarkastelemme kahta tapaa jolla tämä voidaan välttää:

a) Sensitiivisen päivityksen ohjelmointitekniikalla
b) Lukitusjärjestelyllä, jossa vain yksi kilpaileva transaktio kerrallaan pääsee suoritukseen

a) Blind overwriting ongelmalta vältytään **sensitiivisen päivityksen** ohjelmointekniikalla, mikä tarkoittaa tässä sitä, että tilin päivitys tehdään UPDATE-komennolla, joka laskee uuden arvon vanhan arvon perusteella seuraavan esimerkin tapaan:

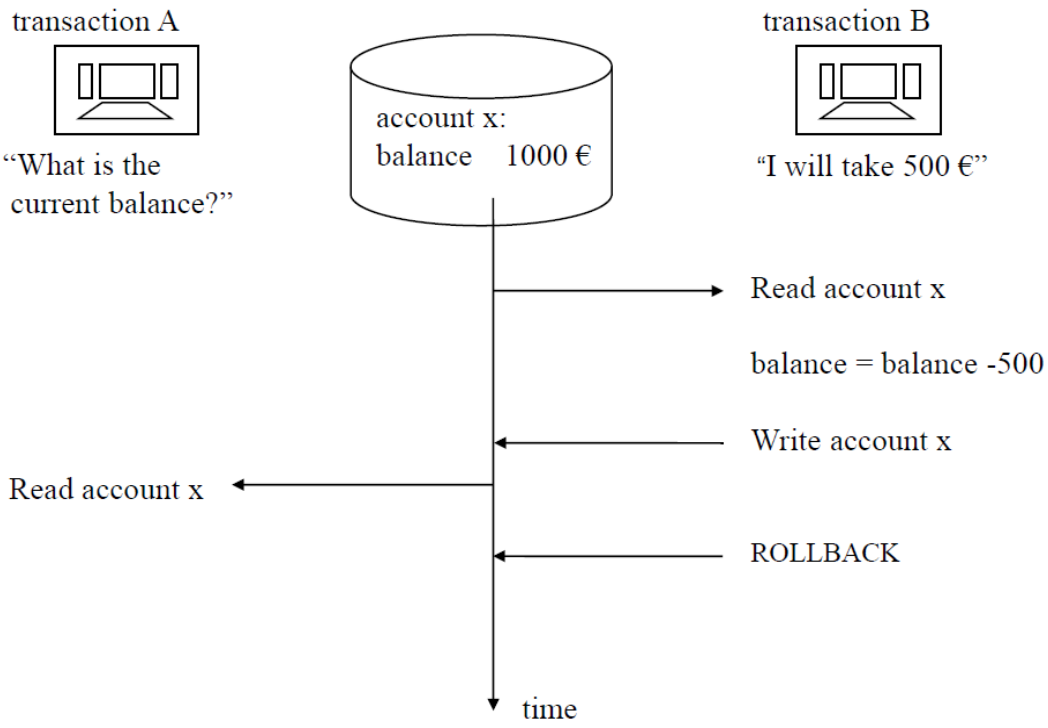
```
UPDATE Accounts SET balance = balance – 200
WHERE acctID = 100;
```

b) Kilpailevien transaktioiden pakotus odottamaan lukituksilla onnistuu useallakin tavalla, mutta riippuen DBMS-järjestelmästä:
   - Jotkut järjestelmät mahdollistavat explisiittisen koko **taulun lukinnan** LOCK TABLE –komennolla. Lukitus tulee tehdä heti transaktion alussa. Kaikki transaktion lukot vapautuvat tyypillisesti aina transaktion lopussa, jolloin odottavat transaktiot pääsevät jatkamaan. Koko taulun lukitus haittaa myös muita transaktioita, jotka käyttäisivät ehkä vain joitakin taulun muita rivejä.
   - Jotkut järjestelmät mahdollistavat tarkemman explisiittisen **rivitason lukinnan** aloittamalla transaktion seuraavaa muotoa olevalla lukuoperaatiolla
     “SELECT ..FROM .. WHERE .. FOR UPDATE”
   - Lukulukkoja tukevan järjestelmän (tarkemmin: Multi-Granular Locking, MGL) tapauksessa kuvan 2.2 transaktiot eivät edellytä ylimääräisiä operaatioita, sillä lukemista varten järjestelmä yrittää automaattisesti hankkia transaktiolle ”shared” eli **S-lukon** ao. kohteeseen. S-lukkoja voi samaan kohteeseen olla usealla transaktiolla. Kirjoittamista varten tarvitan “exclusive” eli **X-lukko**. Tämän voi saada samaan kohteeseen vain yksi kerrallaan ja vain jos kenelläkään ei ole muitakaan lukkoja kohteeseen. Kuva 2.2 tapauksessa molempien transaktioiden kirjoitusoperaatiot jäävät odottamaan toisiaan eli syntyy ns. **Deadlock**-tilanne, minkä DBMS:n deadlock detector –säie purkaa tekemällä automaattisen ROLLBACK-operaation toiselle transaktiolle. Näin vain yksi transaktio kerrallaan voi tulla suoritetuksi ja commitoiduksi. Tämän hintana on perutun transaktion turha työ ja sen vaikutus suoritustehoon. Palaamme näihin lukituksiin myöhemmin.

2.1.2 Likaisen lukemisen ongelma (The Dirty Read Problem)

Likaiseksi lukemiseksi (dirty read) sanotaan lukulukkoja tukevissa järjestelmissä lukuoperaatiota, jossa lukemista varten ei luettavaa kohdetta suojata ensin lukulukolla. Lukija voi saada tietokannasta keskeneräistä, commitoimatonta tietoa, joka voi vielä muuttua tai joka voi peruuntua. Tällainen transaktio ei saisi olla päivittävä, sillä se saattaa kirjoittaa kantaan virheellistä tietoa. Itse asiassa commitoimaton tieto on aina riskaabelia ja voi johtaa vääriin päätöksiin tai toimenpiteisiin.

Kuva 2.3 esittää tyypillistä likaisen lukemisen tapausta. Transaktio B on nostamassa tililtä 500 euroa ja päivittänyt tilin saldoksi 500 euroa, kun transaktio A katsoo tilin saldoa ilman lukulukkoa ja näkee saldoksi vain 500 euroa – väärän tiedon, sillä transaktio peruuntuu.

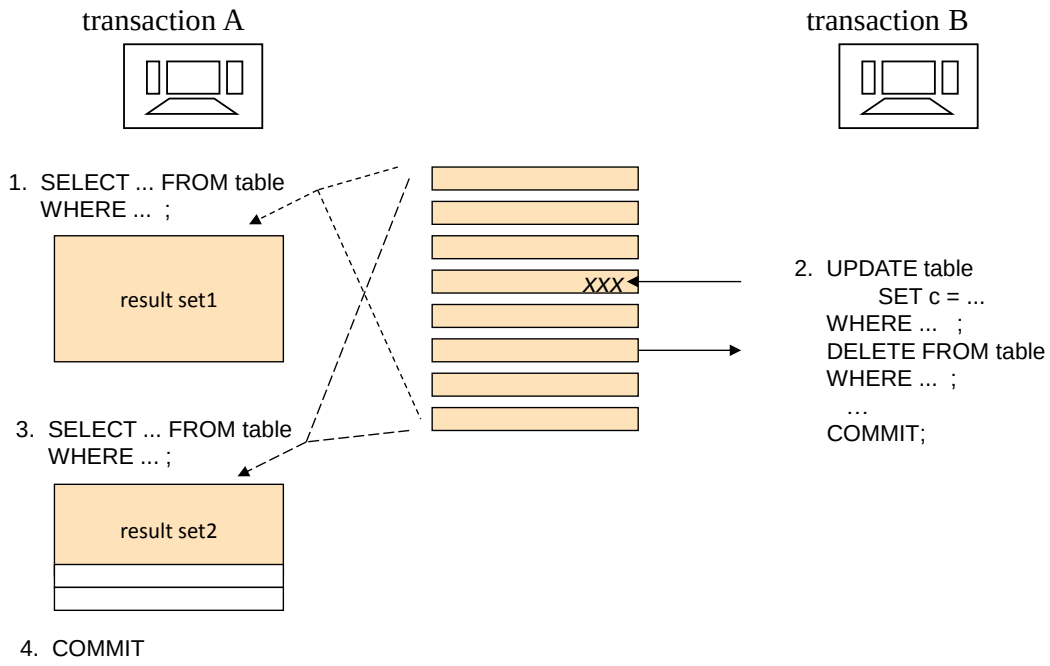


Kuva 2.3 Esimerkki likaisesta lukemisesta

2.1.3 Toistumattoman lukujoukon ongelma (Non-repeatable Read Problem)

Toistumattoman lukujoukon ongelma, josta on kirjallisuudessa käytetty myös nimitystä "inconsistent analysis problem", tarkoittaa että jotkut transaktion tietokannasta lukemat rivit eivät ole luettavissa uudelleen saman transaktion aikana. Samanaikaiset transaktiot ovat saattaneet poistaa jo luettuja rivejä tai päivittää niiden sisältöä siten, etteivät samat hakuehdot enää päde näihin riveihin. Toisin sanoen transaktion lukemien rivien joukko eli lukujoukko voi supistua, jos samoja lukuoperaatioita toistetaan transaktiona aikana.

Kuva 2.4 havainnollistaa toistumattoman lukujoukon ongelmaa. Vaiheessa 1 transaktio A tekee kantaan jonkin kyselyn. Vaiheessa 2 joku samanaikainen transaktio päivittää joidenkin A:n lukemien rivien sisältöjä sellaisiksi etteivät vaiheen 1 kyselyn hakuehto löytäisi näitä rivejä uudelleen ja mahdollisesti poistaa joitakin A:n kannasta lukemia rivejä. Vaiheessa 3 transaktio A uusii vaiheen 1 kyselyn ja ei saa tulosjoukkoonsa enää kaikkia samoja rivejä kuin vaiheessa 1.



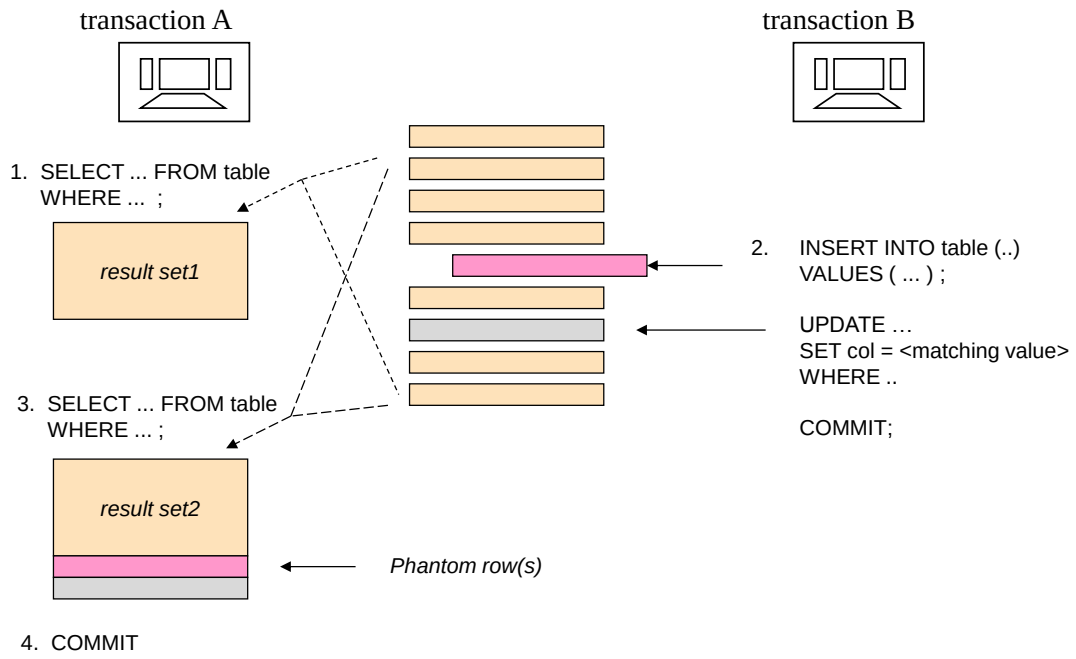
Kuva 2.4 Toistumattoman lukujoukon ongelma

2.1.4 Havaitsemattomien rivien ongelma (The Phantom [Read] Problem)

Havaitsemattomien rivien ongelmaa voisi sanoa myös kasvavan lukujoukon ongelmaksi. ISO SQL-standardi kutsuu tätä Phantom-ongelmaksi ja määrittää sen kuvan 2.5 mukaisesti seuraavasti:
Vaiheessa 1 transaktio A tekee jollakin hakuehdolla kyselyn kantaan. Vaiheessa 2 transaktio B generoi kantaan yhden tai useamman rivin, jotka täyttävät myös A:n kyselyn hakuehdon. Kuvassa 2.5 generointiesimerkkejä ovat rivin lisäys INSERT-komennolla ja joidenkin kannassa aikaisemmin A:n hakuehtoa toteuttamattomien rivien päivitys UPDATE-komennolla sellaisiksi, että ne nyt täyttävät A:n hakuehdon. (Lopuksi transaktio B commitoidaan). Vaiheessa 3 transaktio A tekee uuden haun samalla hakuehdolla kuin vaiheessa 1 ja saa nyt kasvaneen tulosjoukon, jossa näkyvät transaktion B generoimat hakuehdon täyttävät rivit.

Nämä transaktio B:n generoimat rivit ovat siis kannassa transaktion A aikana ja jos A ei tekisi vaiheen 3 kyselyä, ne jäisivät havaitsematta. Näitä generoituja rivejä sanotaan Phantom-riveiksi. Generointikeinon mukaan voitaisiin puhua insert-phantomeista ja update-phantomeista.

ISO SQL:n phantom-määritys voitaisiin tulkita myös siten, että toisten transaktioiden generoimat toistaiseksi huomaamattomat rivit, joihin joku transaktion hakuehto täsmää, mutta joita ei järjestelmän samanaikaisuusmekanismin vuoksi ole edes mahdollista huomata, eivät ole phantom-rivejä.



Kuva 2.5 Havaitsemattomien rivien ongelma

2.2 ACID-periaate transaction ideaalina

Theo Härder ja Andreas Reuter esittivät 1983 ACM Computing Surveys –lehden artikkelissaan monen-käyttäjän ympäristössä luotettavan transaktion ideaaliksi ACID-periaatteen. Nimi tulee ideaalin transaktion seuraavan neljän ominaisuuden englanninkielisistä alkukirjaimista:

Atomic eli **atomisuus**: Transaktion tulee olla atominen eli jakamaton operaatiosarja, joka onnistuu kokonaan tai peruuntuu kokonaan. Tätä kuvaa ilmaisu "All or nothing").

Consistent eli **eheys**: Transaktion operaatiosarja siirtää tietokannan tilan yhdestä ehyestä tilasta toiseen ehyeen tilaan siten, että viimeistään transaktion lopussa mitään eheysrajoitteita (primary key, unique key, foreign key, check) ei ole rikottu transaktion toimesta.

SQL-standardi ja jotkut DBMS-järjestelmät mahdollistavat rajoitetarkastusten viivästämisen transaktion loppuun, mutta useimmat DBMS-järjestelmät tukevat rajoitteiden tarkistusta vain välittömästi jokaisen operation aikana.

Oma eheys-vaatimuksen tulkintamme edellyttää myös koko transaktiologiikan toteuttavan transaktiolle asetetut toimintavaatimukset ja poikkeuskäsittelytarpeet.

Isolated eli **eristyvyys**: Alkuperäinen Härderin ja Reuterin määritelmä on seuraava, "Events within a transaction must be hidden from other transactions running concurrently". Tästä on myöhemmin esitetty myös löyhempiä tulkintoja.

Durable eli **säilyvyys**: Commitoidun transaktion tekemät muutokset tietokantaan säilyvät pysyväismuistissa (levyllä) vaikka järjestelmä kaatuisi. (Katso selitykseksi liite 3)

ACID-periaate ei takaa, että transaktio onnistuisi, mutta se vaatii että transaktio, joka ei täytä näitä ominaisuuksia, tulee peruuttaa joko sovelluksen tai tietokantapalvelimen toimesta.

Tulemme näkemään, että nykyiset DBMS-järjestelmät eivät pysty täyttämään ACID-periaatteen eristyneisyyden täydellistä vaatimusta eli mainokset ACID-kykyisistä DBMS-järjestelmistä todistavat ettei asiaa oikeasti tunneta. Yksittäiset transaktiot tulee kuitenkin pyrkiä rakentamaan eristyneiksi samanaikaisista transaktioista, esimerkiksi välttäen Dirty Read –ongelmia. ACID-periaate on joka tapauksessa tärkeä pedagoginen ajatusmalli.

2.3 Eristyvyystasot (Isolation Levels)

ACID-periaatteen edellyttämä eristyvyysvaatimus on haasteellinen. Riippuen DBMS-järjestelmän käyttämästä samanaikaisuuden hallinnan mekanismista, tiukka eristyvyys voi johtaa samanaikaisuuskonflikteihin ja pitkiin odotusaikoihin hidastaen tietokannan tuotantokäyttöä.

ISO SQL-standardi ei ota kantaa samanaikaisuuden hallinnan mekanismeihin, mutta perustuen edellä esitettyihin samanaikaisuusongelmiin se määrittää kullekin näistä eristyvyystason (isolation level), jolla kyseisen ongelman tulisi ratketa taulukon 2.1 mukaisesti. Taulukko 2.2 listaa ISO SQL:n eristyvyystasojen tarkemmat selitykset ja niiden vastineet DB2-järjestelmissä.

ISO SQL-standardi ei ota kantaa samanaikaisuuden hallinnan mekanismeihin, mutta taustalla on ajatus luku- ja kirjoituslukkoja käyttävästä järjestelmästä (Huom: koko SQL-standardi on aikanaan lähtenyt IBM:n lahjoittamasta DB2:n SQL-versiosta). Tarkastelemme lukituksia myöhemmin. Mitä tiukempi eristyvyystaso, sitä enemmän se voi hidastaa sekä transaktiota itseään että kilpailevia transaktioita.

Kannattaa huomata, että eristyvyystasot eivät mainitse mitään kirjoitusten rajoituksista. Järjestelmästä riippumatta kirjoitus suojataan aina lukoilla ja nämä lukot pidetään aina transaktion loppuun asti estäen hukatun päivityksen ongelman transaktion aikana.

Taulukko 2.1 ISO SQL:n transaktioiden samanaikaisuusongelmat ja ratkaisevat eristyvyystasot

ongelma: / eristyvyystaso:	Hukattu päivitys (Lost Update)	Likainen luku (Dirty Read)	Toistumaton lukujoukko (Non-repeatable Read)	Havaitsematon rivi (Phantom)
READ UNCOMMITTED	Ei mahdollinen	Mahdollinen !	Mahdollinen !	Mahdollinen !
READ COMMITTED	Ei mahdollinen	Ei mahdollinen	Mahdollinen !	Mahdollinen !
REPEATABLE READ	Ei mahdollinen	Ei mahdollinen	Ei mahdollinen	Mahdollinen !
SERIALIZABLE	Ei mahdollinen	Ei mahdollinen	Ei mahdollinen	Ei mahdollinen

Taulukko 2.2 ISO SQL:n ja DB2:n eristyvyystasojen selitykset

Eristyvyystaso		**Eristyvyystason selitys**
ISO SQL	**DB2:n** koodi	
Read Uncommited	UR	sallii commitoimattoman tiedon lukemisen, välittämättä lukoista
Read Commited	CS (CC)	sallii vain commitoidun tiedon lukemisen. Moniversiointia (esitellään myöhemmin) käyttävissä järjestelmissä luetaan aina viimeisin commitoitu riviversio. Lukulukkoja käyttävissä järjestelmissä odotetaan lukulukon saantia, mutta lukemisen jälkeen lukko vapautetaan heti. DB2:ssa eristyvyystason koodi tulee nimestä **Cursor Stability** ja kursorilla luettaessa kursorin osoittama rivi pysyy lukossa kunnes kursoria siirretään eteenpäin. (Versioista 9.7 lähtien DB2:n CS käyttäytyy kuten moniversiointi)
Repeatable Read	RS	sallii vain commitoidun tiedon lukemisen. Lukulukkoja käyttävissä järjestelmissä odotetaan lukulukon saantia ja lukulukot pidetään transaktion loppuun asti, joten luetut rivit voidaan lukea tarvittaessa uudelleen. DB2:ssa eristyvyystason koodi tulee nimestä **Read Stability**
Serializable	RR	sallii vain commitoidun tiedon lukemisen. Estää muilta transaktioilta INSERT, UPDATE ja DELETE komennot käsiteltyihin tauluihin. DB2:ssa eristyvyystason koodi tulee nimestä **Repeatable Read !**

Huom:
Ristiriita ISO SQL:n ja DB2:n eristyvyystasojen nimissä johtuu siitä, että DB2:ssa oli aluksi vain sen nykyiset eristyvyystasot CS ja RR. ISO SQL-standardin kehityksen tekee todellisuudessa ANSI ja jenkki-kirjallisuudessa puhutaankin lähinnä **ANSI SQL**-standardista. ANSI:n SQL työryhmä kehitti standardin 4 eristyvyystasoa ”puhtaalta pöydältä”, mutta DB2:een on vain lisätty uudet eristyvyystasot uusilla nimillä ja koodeilla UR ja RS.

Moniversiointia käyttävissä järjestelmissä on ISO SQL:n eristyvyystasoista poiketen käytössä neljäs eristyvyystaso ”SNAPSHOT”, missä transaktio näkee vain alkuhetkensä mukaisen commitoidun tilanteen tietokannasta. Tämä on käytettävissä PostgreSQL:ssä, Pyrrhossa, SQL Serverissä (konfiguroituna), MySQL/InnoDB:ssä Repeatable Read –nimisenä ja Oraclessa Serializable-nimisenä.

Tarkastelemme myöhemmin mitkä eristyvyystasot ovat missäkin järjestelmässä käytössä ja miten ne on toteutettu. Järjestelmästä riippuen eristyvyystason oletusarvo voidaan konfiguroida tietokantakohtaisesti tai SQL-istunnolle taikka transaktion alussa. Eristyvyystasoa ei standardin mukaan saa vaihtaa enää transaktion aikana, mutta joissakin tuotteissa eristyvyystaso voidaan määrittää lausekohtaisina tai lauseessa taulukohtaisina vihjeinä (hint).

Tyypillinen eristystason oletus järjestelmissä on READ COMMITED, mutta ISO SQL:n mukaan eristystason oletus on SERIALIZABLE ja jos eristystaso määritellään, se tulee tehdä ennen

transaktion alkua. Eristystaso määritetään ISO SQL-standardissa, Oraclessa ja SQL Serverissä seuraavan muotoisella asetuksella

```
SET TRANSACTION ISOLATION LEVEL <isolation level>
```

mutta esimerkiksi DB2:ssa asetuksella

```
SET CURRENT ISOLATION = <isolation level>
```

Riippumattomana DBMS-järjestelmien eroavista syntakseista, eristyvyystasojen nimistä ja erilaisista käyttäytymisistä ODBC ja JDBC API tuntevat eristyvyystasot vain ISO SQL:n käyttämillä nimillä yrittäen häivyttää järjestelmien erot sovellusohjelmoinnissa. Esimerkiksi JDBC API:ssa eristyvyystaso asetetaan connection-olion metodin *setTransactionIsolation* parametrina seuraavasti:

```
<connection>.setTransactionIsolation(Connection.<transaction isolation>);
```

missä `<transaction isolation>` on muodostettu ao. ISO SQL:n eristyvyystason nimestä, jonka alkuun on katenoitu merkkijono "TRANSACTION_", esimerkiksi serializable eristyvyydelle käytetään parametriarvoa TRANSACTION_SERIALIZABLE. Eristyvyystason asetuksesta löytyy esimerkki liitteen 2 Java-ohjelmastamme. Asetettu eristyvyystaso kuvautuu vastaavaksi DBMS-järjestelmän todelliseksi eristyvyystasoksi, jos sellainen on tuettu. Jos vastaavaa eristyvyystasoa ei ole, kirjallisuuden mukaan JDBC-ajuri vaihtaisi automaattisesti eristystason ensimmäiseksi tiukemmaksi eristyvyystasoksi. Kuitenkin esimerkiksi Oraclen JDBC-ajuri laukaisee tästä SQLException-tilanteen. JDBC-ajurin käyttäytymistä voidaan tutkia virtuaalilaboratoriomme Transactions-hakemistosta löytyvällä ohjelmalla **DBMetaData.java**.

2.4 Samanaikaisuuden hallinta (Concurrency Control)

Nykyaikaiset DBMS-järjestelmät käyttävät samanaikaisten transaktioiden luku- ja kirjoitus-operaatioiden eristyvyyteen eli samanaikaisuuden hallintaan (Concurrency Control, CC) seuraavia mekanismeja

- Luku- ja kirjoitusoperaatioiden "varausjärjestelmänä" lukituksia, joita hallitaan monitasoisella lukkotietueiden kirjanpidolla, Multi-Granular Locking scheme[5] (**MGL),** tai jota yleisemmin voisi kutsua nimellä "Locking Scheme concurrency control" (**LSCC**)
- Rivien moniversioinnilla (Multi-Versioning Concurrency Control, **MVCC**), missä muutettujen rivien historiaa talletetaan riittävän kauan, jotta samanaikaisten transaktioiden lukuoperaatioille voidaan tarjota eristyvyystasossa riippuen riittävän oikea-aikainen commitoitu tieto. Näin eliminoidaan lukulukkojen tarve. Kirjoitus-operaatioita varten tarvitaan kuitenkin aina kirjoituslukot.
- Optimistinen samanaikaisuuden hallinta (Optimistic Concurrency Control, **OCC**) ei tarvitse kirjoituslukkoja, sillä kukin transaktio, välittämättä kilpailevista transaktioista, tekee optimistisesti kaikki kirjoitusoperaatiot omaan puskuriinsa, mistä kirjoitusoperaatiot viedään tietokantaan vasta COMMIT-vaiheessa. Täten kukin transaktio näkee vain oman versionsa tietokannan tilasta. Samanaikaisista transaktioista vain ensimmäinen commitoija voittaa ja kilpailevat transaktiot peruuntuvat automaattisesti.

[5] In literature some authors call locking schemes (LSCC) as "pessimistic concurrency control" (PCC) and MVCC as "optimistic concurrency control" although the real OCC has different concurrency semantics. MVCC is used for reading, but still needs some LSCC to protect writing.

2.4.1 Lukituspohjainen samanaikaisuuden hallinta

Implisiittinen lukinta

Samanaikaisuusongelmat on perinteisissä verkkomalliin perustuvissa tietokantajärjestelmissä hoidettu tiedostojen tai tietueiden vahvoilla lukituksilla, missä vain yksi client kerrallaan voi lukita kohteen käyttöönsä, kunnes vapauttaa lukituksen. Relaatiotietokannoissa käsittely perustuu optimoituihin joukko-operaatioihin ja lukitusten on katsottu olevan ”liian vaikeita ihmisten ratkaistaviksi”. Niinpä SQL-standardissa päädyttiin ratkaisuun, jossa ohjelmoija määrittää transaktiolle sopivan eristyvyystason. Standardi ei ota kantaa eristyvyyden toteutukseen, vaan se on jätetty tuotekohtaisesti ratkaistavaksi. Lukituksiin perustuvissa ratkaisuissa DBMS:n optimoija laatii komennon suoritussuunnitelman ja viimekädessä lukkomanager (lock manager) päättää minkälaisen lukituksen kukin operaatio tarvitsee suojakseen. Sovelluskoodin ei tarvitse tietää lukituista kohteista eikä huolehtia lukitusten vapautuksista eli lukinta on sovellusohjelman kannalta implisiittistä.

Rivitason lukinnat

Taulukko 2.3 kuvaa miten saman rivin käsittelyä voidaan hallita luotettavasti samanaikaisten transaktioiden osalta suojaten lukemisia lukulukoilla ja kirjoittamisia kirjoituslukoilla.

Koska vain yksi transaktio kerrallaan voi kirjoittaa (INSERT, UPDATE tai DELETE operaatiolla) samalle riville, DBMS-järjestelmä yrittää hankkia transaktiolle ensin eksklusiivisen (**exclusive**) kirjoituslukon eli X-lukon (**X-lock**) riville. Eksklusiivinen lukko saadaan vain jos kohteeseen ei ole millään muulla clientilla mitään lukkoa (tarkastelemme jatkossa miten tähän vaikuttavat vielä myös muiden kohteiden lukot). Jos lukkoa ei saada heti, jää transaktio odottamaan lukon saantia. Vasta kun X-lukko on saatu, sovellus voi suorittaa kirjoitusoperaationsa. X-lukko ei estä transaktiota itseään myös lukemasta rivin sisältöä. Transaktion X-lukot vapautuvat vasta transaktion lopussa.

Jos transaktion eristyvyystasoksi on asetettu READ UNCOMMITTED ja DBMS sallii sen, niin DBMS ei suojaa lukuoperaatiota lukoilla, vaan transaktio saa suorittaa lukuoperaationsa” omalla vastuullaan”. Muilla ISO SQL:n eristyvyystasoasetuksilla, kun sovellus haluaa lukea jonkin rivin, DBMS yrittää ensin hankkia transaktiolle lukulukon eli jaetun S-lukon (**shared lock**, **S-lock**) riville. Jos riville on ennestään jollakin toisella X-lukko, transaktio jää odottamaan lukon vapautumista. Lukeminen ei muuta rivin sisältöä, joten riville voi samanaikaisesti olla lähes rajaton määrä lukulukkoja. Lukon saannin jälkeen transaktio voi lukea rivin sisällön. Lukulukon vapautuminen riippuu eristyvyystasosta: Jos eristyvyystaso on READ COMMITTED, niin lukulukko vapautuu välittömästi. Eristyvyystasoilla REPEATABLE READ ja SERIALIZABLE lukulukot vapautuvat vasta transaktion lopussa.

Jos rivin lukenut transaktio haluaakin päivittää tai poistaa rivin, tarvitsee se S-lukon kasvatuksen X-lukoksi, minkä se voi saada vasta kun kaikki kilpailevien transaktioiden samanaikaiset lukot kohteeseen on vapautettu. Kun transaktio saa X-lukon, sen S-lukko vapautuu, sillä relaatiokannoissa transaktiolla voi olla samaan kohteeseen vain yksi lukko kerrallaan.

S- ja X-lukkojen lisäksi jotkut järjestelmät käyttävät U-lukkoja. U-lukko on tavallaan X-lukkoa seuraavaksi jonottavan lukko. Se voi olla vain yhdellä kerrallaan, mutta sen voi saada, vaikka toisilla olisi vielä S-lukkoja.

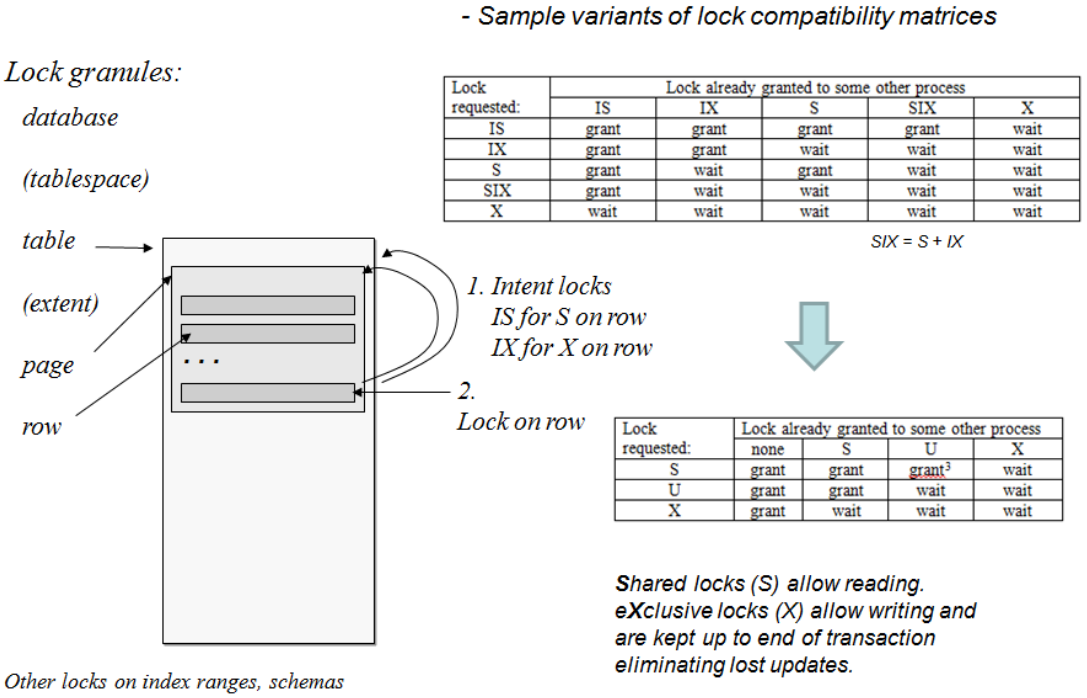
Taulukko 2.3 S- ja X-lukkojen yhteensopivuus

Lukko, jonka transaktio tarvitsee riville	Jos millään toisella transaktiolla ei ole lukkoa riville	Jos jollakin toisella on jo seuraava lukko riville	
		S- lukko	**X- lukko**
S-lukko	saa lukon	saa lukon	odottaa lukkoa
X- lukko	saa lukon	odottaa lukkoa	odottaa lukkoa

Monitasoiset lukinnat (Multi-Granular Locking, MGL)

Rivitason lukitus ei estä samanaikaisia transaktioita käsittelemästä muita rivejä. Suoriin rivitason lukituksiin päästään, jos komennon suoritussuunnitelma voi käyttää indeksejä rivin paikallistamiseen. Riippuen järjestelmästä, komennon suoritussuunnitelman perusteella järjestelmä voi päätyä myös koko sivutason, taulutason, tai indeksivälin (index range) lukituksiin. DDL-komentoja varten järjestelmä voi lukita myös käsiteltävän scheman.

Kuinka sitten järjestelmä valvoo eri rakeisten (granule) lukkojen yhteensopivuutta, esimerkiksi että rivitasolla ei tehtäisi sellaista mikä rikkoisi taulutason lukituksia? Ratkaisuksi tähän on kehitetty aielukitus (intent locking), jota selittää kuva 2.6.



Lock requested:	Lock already granted to some other process				
	IS	IX	S	SIX	X
IS	grant	grant	grant	grant	wait
IX	grant	grant	wait	wait	wait
S	grant	wait	grant	wait	wait
SIX	grant	wait	wait	wait	wait
X	wait	wait	wait	wait	wait

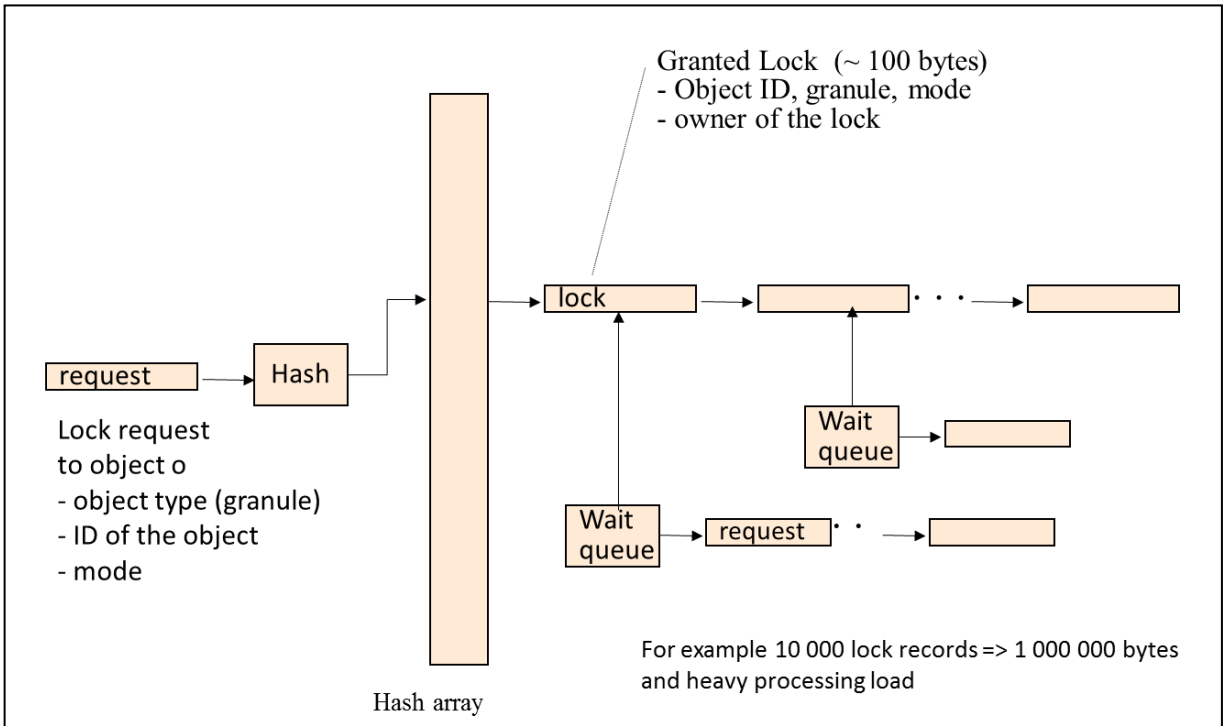
Lock requested:	Lock already granted to some other process			
	none	S	U	X
S	grant	grant	grant[3]	wait
U	grant	grant	wait	wait
X	grant	wait	wait	wait

Kuva 2.6 Eritasoisten lukkojen yhteensopivuuden valvonta

Ennen rivitason S- tai X-lukkoa, jonotetaan vastaava intent lukko, S-lukolle IS-lukko ja X-lukolle IX-lukko ensin taulutasolle, sitten mahdollisesti sivutasolle ja vasta lopuksi jonotetaan tarvittu rivitason lukko.

Jos samalla transaktiolla on taulutasolle S-lukko ja se tarvitsee jollekin taulun riville X-lukon mikä generoi taulutasolle IX-lukon, yhdistetään S ja IX-lukot yhdeksi SIX-lukoksi, koska

samaan kohteeseen transaktiolla voi olla vain yksi lukko. Lukkojen ja lukko-odotusten hallinta tapahtuu DBMS:n muistissa suunnilleen kuvan 2.6b tapaisesti hajautettujen lukkotietueketjujen avulla. Jos lukkoja kertyy paljon, ne vievät tilaa muistissa ja niiden käsittely Lukkotietueet vievät tilaa ja niiden käsittely kuluttaa DBMS:n resursseja.



Kuva 2.6b Lukkojen hallinta ketjutetuilla tietueilla

Kun taulun rivilukkojen määrä ylittää tietyn rajan, järjestelmä pyrkii säästämään lukkokirjanpidon vaatimaa tilaa vaihtamalla rivilukkojen joukon yhdeksi taulutason lukoksi. Tätä kutsutaan rivilukkojen eskaloinniksi (escalation) taulutason lukoksi, jota mahdollisesti joudutaan odottamaan.

Eksplisiittinen lukinta

Eristystasojen generoimat implisiittiset lukitukset eivät aina riitä ja esimerkiksi Oracle- ja MySQL-järjestelmään eristystasot tulivat melko myöhään. Tätä ennen näissä järjestelmissä käytettiin eksplisiittisiä rivi- ja taulutason lukituksia.

Oraclessa ja MySQL:ssä rivitason lukitus tehdään SELECT-lauseen loppuun liitetyllä FOR UPDATE -optiolla siten, että kaikki lauseen hakuehdon löytämät rivit lukitaan explisiittisesti. Nämä lukot vapautuvat vasta transaktion lopussa.

Rivilukituksia yleisempiä ovat taulutason lukitukset LOCK TABLE –komennolla. Taululukituksen moodi voi olla SHARED tai EXCLUSIVE.

DB2, Oracle ja MySQL –järjestelmissä sovellus voi myös vapauttaa LOCK TABLE komennolla lukitun taulun UNLOCK TABLE –komennolla.

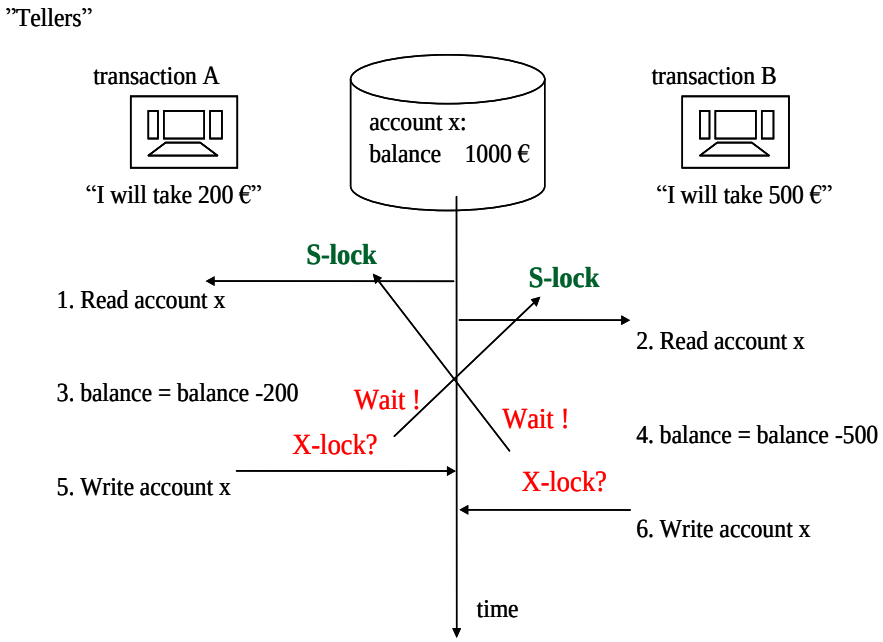
SQL Serverissä eksplisiittinen lukinta voidaan tehdä lause/taulukohtaisesti lauseen WITH-vihjeellä.

Lukko-odotuksista

Lukitusprotokolla ratkaisee transaktion osalta kirjoituslukoilla hukatun päivityksen ongelman, mutta jos kilpailevat transaktiot käyttävät myös transaktion loppuun kestäviä lukulukkoja, johtaa lukitusprotokolla toiseen ongelmaan: kuva 2.7 esittää tilannetta missä kilpailevien transaktioiden saamat lukulukot (S-lock) pakottavat molemmat odottamaan ”ikuisesti” toista osapuolta kirjoituslukkojen (X-lock) saamiseksi. Näin syntynyttä konfliktia kutsutaan **deadlock**-tilanteeksi.

Useissa järjestelmissä on mahdollista asettaa lukko-odotukselle aikaraja **TIMEOUT**-parametrilla, esimerkiksi 10 sekunniksi. Jos lukkoa ei saada tässä ajassa, odottava komento perutaan ja järjestelmä nostaa clientille exceptionin. Joissakin järjestelmissä LOCK TABLE –komentoon voidaan liittää NOWAIT optio, jolloin jos lukkoa ei saada heti, komento kaatuu exceptioniin.

Nykyaikaiset DBMS-järjestelmät pystyvät selviävät deadlock-tilanteesta varsin nopeasti **deadlock detector** –säikeen avulla. Säie vahtii lukkojen odotusketjuja torkahtaen aina noin parin sekunnin ajaksi. Tietokannan hoitaja voi säätää tätä säikeen nukkuma-ajan pituutta. Kun säi löytää joidenkin transaktioiden odotussyklin, se valitsee yhden transaktioista uhriksi (**deadlock victim**) ja tekee tälle automaattisen rollback-operaation.



Kuva 2.7 Deadlock-tilanteeseen johtanut hukatun päivityksen ratkaisu

Uhri saa deadlock-tilanteesta virheilmoitukseen. Sen tulisi nyt odottaa joku sekunnin osa, jotta tilanteen voittanut transaktio ehtii saada työnsä valmiiksi, ja sen jälkeen käynnistää menetetty transaktio uudestaan. Tästä on esimerkki liitteen 2 BankTransfer java-ohjelman "**retry wrapper**"-osassa. Toisin kuin joissakin kirjoissa väitetään, DBMS ei voi käynnistää uhriksi joutunutta transaktiota uudelleen, sillä transaktion logiikka voi riippua tietokannan tilasta ja tilahan on voinut muuttua.

Yleinen väärinkäsitys on, että deadlock ja sen purku olisi joku virhetilanne. Se on poikkeustilanne ja DBMS:n purkupalvelun avulla tuotanto voi jatkua muuten ”mahdottomasta tilanteesta”.

Oracle-järjestelmässä deadlock-tilanne on erilainen kuin edellä. Siinä rivitason lukituksia ei hallita lukkotietueilla, vaan transaktio lukitsee rivin leimaamalle rivi transaktion järjestelmältä saamalla SCN-järjestysnumerolla. Jokaisen komennon osalta Oracle tarkistaa heti, johtaisiko se deadlock-tilanteeseen, ja jos näin on, komento peruutetaan ja client saa deadlock exception ilmoituksen. Komennon transaktio tavallaan valitaan siis uhriksi, mutta sille ei tehdä automaattista rollback-operaatiota, vaan clientin tulee itse tehdä ROLLBACK.

2.4.2 Moniversiointi (MVCC)

Rivien moniversiointi-tekniikka (**multi-versioning concurrency control**, **MVCC**) mahdollistaa käynnissä oleville transaktioille luettavaksi niiden alkuhetkien mukaiset **snapshot**-tilannekuvat tietokannasta. Tätä varten DBMS-järjestelmä leimaa kaikki lisätyt tai päivitetyt rivit transaktion järjestysnumerolla, jota kuvassa 2.8 vastaa Oraclen ”system change number” SCN, ja ylläpitää kaikkien rivien päivityksistä riittävän pitkää historiaa, jotta versioiden joukosta on luettavissa transaktioiden mahdollisesti tarvitsemat commitoidut snapshot-tilanteet. Moniversiointi palvelee siis vain lukuoperaatioita siten, ettei lukulukkoja tarvita ja lukuoperaatioiden ei siis koskaan tarvitse odottaa lukulukon saantia. MVCC-järjestelmät tarjoavat vain kaksi eristyvyystasoa:

- LATEST COMMITTED näkee joko rivin taulussa, jos se ei ole kirjoituslukossa eli se sisältää vain commitoitua tietoa, tai jos rivi on lukossa, rivin viimeisimmän commitoidun version historiaketjun lopussa (jonoon viimeksi viedyn version).
- SNAPSHOT näkee joko rivin taulussa, jos se ei ole kirjoituslukossa ja sen SCN on pienempi kuin transaktion oma SCN, tai etenee rivin historiaketjua ensimmäiseen riviin, jonka SCN on pienempi kuin transaktion oma SCN.

Snapshot on aluksi eheä kuva tietokannasta tiettynä hetkenä. Transaktio voi kuitenkin myös kirjoittaa tietokantaan muuttaen näin näkemistään tietokannan sisällöstä. Eristyvyystasosta riippumatta rivien INSERT, UPDATE ja DELETE–operaatiot tapahtuvat kirjoituslukon suojaamina. MVCC-järjestelmissä rivi lukitaan leimaamalla se transaktion SCN-numerolla. Rivi on tämän jälkeen lukossa niin kauan kuin sen SCN-numero löytyy aktiivien transaktioiden SCN-kirjanpidosta. Jos päivitettävä rivi onkin vielä lukossa, kirjoitusoperaatio kaatuu, generoi seuraavan tapaisen virheilmoituksen "Your snapshot is too old" ja koko transaktio peruuntuu automaattisesti.

Snapshot-eristyvyystaso ei estä kilpailijoiden tietokantaan tekemiä muutoksia, **phantom**-rivejä, eikä näe niitä muuten kuin välillisesti jos sen kirjoitusyritys törmää tällaiseen. Tavallaan se ratkaisee “Phantom Read” –ongelman. Sen sijaan snapshot-eritysvyystaso voi nähdä tietokannassa “olemattomia” eli kummitusrivejä "**ghost rows**", jotka voivat paljastua vain, jos niitä yritetään päivittää.

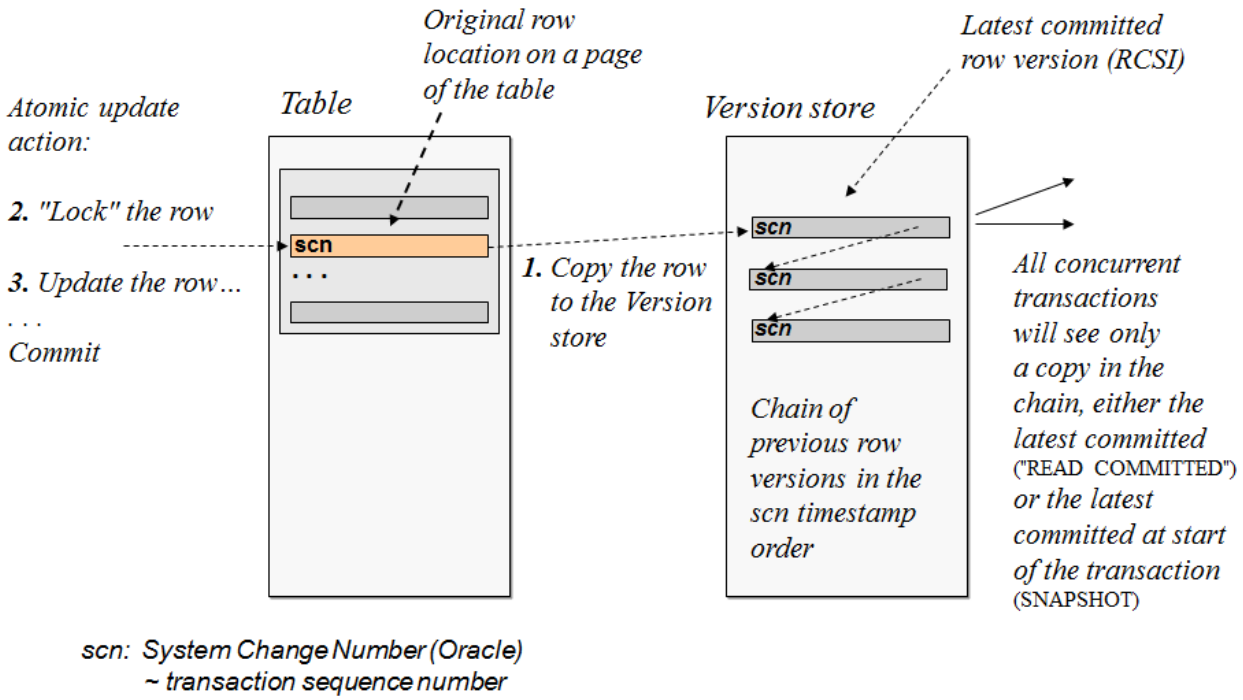
MVCC-toteutuksia

Virtuaalilaboratoriossamme MVCC on käytössä Oraclessa, MySQL/InnoDB:ssä ja PostgreSQL:ssä. Lisäksi myös SQL Server voidaan konfiguroida käyttämään MVCC-tekniikkaa.. Kalen Delaney on kuvannut tätä kattavasti kirjassaan ”SQL Server Concurrency – Locking, Blocking and Row Versioning”, 2012.

MVCC-tekniikan mahdollistamat eristyvyystasot poikkeavat ANSI/ISO SQL:n eristyvyys-tasoista[6]. Järjestelmien näille käyttämät nimet ovat joko harhaanjohtavia tai poikkeavat toisistaan, joten olemme keksineet itse kuvaavamman nimen ”Latest Committed[7]”. Seuraava taulukko listaa järjestelmien näille käyttämät nimet:

Taulukko 2.3b MVCC-järjestelmien eristyvyystasot

DBMS-järjestelmä	**LATEST COMMITTED**	**SNAPSHOT**
Oracle	READ COMMITTED (oletus)	SERIALIZABLE
SQL Server	READ COMMITTED (RCSI, oletus)	SNAPSHOT (SI)
MySQL/InnoDB	READ COMMITTED	REPEATABLE READ (oletus)
PostgreSQL	READ COMMITTED (oletus)	SERIALIZABLE



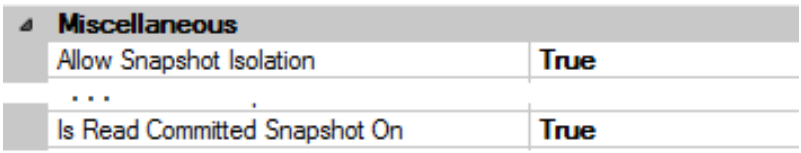
Kuva 2.8 Rivien moniversioinnin (MVCC) historiaketjutus

[6] SQL-standardin eristyvyystasoja onkin kritisoitu jo 1995 Berensonin ja kumppaneiden kirjoittamassa artikkelissa ”A Critique of ANSI SQL Isolation Levels”, joka löytyy mm. osoitteesta http://research.microsoft.com/jump/69541

[7] DB2:n Cursor Stability –eristyvyyden semantiikka on muutettu Latest Committed –eristyvyydeksi ja sille käytetään myös nimeä Currently Committed. Version storena riveille toimii tässä lokipuskuri. Rivihistorian ketjuja ei kuitenkaan tueta DB2:ssa, paitsi taulukohtaisesti luotaville history-tauluihin. Snapshot ei ole tuettu.

Kuva 2.8 esittää MVCC-tekniikan yleistyksenä Oraclen ja SQL Serverin toteutuksista. Oraclen SCN-numeroa vastaa SQL Serverin XSN, MySQL/InnoDB:n DB_TRX_ID ja PostgreSQL:n XMIN. Termi ”version store” rivien historiaketjujen talletuspaikasta on Delaneyn käyttämä. Oraclessa paikka on tietokannan UNDO TABLESPACE, SQL serverissä instanssin TempDB-tietokanta, MySQL/InnoDB:ssä rollback segmentit.

Versiosta 2005 lähtien yksittäinen SQL Server instanssin tietokanta voidaan konfiguroida käyttämään rivien moniversiointia kuvassa 2.9 esitetyillä optioilla



Kuva 2.9 SQL Server 2012 configurointi tukemaan MVCC-tekniikkaa

MySQL/InnoDB:n samanaikaisuuden hallinta on todellinen hybridi, jossa eristyvyystasojen toteutukset vaihtelevat seuraavasti:

- `READ UNCOMMITED` lukee rivejä ilman S-lukkojen suojaa eli sallii Dirty Readin,
- `READ COMMITTED` lukee rivin viimeisimmän commitoidun version,
- `REPEATABLE READ` vastaa Snapshot-eristyvyyttä MVCC-tekniikalla
- `SERIALIZABLE` käyttää monitasoisia lukituksia (MGL) estäen phantom-rivit.

Eksplisiittisten lukitusten avulla voidaan näkymättömien Phantom-rivien ilmaantuminen tietokantaan estää tarvittaessa myös MVCC-toteutuksissa.

Oraclessa, MySQL:ssä ja PostgreSQL:ssä transaktio voidaan myös rajata vain READ ONLY transaktioksi, jolloin se ei voi tehdä päivityksiä, sarjallistuvuuskonfliktit eivät ole mahdollisia ja samanaikaisuuden valvonta siis kevenee DBMS:n osalta. READ ONLY on ISO SQL-standardin mukainen transaktion optio.

2.4.3 Optimistinen samanaikaisuuden hallinta (OCC)

Tästä esiintyy monenlaisia tulkintoja. Joidenkin mielestä MVCC on optimistinen menetelmä, silloin kun Snapshotin päivitys estetään konfliktitilanteissa. DB2:ssa ja SQL Serverissä voidaan kursorikäsittelyyn määrittää optimistiseksi kutsuttu riviversion tarkastava päivityskäsittely.

Alkuperäinen optimistinen samanaikaisuuden hallinta (optimistic concurrency control, OCC) tarkoittaa toteutusta, jossa kaikki transaktion tekemät kirjoitusoperaatiot tehdään ensin client-kohtaiseen puskuriin ja synkronoidaan tietokantaan vasta COMMIT-vaiheessa siten, että samaan aikaan käynnissä olleista transaktioista ensimmäinen commitoija voittaa ja muille tehdään automaattinen ROLLBACK. Tämä on toteutettu tietokantalaboratoriomme järjestelmistä vain Pyrrhossa, jonka valmistaja on University of the West of Scotland. Pyrrhon ainoa ja implisiittinen eristyvyystaso on SERIALIZABLE. Uusimmat tiedot ja versiot Pyrrho:sta löytyvät osoitteesta http://**www.pyrrhodb.com.**

2.4.4 Yhteenvetoa

ISO SQL-standardia kehittää ANSI, jolle IBM lahjoitti 1980-luvulla DB2:n SQL-kielen version standardin pohjaksi. Samanaikaisuuden hallintamekanismi DB2:ssa perustuu monitasoiseen lukitukseen (Multi-Granular Locking, MGL) ja tuolloin DB2:sa oli vain eristyvyystasot Cursor Stability (CS) ja Repeatable Read (RR). Tämä on vaikuttanut ANSI/ISO SQL:n eristyvyystasojen määrityksiin, jotka voidaan ymmärtää perustuvan ajatuksiin käytettävissä olevista lukitus-mekanismeista. SQL-standardi ei virallisesti ota kantaa toteutusten tekniikkaan. Niinpä MVCC-tekniikkaa käyttävät järjestelmät ovat pyrkineet näyttämään yhteensopivuutta SQL-standardin kanssa käyttäen eristyvyystasoille samoja nimiä kuin standardissa on käytetty, vaikka semantiikka ei ole aivan sama. Tämä on aiheuttanut sekaannusta ja ryhmä standardin kehittäjiä ja oppikirjojen kirjoittajia on puuttunut ongelmiin artikkelilla "A Critique of ANSI SQL Isolation Levels" (Berenson et al, 1995).

Seuraavasssa luvussa esiteltävien samanaikaisuusharjoitusten scriptit on sovitettu kaikille laboratoriomme järjestelmille. Erityisesti harjoitus 2.7 tuottaa eri järjestelmillä mielenkiintoisia eroavia tuloksia.

Taulukkoon 2.4 olemme koonneet DBMS-järjestelmien eroavaisuuksia

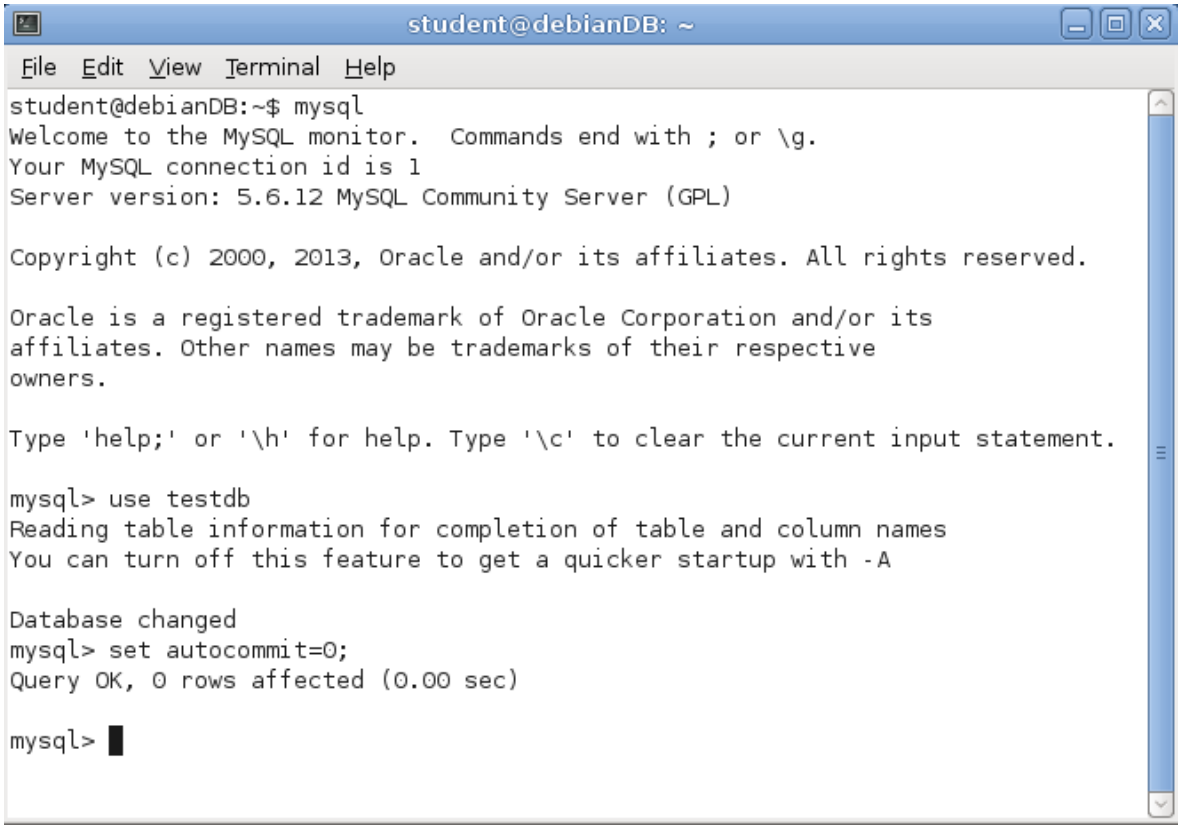
Taulukko 2.4 ISO SQL-standardin ja DBMS-järjestelmien transaktiokäsittelyn eroja

	ANSI/ISO SQL	DB2	Oracle	SQL SERVER	MySQL/InnoDB	PostgreSQL	Pyrrho
	SQL:2006	LUW 9.7	12g1	2012	5.6	9.2	4.8
autocommit (server-side)	n/a	n/a	n/a	yes	yes	yes	yes
Transaction Limits							
explicit start	yes	n/a	n/a	yes	yes	yes	yes
implicit start	yes	yes	yes	(configurable)	(configurable)	n/a	n/a
COMMIT	yes	yes	yes	yes	yes	yes	yes
implicit commit on DDL	n/a	n/a	yes	n/a	yes	n/a	n/a
ROLLBACK	yes	yes	yes	yes	yes	yes	yes
implicit rollback on concurrency conflict (deadlock)	yes	yes	no (exception raised)	yes	yes	no (transaction invalidated)	yes, at commit
implicit rollback on error	implementation dependent	n/a	n/a	(configurable)	n/a	no (transaction invalidated)	yes
SAVEPOINT	yes	yes	yes	yes	yes	yes	n/a
ROLLBACK TO SAVEPOINT	yes	yes	yes	yes	yes	yes	n/a
RELEASE SAVEPOINT	yes	yes	yes	n/a	yes	yes	n/a
Isolation levels							
READ UNCOMMITTED	yes	UR	n/a	yes	yes	n/a (1)	n/a
"read latest committed"	n/a	CS (currently committed)	"read committed"	(configurable)	"read committed"	"read committed"	n/a
READ COMMITTED	yes	CS	n/a	yes	n/a	n/a (2)	n/a
REPEATABLE READ	yes	RS	n/a	yes	n/a	n/a (2)	n/a
snapshot		n/a	"serializable"	(configurable)	"repeatable read"	"serializable"	"serializable"
SERIALIZABLE	yes	RR	explicit locking	yes	yes	explicit locking	"serializable"
note: isolation levels in upper-case stand for ISO/SQL semantics						(1) migrate to "read latest committed"	
						(2) migrate to snapshot	

Taulukosta 2.4 ilmenee, että DB2 on ainoa laboratoriomme DBMS, joka ei tue Snapshot-eristyvyyttä.

2.5 Samanaikaisuuden hands-on -harjoitukset

Aloitetaan uusi mysql-istunto kuvan 2.10 mukaisesti:

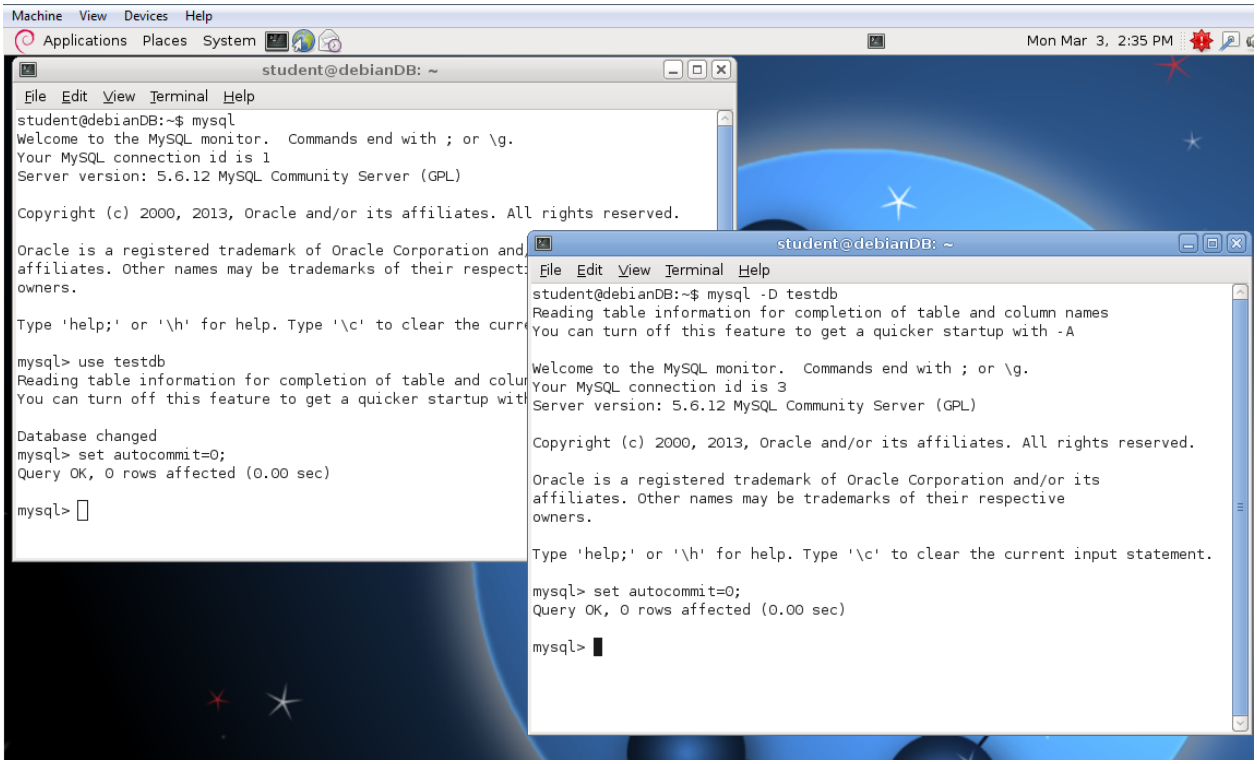


Kuva 2.10 MySQL user session initiation

HARJOITUS 2.0

Samanaikaisuuden harjoituksia varten tarvitsemme myös toisen terminal-ikkunan ja mysql-istunnon kuten kuvassa 2.11. Olkoon vasemmanpuoleinen istunto clientin A ja oikeanpuoleinen clientin B. Molemmat avaavat yhteyden tietokantaan testdb (B näyttää miten tietokanta voidaan valita jo mysql:n komentorivillä) ja kääntävät autocommit-moodin pois päältä:

```
use testdb
SET AUTOCOMMIT = 0;
```



Kuva 2.11 Tietokantalaboratorion terminal / mysql-istuntoja samaan tietokantaan samanaikasuusongelmien testaamista varten

Samanaikaiset istunnot voivat hidastaa toisiaan, kuten tulemme näkemään. Siksi transaktioiden tulisi olla mahdollisimman lyhyitä tehden vain tarvittavat tehtävät. Dialogia loppukäyttäjän kanssa tulee välttää transaktion aikana, sillä tämä hidastaisi kilpailevia transaktiota kohtuuttomasti ja voisi moni-käyttäjäympäristössä johtaa katastrofaalisiin odotusaikoihin tuotantoympäristössä. On siis tärkeää, että kontrolli ei siirry käyttäjälle missään vaiheessa transaktion aikana.

Harjoitukset perustuvat tilisiirtoesimerkkiimme. Testiaineistoa varten poistamme varmuuden vuoksi vanhan Accounts-taulun ja luomme tilalle uuden lisäten siihen 2 tiliä:

```
DROP TABLE Accounts;
CREATE TABLE Accounts (
acctID  INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL,
CONSTRAINT remains_nonnegative CHECK (balance >= 0)
);
SET AUTOCOMMIT = 0;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
```

ISO SQL-standardin mukaan transaktion eristyvyystaso tulee asettaa transaktion alussa ja sitä ei saa muuttaa transaktion aikana. Koska järjestelmissä on tämän suhteen eroja, testaamme seuraavassa kuinka MySQL/InnoDB tämän suhteen käyttäytyy:

```
START TRANSACTION;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT COUNT(*) FROM Accounts;
ROLLBACK;

-- ja toinen yritys eri järjestyksessä:
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT COUNT(*) FROM Accounts;
ROLLBACK;
```

ISO SQL-standardin mukaan READ UNCOMMITTED –eristyvyystason transaktiossa ei ole mahdollista kirjoittaa tietokantaan. Testaamme seuraavassa kuinka MySQL/InnoDB tämän suhteen käyttäytyy:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
DELETE FROM Accounts;
SELECT COUNT(*) FROM Accounts;
ROLLBACK;
```

Kysymyksiä

- Mitä opimme näistä kokeiluista?

HARJOITUS 2.1

Hukantun päivityksen ongelma tarkoittaa, että joku kilpaileva transaktio päällekirjoittaen hukkaa transaktion kirjoittaman tiedon jo ENNEN transaktion päättymistä. Tämä ei ole mahdollista nykyaikaisten DBMS-järjestelmien tapauksessa, sillä kirjoitus suojataan aina lukolla transaktion loppuun asti. Kuitenkin transaktion commitoinnin jälkeen mikä tahansa transaktio voi päällekirjoittaa tiedon katsomatta ensin sen commitoitua sisältöä. Kutsumme tätä sokeaksi päällekirjoitukseksi (**Blind Overwriting).**

Sokea päällekirjoitus tapahtuu esimerkiksi, kun sovellus lukee tiedon tietokannasta, muuttaa arvoa muistissansa ja lopuksi kirjoittaa päivitetyn tiedon takaisin tietokantaan. Jos joku kilpaileva transaktio on välillä päivittänyt tietoa ja commitoinut sen, tulee sovellus hukanneeksi tuon aikaisemman päivityksen.

Taulukon 2.4 harjoituksessa simuloimme sovellusohjelman osuutta käyttäen MySQL:n paikallisia muuttujia, joiden nimen tulee aina alkaa @-merkillä. Muuttujilla voi olla vain yksittäisiä skalaareja arvoja. Harjoituksessa on 2 samanaikaista istuntoa, A ja B, joiden osuudet harjoituksen skenaariossa esitetään vaiheittain omissa sarakkeissaan. Vaiheiden järjestys-numerot on esitetty taulukon vasemmassa laidassa. Tarkoitus on simuloida kuvan 2.2 tililtänostoja, missä A lukee tilin saldon ja nostaa 200 euroa tililtä 101 ja B vastaavasti nostaa 500 euroa samalta tililtä. B:n tekemän sokean päällekirjoituksen vuoksi A:n nostaman 200 euron kohtalo jää pankille tuntemattomaksi.

Jotta harjoitukset olisivat toistettavissa, alustamme aina ennen harjoitusta taulun sisällön seuraavalla alustustransaktiolla:

```
SET AUTOCOMMIT = 0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID, balance) VALUES (101, 1000);
INSERT INTO Accounts (acctID, balance) VALUES (202, 2000);
SELECT * FROM Accounts;
COMMIT;
```

Taulukko 2.4 Sokean päällekirjoituksen ongelma simuloiden @muuttujilla sovellusta

	Session A	Session B

1	`SET AUTOCOMMIT = 0;`
	`SET TRANSACTION`
	`  ISOLATION LEVEL READ COMMITTED;`
	
	`-- Amount to be transfered by A`
	`SET @amountA = 200;`
	`SET @balanceA = 0;  -- Init value`
	
	`SELECT balance INTO @balanceA`
	`FROM Accounts WHERE acctID = 101;`
	
	`SET @balanceA = @balanceA - @amountA;`
	`SELECT @balanceA;`	

2		`SET AUTOCOMMIT = 0;`
		`SET TRANSACTION`
		`  ISOLATION LEVEL READ COMMITTED;`
		
		`-- Amount to be transfered by B`
		`SET @amountB = 500;`
		`SET @balanceB = 0;  -- Init value`
		
		`SELECT balance INTO @balanceB`
		`FROM Accounts WHERE acctID = 101;`
		
		`SET @balanceB = @balanceB - @amountB;`

3	`UPDATE Accounts`
	`SET balance = @balanceA`
	`WHERE acctID = 101;`	

4		`set lock_wait_timeout = 300;`
		`UPDATE Accounts`
		`SET balance = @balanceB`
		`WHERE acctID = 101;`

5	`-- continue without waiting for B!`
	`SELECT acctID, balance`
	`FROM Accounts`
	`WHERE acctID = 101;`
	`COMMIT;`	

6		`SELECT acctID, balance`
		`FROM Accounts`
		`WHERE acctID = 101;`
		`COMMIT;`

Huom: MySQL:n lukko-odotuksen timeout-oletus on 90 sekuntia. Jos vaiheen 5 osalta jäädään miettimään liian kauan, voi harjoituksen opetus saada väärän luonteen. Tämän vuoksi vaiheen 4 alkuun on lisätty kohtuullisen pitkän miettimisajan varalta 5 minuutin timeout.

Kysymyksiä

- Käyttäytyikö sovellus odotetusti?
- Katosiko rahaa?

HARJOITUS 2.2a

Toistetaan SELECT-UPDATE harjoitus 2.1, mutta nyt käyttäen eristyvyystasoa SERIALIZABLE.

Palautetaan aina ensin alkutilanne:

```
SET AUTOCOMMIT=0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
```

Taulukko 2.5a Skenaario 2.1 pitäen S-lukot transaktion loppuun asti

	Session A	**Session B**

1	`SET AUTOCOMMIT = 0;`
	`SET TRANSACTION`
	`ISOLATION LEVEL SERIALIZABLE;`
	
	`-- Amount to be transfered by A`
	`SET @amountA = 200;`
	`SET @balanceA = 0;  -- Init value`
	
	`SELECT balance INTO @balanceA`
	`FROM Accounts WHERE acctID = 101;`
	
	`SET @balanceA = @balanceA - @amountA;`
	
	`SELECT @balanceA;`	

2		`SET AUTOCOMMIT = 0;`
		`SET TRANSACTION`
		`ISOLATION LEVEL SERIALIZABLE;`
		
		`-- Amount to be transfered by B`
		`SET @amountB = 500;`
		`SET @balanceB = 0;  -- Init value`
		
		`SELECT balance INTO @balanceB`
		`FROM Accounts WHERE acctID = 101;`
		
		`SET @balanceB = @balanceB - @amountB;`

3	`set lock_wait_timeout = 300;`
	`UPDATE Accounts`
	`SET balance = @balanceA`
	`WHERE acctID = 101;`	

4		`-- continue without waiting for A!`
		`set lock_wait_timeout = 300;`
		`UPDATE Accounts`
		`SET balance = @balanceB`
		`WHERE acctID = 101;`

5	`SELECT acctID, balance`	

	FROM Accounts WHERE acctID = 101; **COMMIT;**	
6		SELECT acctID, balance FROM Accounts WHERE acctID = 101; **COMMIT;**

Kysymyksiä

a) Mitä tästä opittiin?
b) Entä jos molemmat transaktiot käyttäisivät REPEATABLE READ eristyvyystasoa?

HARJOITUS 2.2b

Toistetaan harjoitus 2.2a, mutta nyt käyttäen SELECT–UPDATE skenaariossamme "sensitiivisiä päivityksiä" ilman sovelluskoodin simulointia paikallisilla muuttujilla.

Palautetaan ensin tauluun alkutilanne:

```
SET AUTOCOMMIT = 0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
```

Taulukko 2.5b SELECT – UPDATE –skenaario soveltaen sensitiivisiä päivityksiä

	Session A	**Session B**

1	`SET AUTOCOMMIT = 0;`
	`SET TRANSACTION`
	`  ISOLATION LEVEL SERIALIZABLE;`
	
	`SELECT balance FROM Accounts`
	`WHERE acctID = 101;`	

2		`SET AUTOCOMMIT = 0;`
		`SET TRANSACTION`
		`  ISOLATION LEVEL SERIALIZABLE;`
		
		`SELECT balance FROM Accounts`
		`WHERE acctID = 101;`

3	`UPDATE Accounts`
	`SET balance = balance - 200`
	`WHERE acctID = 101;`	

4		`set lock_wait_timeout = 300;`
		`UPDATE Accounts`
		`SET balance = balance - 500`
		`WHERE acctID = 101;`

5	`-- continue without waiting for B!`
	`SELECT acctID, balance`
	`FROM Accounts`
	`WHERE acctID = 101;`
	
	`COMMIT;`	

6		`SELECT acctID, balance`
		`FROM Accounts`
		`WHERE acctID = 101;`
		
		`COMMIT;`

Kysymys

- Mitä tästä opittiin?

HARJOITUS 2.3 UPDATE–UPDATE -kilpailu kahdesta resurssista eri järjestyksessä

Palautetaan ensin tauluun alkutilanne:

```
SET AUTOCOMMIT = 0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
```

Taulukko 2.6 UPDATE-UPDATE Scenario

	Session A	**Session B**

1	`SET AUTOCOMMIT = 0;`
	`SET TRANSACTION`
	`ISOLATION LEVEL READ COMMITTED;`
	
	`UPDATE Accounts`
	`SET balance = balance - 100`
	`WHERE acctID = 101;`	

2		`SET AUTOCOMMIT = 0;`
		`SET TRANSACTION`
		`ISOLATION LEVEL READ COMMITTED;`
		
		`UPDATE Accounts`
		`SET balance = balance - 200`
		`WHERE acctID = 202;`

3	`UPDATE Accounts`
	`SET balance = balance + 100`
	`WHERE acctID = 202;`	

4		`UPDATE Accounts`
		`SET balance = balance + 200`
		`WHERE acctID = 101;`

5	**`COMMIT;`**	
6		**`COMMIT;`**

Kysymys

- Mitä tästä opittiin?

Huom:
Transaktioiden eristyvyystasoilla ei ole merkitystä tässä skenaariossa, mutta on hyvä tapa määrittää eristyvyystaso aina ennen transaktion varsinaista alkua. Saattaahan olla taustalla tapahtuvia lukuoperaatioita, esimerkiksi viiteavainten tarkistuksia tai triggereitä ja eristyvyystaso voi vaikuttaa myös näiden suoritukseen.

HARJOITUS 2.4 Suojaamaton luku (Dirty Read)

Jatketaan samanaikaisuusongelmien kokeiluja. Aloitetaan suojaamattomalla lukuoperaatiolla (Dirty Read). Transaktion A käyttää REPEATABLE READ eristyvyyttä (mikä on myös MySQL:n oletus), kun taas transaktio B käyttää READ UNCOMMITTED eristyvyyttä:

Palautetaan aina ensin tauluun alkutilanne:

```
SET AUTOCOMMIT = 0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
```

Taulukko 2.7 Dirty read problem

	Session A	**Session B**

1	`SET AUTOCOMMIT = 0;`
	`SET TRANSACTION`
	`ISOLATION LEVEL REPEATABLE READ;`
	
	`UPDATE Accounts`
	`SET balance = balance - 100`
	`WHERE acctID = 101;`
	
	`UPDATE Accounts`
	`SET balance = balance + 100`
	`WHERE acctID = 202;`	

2		`SET AUTOCOMMIT = 0;`
		`SET TRANSACTION`
		`ISOLATION LEVEL READ UNCOMMITTED;`
		
		`SELECT * FROM Accounts;`
		`COMMIT;`

3	**`ROLLBACK;`** `SELECT * FROM Accounts;` **`COMMIT;`**	

Kysymyksiä

a) Toimiiko transaktio B luotettavasti? Entä jos se lopuksi nostaisi 2200 euroa tililtä 202?

b) Miten transaktio B tulisi korjata?

HARJOITUS 2.5 Toistumaton lukujoukko **(**Non-Repeatable Read)

Palautetaan aina ensin tauluun alkutilanne:

```
SET AUTOCOMMIT = 0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
```

Taulukko 2.8 Toistumattoman lukujoukon ongelma

	Session A	**Session B**

1	`SET AUTOCOMMIT = 0;`
	`SET TRANSACTION`
	`ISOLATION LEVEL READ COMMITTED;`
	
	`SELECT * FROM Accounts`
	`WHERE balance > 500;`	

2		`SET AUTOCOMMIT = 0;`
		
		`UPDATE Accounts`
		`SET balance = balance - 500`
		`WHERE acctID = 101;`
		
		`UPDATE Accounts`
		`SET balance = balance + 500`
		`WHERE acctID = 202;`
		
		`SELECT * FROM Accounts;`
		`COMMIT;`

3	`-- Repeating the same query`
	`SELECT * FROM Accounts`
	`WHERE balance > 500;`
	
	`COMMIT;`	

Kysymyksiä

a) Saako transaktio A siis saman lukutuloksen vaiheessa 3 minkä se saa vaiheessa 1?

b) Muuttuuko tulos, jos A käyttää eristyvyyttä `REPEATABLE READ` tai `SERIALIZABLE`?

HARJOITUS 2.6

Kokeillaan nyt mitä tapahtuu oppikirjojen **'insert phantom'** -tapaukselle:

Palautetaan aina ensin tauluun alkutilanne:

```
SET AUTOCOMMIT = 0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
```

Taulukko 2.9 Insert phantom problem

	Session A	**Session B**

1	`SET AUTOCOMMIT = 0;`
	`SET TRANSACTION`
	`ISOLATION LEVEL REPEATABLE READ;`
	`START TRANSACTION READ ONLY;`	

2		`SET AUTOCOMMIT = 0;`
		`INSERT INTO Accounts (acctID,`
		`balance) VALUES (301,3000);`
		`COMMIT;`

3	`SELECT * FROM Accounts` `WHERE balance > 1000;`	

4		`INSERT INTO Accounts (acctID,`
		`balance) VALUES (302,3000);`
		`COMMIT;`

5	`-- Can we see accounts 301 and 302?`
	`SELECT * FROM Accounts`
	`WHERE balance > 1000;`
	
	`COMMIT;`	

Kysymyksiä

a) Joutuuko transaktio B odottamaan transaktiota A ja jos joutuu, niin missä vaiheessa?

b) Näkyvätkö transaktio B:n lisäämät uudet tilit 301 tai 302 transaktiolle A?

c) Muuttuuko vaiheen 4 tulos jos vaihdamme vaiheiden 2 ja 3 järjestystä?

d) Miten harjoitus muuttuu, jos A:n eristyvyystaso onkin SERIALIZABLE?

e) *(vaativamman taso kysymys)*
MySQL/InnoDB käyttää MVCC-tekniikkaa REPEATABLE READ eristyvyydelle, mikä on transaction snapshotin todellinen aikaleima, onko se START TRANSACTION asetuksen hetken vai ensimmäisen SQL-komennon hetken aikaleima? Huomaa, että myös transaktionaalisessa moodissa tarvitaan START TRANSACTION komentoa tekemään tiettyjä asetuksia, esimerkiksi määrittämään transaktio READ ONLY transaktioksi
.

HARJOITUS 2.7 SNAPSHOT-kokeilu insert/update phantom ja ghoast –rivien tapauksessa

Avataan kahden terminaalimme lisäksi kolmas terminaali istunnolle C, mitä tarvitsemme myöhemmin kokeilussamme, ja vaihdetaan taulun T rakenne toiseksi:

```
mysql –D testdb

SET AUTOCOMMIT = 0;
DROP TABLE T;

CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(40), i SMALLINT);

INSERT INTO T (id, s, i) VALUES (1, 'first', 1);
INSERT INTO T (id, s, i) VALUES (2, 'second', 2);
INSERT INTO T (id, s, i) VALUES (3, 'third', 1);
INSERT INTO T (id, s, i) VALUES (4, 'fourth', 2);
INSERT INTO T (id, s, i) VALUES (5, 'to be or not to be', 1);
COMMIT;
```

Taulukko 2.10 Insert- ja Update-phantom ongelmat ja haamurivin päivitys

	Session A	**Session B**

1	`SET AUTOCOMMIT = 0;`
	`SET TRANSACTION`
	`ISOLATION LEVEL REPEATABLE READ;`
	
	`SELECT * FROM T WHERE i = 1;`	

2		`SET AUTOCOMMIT = 0;`
		`SET TRANSACTION`
		`ISOLATION LEVEL READ COMMITTED;`
		
		`UPDATE T SET s =` **`'Update by B'`**
		`WHERE id = 1;`
		
		`INSERT INTO T (id, s, i)`
		`VALUES (6,` **`'Insert Phantom'`**`, 1);`
		
		`UPDATE T`
		`SET s =` **`'Update Phantom'`**`, i = 1`
		`WHERE id = 2;`
		
		`DELETE FROM T WHERE id = 5;`
		
		`SELECT * FROM T;`

3	`-- Repeat the query and do updates`
	`SELECT * FROM T WHERE i = 1;`
	
	`INSERT INTO T (id, s, i)`
	`VALUES (7,` **`'inserted by A'`**`, 1);`
	
	`UPDATE T SET s =`
	`'update by A inside the snapshot'`
	`WHERE id = 3;`	

	UPDATE T SET s = **'update by A outside the snapshot'** WHERE id = 4; UPDATE T SET s = **'update by A after update by B'** WHERE id = 1;	
3.5		**-- ClientC:** **-- what's the current content?** SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; SELECT * FROM T;
4		**-- Client B continues** **-- without waiting for A** **COMMIT;** **SELECT * FROM T;**
5	SELECT * FROM T WHERE i = 1; UPDATE T SET s = **'updated after delete?'** WHERE id = 5; SELECT * FROM T WHERE i = 1; **COMMIT;**	
6	SELECT * FROM T; **COMMIT;**	
7		-- Client C does the final select SELECT * FROM T; COMMIT;

Kysymyksiä

a) Näkeekö transaktio A transaction B lisäämät tai päivittämät rivit?

b) Mitä tapahtuu, kun A yrittää päivittää B:n jo päivittämää riviä?

c) Mitä tapahtuu, kun A yrittää päivittää hakuehdolla "id = 5" transaktion B poistamaa riviä?

Listaus 2.1 Harjoitus 2.7:n testituloksia

```
mysql> -- 5. Client A continues
mysql> SELECT * FROM T WHERE i = 1;
+----+--------------------------------+------+
| id | s                              | i    |
+----+--------------------------------+------+
|  1 | update by A after update by B |    1 |
|  3 | update by A inside snapshot   |    1 |
|  5 | to be or not to be            |    1 |
|  7 | inserted by A                 |    1 |
+----+--------------------------------+------+
4 rows in set (0.00 sec)

mysql> UPDATE T SET s = 'updated after delete?' WHERE id = 5;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0
mysql> SELECT * FROM T WHERE i = 1;
```

```
+----+------------------------------+------+
| id | s                            | i    |
+----+------------------------------+------+
|  1 | update by A after update by B |    1 |
|  3 | update by A inside snapshot   |    1 |
|  5 | to be or not to be            |    1 |
|  7 | inserted by A                 |    1 |
+----+------------------------------+------+
4 rows in set (0.00 sec)
mysql> COMMIT;
Query OK, 0 rows affected (0.03 sec)

mysql> -- 6. Client A continues with a new transaction
mysql> SELECT * FROM T;
+----+------------------------------+------+
| id | s                            | i    |
+----+------------------------------+------+
|  1 | update by A after update by B |    1 |
|  2 | Update Phantom                |    1 |
|  3 | update by A inside snapshot   |    1 |
|  4 | update by A outside snapshot  |    2 |
|  6 | Insert Phantom                |    1 |
|  7 | inserted by A                 |    1 |
+----+------------------------------+------+
6 rows in set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

Huom: Vertaa näitä tuloksia liitteessä 1 esitettyyn vastaavaan testiin SQL Server 2012:lla.

3 Suosituksia transaktio-ohjelmointiin

Käyttötapauksen toteutus vaatii tyypillisesti useita tietokantatransaktioita. Monet näistä transaktioista vain hakevat tietoja käyttäjälle, mutta käyttötapauksen viimeinen ”Talleta” painikkeella tms. laukaistava tietokantatransaktio päivittää käyttötapauksen tiedot tietokantaan.

SQL transaktioilla, jopa saman käyttötapauksen toteutuksessa, voi olla erilaisia luotettavuus ja eristyvyysvaatimuksia. Hyvä tapa on määrittää transaktion tarvitsema eristyvyystaso aina jokaisen transaktion alussa.

ISO SQL-standardin mukaan READ UNCOMMITTED eristyvyystaso voidaan määrittää vain READ ONLY –transaktiolle (Melton & Simon 2002). DBMS-järjestelmät eivät pakoita tätä, mutta päivityksiä tulisi välttää READ UNCOMMITTED eristyvyystasolla.

DBMS-järjestelmien samanaikaisuus- ja transaktiopalvelut eroavat toisistaan, joten on tärkeää kehitettävien sovellusten luotettavuuden ja suorituskyvyn vuoksi että sovelluskehittäjät tuntevat käytettävän DBMS-tuotteen käyttäytymisen palvelujen toteutuksissa.

Luotettavuus on tietokantakäsittelyssä tärkein kriteeri, tärkeämpi kuin suoritusteho ym. Kuitenkin useiden DBMS-järjestelmien oletusmoodina on AUTOCOMMIT ja eristyvyystason oletusarvona on tyypillisesti READ COMMITTED suosien suoritustehoa. Riittävä eristyvyystaso tulee suunnitella huolella. Suoritustehoa ajatellen eristyvyystaso ei kuitenkaan saisi olla liikaa kilpailijoita rajoittava. On myös tärkeää muistaa että SERIALIZABLE tarkoittaakin useissa järjestelmissä SNAPSHOT-eristyvyyttä, mikä tosin takaa transaktiolle konsistentin READ ONLY käsittelyn, mutta luotettavia päivityksiä varten voi tarvita suojaksi eksplisiittisiä lukituksia.

Tietokantatransaktioiden tulee olla mahdollisimman lyhyitä välttäen kilpailevien transaktioiden jarruttelua lukituksin. Tämän vuoksi transaktion aikana ei saa keskustelua käyttäjän kanssa. Transaktion tulisikin käsitellä vain tietokantaa siten, että transaktion perumisesta ja mahdollisista uusintayrityksistä ei seuraa sovellukseen sivuvaikutuksia.

Transaktiossa ei pitäisi käyttää DDL-komentoja, sillä joissakin järjestelmissä ne aiheuttavat transaktion implisiittisen commitin, mitä ohjelmoija ei ehkä muista huomioida.

Jokaisella transaktiolla tulee olla huolella määritelty tehtävä, joka alkaa ja loppuu saman sovelluskomponentin samassa sovelluslohkossa. Transaktiossa käytetyt SQL-komennot on syytä testata erikseen vuorovaikutteisella SQL:llä.

Tässä tutorialissa emme ole käsitelleet talletettuja proseduureja (stored procedures, stored routines). Näissä käytetyt SQL:n proseduraaliset laajennukset ovat järjestelmäkohtaisia, koska laajennukset tehtiin standardiin myöhässä totetuksista. Jotkut ohjelmistotalot suosivat transaktioita proseduurien sisällä suoritustehon vuoksi, mutta tässä on omat vaaransa. Transaktio voi tulla automaattisesti peruutetuksia ja siksi proseduuriin sisältyvän sovelluslogiikan tulee varautua myös tähän. Tämä lisää proseduuriin tarvittavaa suunnittelua ja koodia. Jotkut DBMS-järjestelmät eivät edes hyväksy COMMIT-lauseita talletetuissa proseduureissa

SQL-transaktion tekninen konteksti on tietokantayhteys (connection) eli SQL-istunto. Jos sovelllus tarvitsee ehyenä kokonaisuutena useita samanaikaisia tietokantayhteyksiä, tarvitaan hajautettujen transaktioiden (distributed transactions) hallintaa, mitä emme ole käsitelleet.

Jos transaktio kaatuu samanaikaisuuskonfliktiin, esimerkiksi deadlockin uhrina, niin useimmissa tapauksissa sovelluksen ei kannattaisi vaivata tällä heti loppukäyttäjää, vaan transaktio voi hyvinkin onnistua jollakin uudelleenyrityksellä. Tästä olemme esittäneet transaktion retry wrapper – data access pattern'ia. Uudelleenyritysten määrä kasvattaa loppukäyttäjän kokemaa vastausaikaa, joten yrityskerroille on syytä asettaa joku kohtuullinen yläraja. Jos tietokantayhteys on katkennut, on uusintayritys aloitettava avaamalla uusi tietokantayhteys.

On myös tapauksia, joissa transaktio jatkaa aikaisemman transaktion työtä ja joku kilpaileva transaktio on ehtinyt päivittää samoja tietoja siten, että transaktion edellytykset on menetetty. Tällöin uusintayrityksiä ei tule tehdä, vaan on palattava käyttötapauksen alkuun.

Lisätietoja ja materiaalien osoitteita

Lähteitä

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E. & O'Neil, P., "A Critique of ANSI SQL Isolation Levels", Technical Report MSR-TR-95-51, Microsoft Research, 1995. http://research.microsoft.com/apps/pubs/default.aspx?id=69541

Delaney, K., “SQL Server Concurrency – Locking, Blocking and Row Versioning”, Simple Talk Publishing, July 2012

Melton, J. & Simon, A. R., "SQL:1999: Understanding Relational Language components", Morgan Kaufmann, 2002.

Data Management: Structured Query Language (SQL) Version 2, The Open Group, 1996. http://www.opengroup.org/onlinepubs/9695959099/toc.pdf

DBTechNet-verkoston dokumentteja

DB2, Oracle ja SQL Server -järjestelmien samanaikaisuuden hallinnasta

http://www.dbtechnet.org/papers/SQL_ConcurrencyTechnologies.pdf

Dokumentti optimistisesta lukinnasta eli riviversion verifiointitekniikasta (Row Version Verification) erilaisten data access teknologioiden tapauksissa

http://www.dbtechnet.org/papers/RVV_Paper.pdf

Virtuaalilaboratorio

Virtuaalilaboratorion DebianDB version 6 OVA-tiedosto ja dokumentit, mukaan lukien ”Quick Start Guide” ja harjoitustemme skriptit sovitettuina eri järjestelmille löytyvät osoitteesta

http://www.dbtechnet.org/download/DebianDBVM06.zip
(4.8 GB, MySQL 5.6, DB2 Express-C 9.7, Oracle XE 10.1, PostgreSQL 8.4, Pyrrho 4.8)

Liite 1 Transaktiokokeiluja SQL Server -järjestelmällä

Tässä liitteessä teemme kirjan lukujen harjoitukset käyttäen ilmaista SQL Server Express 2012 –järjestelmää, joka toimii vain Windows-alustalla ja ei siis ole Linux-pohjaisessa virtuaalilaboratoriossamme. Tämä ei ole iso ongelma, koska ilmaisen SQL Server Express 2012 järjestelmän voi vapaasti ladata kuitenkin Microsoftin websivuilta. Järjestelmien pienten erojen vuoksi harjoituksissa on joitakin eroja, mutta harjoitusten numeroinnin avulla vastaavat harjoitukset on helposti tunnistettavissa ja näin SQL Serverin osalta esittämiämme tuloksia voi verrata omiin virtuaalilaboratoriossa saatuihin tuloksiin.

MySQL:n osalta käytimme edellä merkkipohjaista mysql-client-ohjelmaa, mutta seuraavassa käytämme SQL Serverin Management Studio –ohjelmaa (SSMS). Luomme aluksi instanssiimme uuden tietokannan "Testdb" oletusarvoin ja avaamme siihen istunnon USE-komennolla. Toisin kuin Linuxissa Windows-alustalla tietokantojen nimet eivät ole case-sensitiivisiä.

```
CREATE DATABASE Testdb;
USE Testdb;
```

Osa 1. Kokeilut yksittäisillä transaktioilla

SQL Server –istunnot toimivat oletuksena AUTOCOMMIT-moodissa, jossa voidaan kyllä käyttää myös eksplisiittisiä transaktioita. SQL Server voidaan konfiguroida “moneen makuun”: koko instanssi voidaan konfiguroida myös käyttämään implisiittisiä transaktioita, tai yksittäinen SQL-istunto voidaan konfiguroida käyttämään implisiittisiä transaktioita asetuksella

```
SET IMPLICIT_TRANSACTIONS ON;
```

mikä on voimassa istunnon loppuun asti ellei sitten palata AUTOCOMMIT-moodiin asetuksella

```
SET IMPLICIT_TRANSACTIONS OFF;
```

Kokeilemme aluksi yksittäisiä transaktioita ja katsomme samalla tulokset:

```
-----------------------------------------------------------------------
-- Harjoitus 1.1
-----------------------------------------------------------------------
--  Autocommit mode
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(30), si SMALLINT);
Command(s) completed successfully.

INSERT INTO T (id, s) VALUES (1, 'first');
(1 row(s) affected)

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL

(1 row(s) affected)


ROLLBACK;
Msg 3903, Level 16, State 1, Line 3
The ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION.
```

```
SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL

(1 row(s) affected)


BEGIN TRANSACTION;  -- An explicit transaction begins
INSERT INTO T (id, s) VALUES (2, 'second');
SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
2           second                         NULL
(2 row(s) affected)

ROLLBACK;

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
(1 row(s) affected)
```

ROLLBACK-komento AUTOCOMMIT-moodissa tuottaa siis virheilmoituksen, mutta toimii jos transaktio on ollut käynnissä.

```
------------------------------------------------------------------------
-- Harjoitus 1.2
------------------------------------------------------------------------
INSERT INTO T (id, s) VALUES (3, 'third');
(1 row(s) affected)

ROLLBACK;
Msg 3903, Level 16, State 1, Line 3
The ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION.

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
2           third                          NULL
(2 row(s) affected)

COMMIT;
Msg 3902, Level 16, State 1, Line 2
The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION.



------------------------------------------------------------------------
-- Harjoitus 1.3
------------------------------------------------------------------------
BEGIN TRANSACTION;

DELETE FROM T WHERE id > 1;
(1 row(s) affected)

COMMIT;
```

```
SELECT * FROM T;
id          s                            si
----------- ---------------------------- ------
1           first                        NULL
(1 row(s) affected)


------------------------------------------------------------------
-- Harjoitus 1.4
-- DDL tulee sanoista Data Definition Language. SQL:ssä komennot
-- CREATE, ALTER ja DROP kuuluvat DDL-lauseiden joukkoon.
-- Testaamme seuraavaki voiko DDL-komento sisältyä T-SQL transaktioon?
------------------------------------------------------------------
SET IMPLICIT_TRANSACTIONS ON;
INSERT INTO T (id, s) VALUES (2, 'will this be COMMITTED?');
CREATE TABLE T2 (id INT);
INSERT INTO T2 (id) VALUES (1);
SELECT * FROM T2;
ROLLBACK;
GO  -- GO lähettää T-SQL:ssä komentoerän (batch) serverin suoritettavaksi
(1 row(s) affected)

(1 row(s) affected)

id
-----------
1
(1 row(s) affected)

SELECT * FROM T;    -- Mitä tapahtui taululle T ?
id          s                            si
----------- ---------------------------- ------
1           first                        NULL
(1 row(s) affected)

SELECT * FROM T2;   -- Mitä tapahtui taululle T2 ?
Msg 208, Level 16, State 1, Line 2
Invalid object name 'T2'.



------------------------------------------------------------------
-- Harjoitus 1.5a
------------------------------------------------------------------
DELETE FROM T WHERE id > 1;
COMMIT;
------------------------------------------------------------------
-- Seuraavaksi testaamme kaataako SQL-virhe koko T-SQL -transaktion
--   @@ERROR on T-SQL:n SQLCode-indikaattori
--   @@ROWCOUNT indikaattori kertoo käsiteltyjen rivien määrän
------------------------------------------------------------------
INSERT INTO T (id, s) VALUES (2, 'The test starts by this');
(1 row(s) affected)

SELECT 1/0 AS dummy;   -- Nollalla jako on yleensä virhe!
dummy
-----------
Msg 8134, Level 16, State 1, Line 1
Divide by zero error encountered.
```

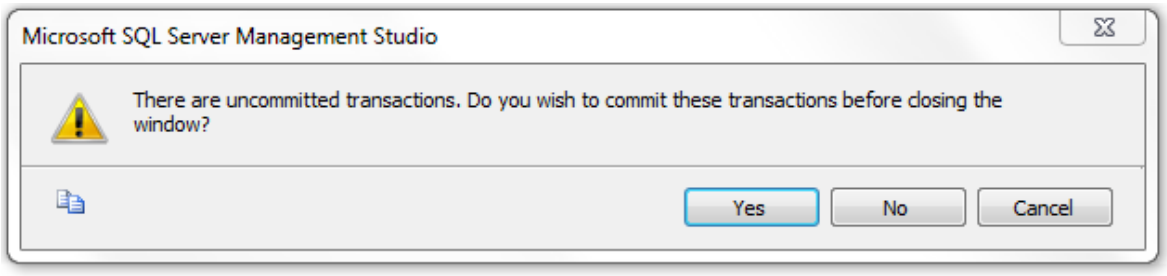
```
SELECT @@ERROR AS 'sqlcode'
sqlcode
-----------
8134
(1 row(s) affected)

UPDATE T SET s = 'foo' WHERE id = 9999;  -- Olemattoman rivin päivitys
(0 row(s) affected)

SELECT @@ROWCOUNT AS 'Updated'
Updated
-----------
0
(1 row(s) affected)

DELETE FROM T WHERE id = 7777;   -- Olemattoman rivin poisto
(0 row(s) affected)

SELECT @@ROWCOUNT AS 'Deleted'
Deleted
-----------
0
(1 row(s) affected)

COMMIT;

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
2           The test starts by this        NULL
(2 row(s) affected)

INSERT INTO T (id, s) VALUES (2, 'Hi, I am a duplicate')
INSERT INTO T (id, s) VALUES (3, 'How about inserting too long string value?')
INSERT INTO T (id, s, si) VALUES (4, 'Smallint overflow for 32769?', 32769);
INSERT INTO T (id, s) VALUES (5, 'Is the transaction still active?');
SELECT * FROM T;
COMMIT;
GO
Msg 2627, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'PK__T__3213E83FD0A494FC'. Cannot insert
duplicate key in object 'dbo.T'. The duplicate key value is (2).
The statement has been terminated.
Msg 8152, Level 16, State 14, Line 2
String or binary data would be truncated.
The statement has been terminated.
Msg 220, Level 16, State 1, Line 3
Arithmetic overflow error for data type smallint, value = 32769.
The statement has been terminated.
Msg 8152, Level 16, State 14, Line 4
String or binary data would be truncated.
The statement has been terminated.
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
2           The test starts by this        NULL
(2 row(s) affected)
```

```
BEGIN TRANSACTION;
SELECT * FROM T;
DELETE FROM T WHERE id > 1;
COMMIT;


-----------------------------------------------------------------
-- Harjoitus 1.5b
-- Huom:  Tämä toimii vain SQL Serverissä!
-----------------------------------------------------------------
SET XACT_ABORT ON;  -- Nyt SQL-virhe laukaisee automaattisen rollback'in!
SET IMPLICIT_TRANSACTIONS ON;

SELECT 1/0 AS dummy;   -- Division by zero
INSERT INTO T (id, s) VALUES (6, 'insert after arithm. error');
COMMIT;
SELECT @@TRANCOUNT AS 'do we have an transaction?'
GO

dummy
-----------
Msg 8134, Level 16, State 1, Line 3
Divide by zero error encountered.

SET XACT_ABORT OFF; -- Nyt SQL-virhe ei laukaise automaattista rollback'iä!
SELECT * FROM T;
id          s                           si
----------- --------------------------- ------
1           first                       NULL
2           The test starts by this     NULL
(2 row(s) affected)

-- What happened to the transaction?


-----------------------------------------------------------------
-- Harjoitus 1.6   Transaktiologiikan kokeiluja
-----------------------------------------------------------------
SET NOCOUNT ON;  -- tällä estetään "n row(s) affected" -viestit
DROP TABLE Accounts;
SET IMPLICIT_TRANSACTIONS ON;

CREATE TABLE Accounts (
acctID  INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL
        CONSTRAINT unloanable_account CHECK (balance >= 0)
);

COMMIT;

INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000

COMMIT;
```

```
-- Let's try the bank transfer
UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;
SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         900
202         2100

ROLLBACK;

-- Kokeillaan CHECK-rajoitteen toimivuutta:
UPDATE Accounts SET balance = balance - 2000 WHERE acctID = 101;
Msg 547, Level 16, State 0, Line 2
The UPDATE statement conflicted with the CHECK constraint
"unloanable_account". The conflict occurred in database "Testdb", table
"dbo.Accounts", column 'balance'.
The statement has been terminated.

UPDATE Accounts SET balance = balance + 2000 WHERE acctID = 202;
SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         4000

ROLLBACK;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000

UPDATE Accounts SET balance = balance - 2000 WHERE acctID = 101;
Msg 547, Level 16, State 0, Line 4
The UPDATE statement conflicted with the CHECK constraint
"unloanable_account". The conflict occurred in database "Testdb", table
"dbo.Accounts", column 'balance'.
The statement has been terminated.

-- Transaktiologiikka voidaan rakentaa myös T-SQL:n IF-rakenteilla
IF @@error <> 0 OR @@rowcount = 0
   ROLLBACK
ELSE BEGIN
   UPDATE Accounts SET balance = balance + 2000 WHERE acctID = 202;
   IF @@error <> 0 OR @@rowcount = 0
      ROLLBACK
   ELSE
      COMMIT;
   END;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000
COMMIT;
```

```
-- Olemattoman tilin päivitys:
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 777;

SELECT * FROM Accounts ;
acctID      balance
----------- -----------
101         500
202         2000

ROLLBACK;

SELECT * FROM Accounts ;
acctID      balance
----------- -----------
101         1000
202         2000

-- Korjataan logiikkaa IF-rakenteella

UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
IF @@error <> 0 OR @@rowcount = 0
   ROLLBACK
ELSE BEGIN
   UPDATE Accounts SET balance = balance + 500 WHERE acctID = 707;
   IF @@error <> 0 OR @@rowcount = 0
      ROLLBACK
   ELSE
      COMMIT;
   END;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000

ROLLBACK;

-----------------------------------------------------------------------
-- Harjoitus 1.7   Testataan commitoimattoman transaktion kohtaloa
--                 tietokantayhteyden katkettua
-----------------------------------------------------------------------
DELETE FROM T WHERE id > 1;
COMMIT;

BEGIN TRANSACTION;
INSERT INTO T (id, s) VALUES (9, 'What happens if ..');

SELECT * FROM T;
id          s                           si
----------- --------------------------- ------
1           first                       NULL
9           What happens if ..          NULL
```

Katkaistaan SQL Server Management Studion istunto, mistä saadaan seuraava ilmoitus



ja vastataan tähän painamalla "No"-valintaa.

Kun käynnistetään Management Studio uudelleen ja avataan yhteys Testdb-tietokantaan voimme tutkia löytyykö lisäämämme rivi 9 edelleen taulusta T:

```
SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
```

Rivi 9 puuttuu ja transaktio on siis peruuntunut automaattisesti.

Osa 2. Kokeilut samanaikaisilla transaktioilla

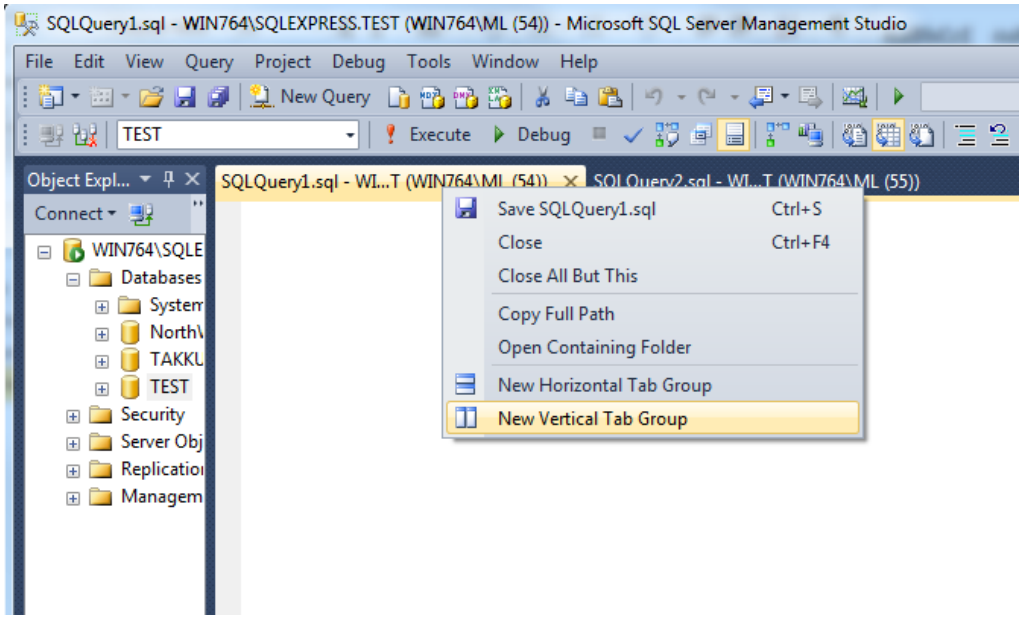
Samanaikaisuuskokeiluja varten avaamme Management Studiossa kaksi samanaikaista istuntoa Testdb-kntaamme. Kutsumme toista nimellä "client A" ja toista nimellä "client B". Tulosten kopiointia varten tekstitiedotoihin teemme molempiin istuntoihin seuraavat valinnat

Query > Results To > Results to Text

ja asetamme istunnot käytttämään implisiittisiä transaktioita

```
SET IMPLICIT_TRANSACTIONS ON;
```

Lisäksi asettelemme istuntojen ikkunat näkymään rinnakkain Management Studiossa painamalla ensin hiiren vaihtoehtoiselle painikkeella toisen ikkunan SQLQuery-otsaketta ja valiten hiirellä vaihtoehdon "New Vertical Tab Group" (katso Kuva 1.1).



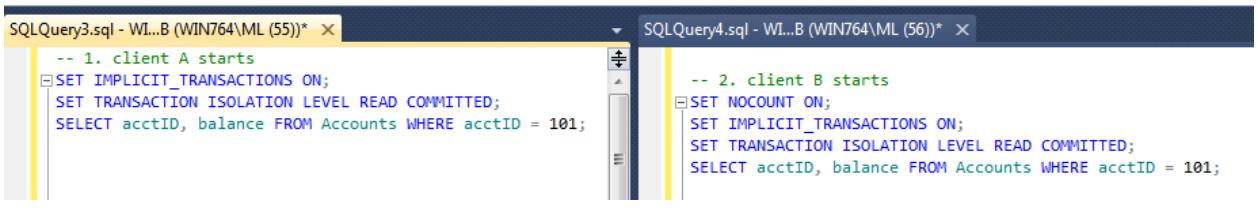
Kuva A1.1 Asetetaan SQLQuery-ikkunat näkymään rinnakkain

```
-------------------------------------------------------------
-- Harjoitus 2.1    "Hukatun päivityksen simulointia "
-------------------------------------------------------------
-- 0. Taulun alustus kokeilua varten
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;
```

Hukatun päivityksen ongelma syntyy jos jotakin lisättyä tai päivitettyä riviä päivitetään tai poistetaan ennen transaktion päättymistä jonkun kilpailevan transaction toimesta. Tämä voi onnistua tiedostopohjaisissa "NoSQL"-ratkaisuissa, mutta nykyaikaiset DBMS-järjesetelmät estävät tämän transaktion loppuun asti pidetyillä lukoilla. Kuitenkin, kun transaktio on commitoitu, kilpailevat transaktiot voivat päivittää tai poistaa rivin. Jos ne päivittävät rivin katsomatta kannassa olevan rivin tilaa, ne syyllistyvät sokeaan päällekirjoitukseen.

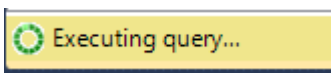
Seuraavassa simuloimme hukattua päivitystä käyttäen READ COMMITTED eristyvyyttä, mikä lukitsee luettavat tiedot vain lukemisen ajaksi.

```
-- 1. Client A starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
acctID      balance
----------- -----------
101         1000

-- 2. Client B starts
SET NOCOUNT ON;
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
```



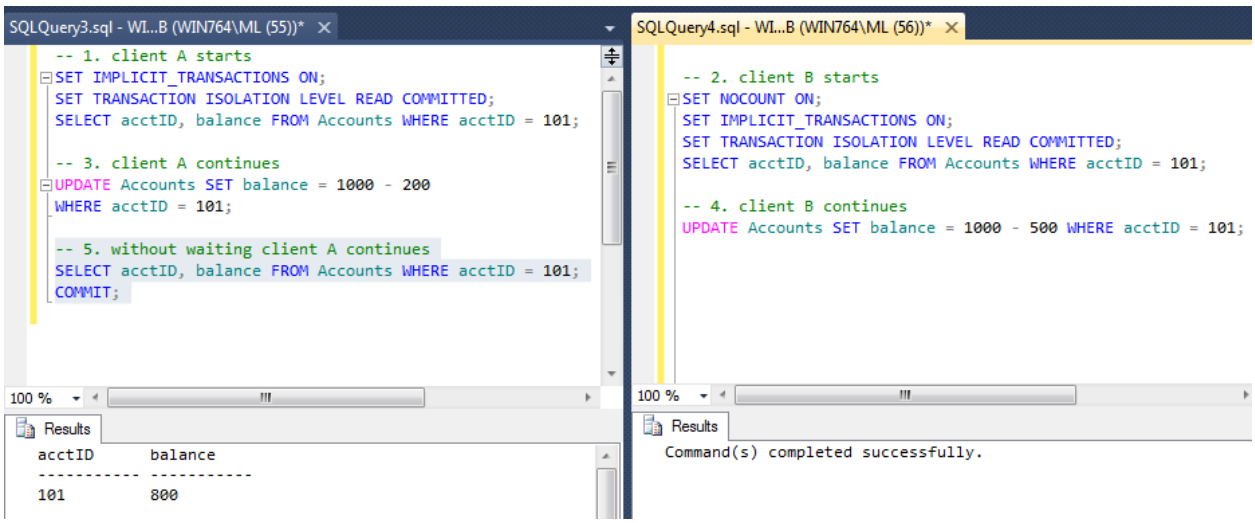
Kuva A1.2 Kilpailevat SQL-istunnot terminaali-ikkunoissaan

```
-- 3. Client A continues
UPDATE Accounts SET balance = 1000 - 200 WHERE acctID = 101;

-- 4. Client B continues
UPDATE Accounts SET balance = 1000 - 500 WHERE acctID = 101;
```



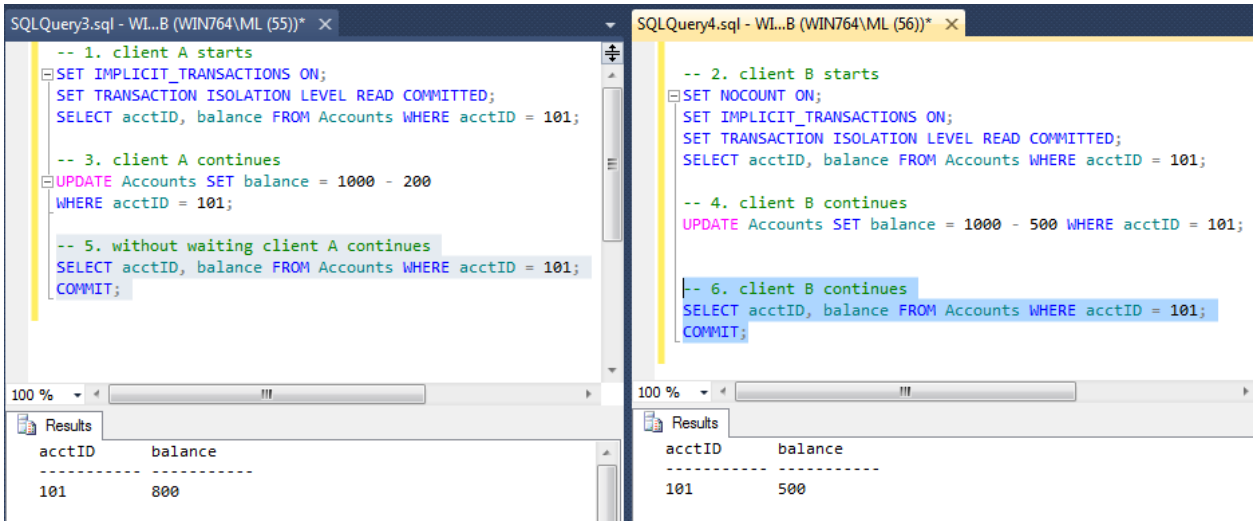
... waiting for A ...

```
-- 5. Client A continues
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
COMMIT;
```



Kuva A1.3 Tilanne vaiheessa 5

```
-- 6. Client B continues
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
COMMIT;
```



Kuva A1.4 Tilanne vaiheessa 6

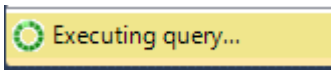
Lopulta tilanne on siis virheellinen!

Huom:

Tarkasti ottaen kyseessä ei ollut “hukatun päivityksen ongelma", mutta A:n committoitua B voi jatkaa ja päällekirjoittaen hukata A:n tekemän päivityksen. Kutsumme tätä ongelmaa ”sokeaksi päällekirjoitukseksi” tai ”likaiseksi kirjoitukseksi” ("**Dirty Write**"). Tämä voidaan välttää UPDATE –komennon **sensitiivisellä** muodolla, jossa komento lukee nykyarvon tietokannasta

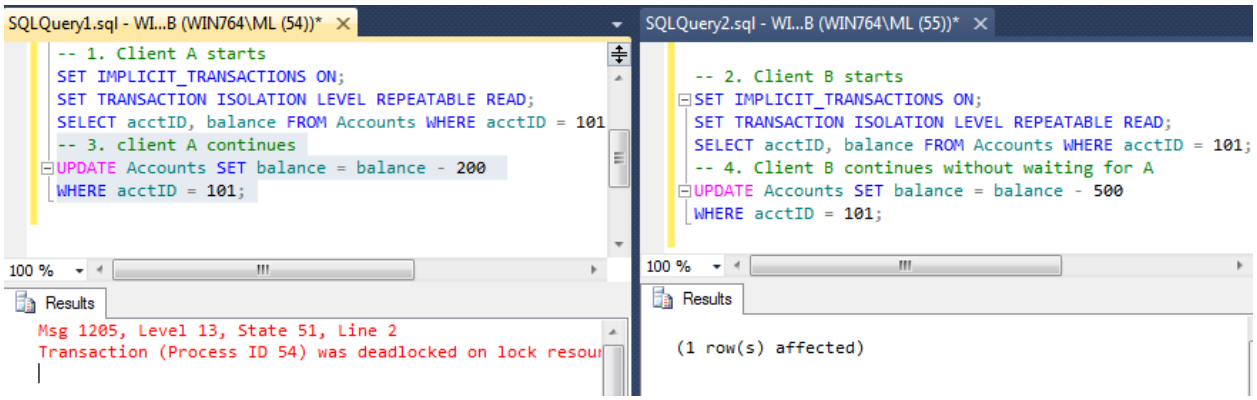
```
SET balance = balance - 500
```

```
-----------------------------------------------------------------
-- Harjoitus 2.2   "Hukatun päivityksen" ratkaisu lukoilla
-----------------------------------------------------------------
--    Kilpailu samasta resurssista SELECT .. UPDATE -skenaariolla
--    jossa sekä A ja B yrittävät nostaa summansa samalta tililtä
--
-- 0. Taulun alustus kokeilua varten
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;

-- 1. Client A starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
acctID      balance
----------- -----------
101         1000

-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
acctID      balance
----------- -----------
101         1000

-- 3. client A continues
UPDATE Accounts SET balance = balance - 200
WHERE acctID = 101;
```



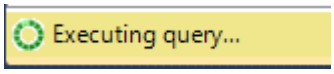
... waiting for B ...

```
-- 4. Client B continues without waiting for A
UPDATE Accounts SET balance = balance - 500
WHERE acctID = 101;
```



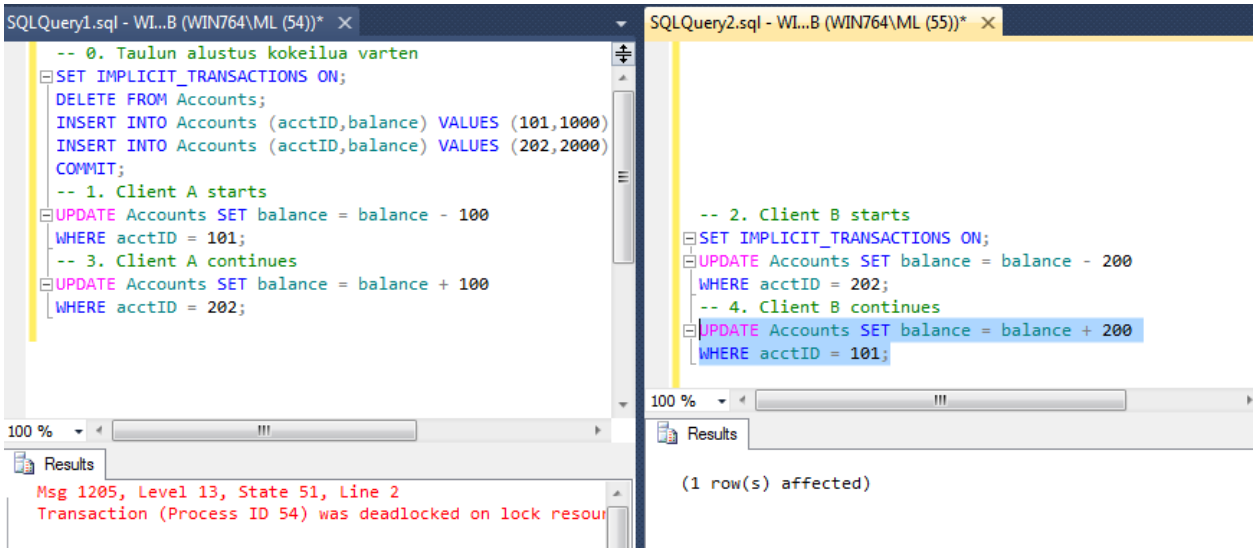

```
Msg 1205, Level 13, State 51, Line 2
Transaction (Process ID 54) was deadlocked on lock resources with another process and has been
chosen as the deadlock victim. Rerun the transaction.

-- 5. The client which survived will commit
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
acctID      balance
----------- -----------
101         500

COMMIT;
```

```
-----------------------------------------------------------------
-- Harjoitus 2.3  UPDATE – UPDATE -kilpailu kahdesta resurssista
--                eri järjestyksessä
-----------------------------------------------------------------
-- A siirtää 100 euroa tililtä 101 tilille 202
-- B siirtää 200 euroa tililtä 202 tilille 101
--
-- 0. Taulun alustus kokeilua varten
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;

-- 1. Client A starts
UPDATE Accounts SET balance = balance - 100
WHERE acctID = 101;

-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
UPDATE Accounts SET balance = balance - 200
WHERE acctID = 202;

-- 3. Client A continues
UPDATE Accounts SET balance = balance + 100
WHERE acctID = 202;
```



... waiting for B ...

```
-- 4. Client B continues
UPDATE Accounts SET balance = balance + 200
WHERE acctID = 101;
```



Harjoituksissa 2.4 – 2.7 kokeilemme ISO SQL –standardin kuvaamia samanaikaisuusongelmia. Tunnistammeko tapaukset? Entä miten ne on korjattavissa?

```
-----------------------------------------------------------------
-- Harjoitus 2.4   Suojaamaton eli likainen luku (Dirty Read)?
-----------------------------------------------------------------
--
-- 0. Taulun alustus kokeilua varten
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;


-- 1. Client A starts
SET IMPLICIT_TRANSACTIONS ON;
UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;


-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM Accounts;

acctID      balance
----------- -----------
101         900
202         2100
COMMIT;


-- 3. Client A continues
ROLLBACK;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000

COMMIT;
```

```
-----------------------------------------------------------------
-- Harjoitus 2.5   Ei-toistettavissa oleva luku (Non-repeatable Read)?
-----------------------------------------------------------------

-- 0. Taulun alustus kokeilua varten
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;

-- 1. Client A starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

--  Listing accounts having balance > 500 euros
SELECT * FROM Accounts WHERE balance > 500;
acctID      balance
----------- -----------
101         1000
202         2000

-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 202;
COMMIT;

-- 3. Client A continues
-- Can we still see the same accounts as in step 1?
SELECT * FROM Accounts WHERE balance > 500;
acctID      balance
----------- -----------
202         2500

COMMIT;
```

```
-----------------------------------------------------------------
-- Harjoitus 2.6   Insert-Phantom?
-----------------------------------------------------------------
-- Insert-phantom rivillä tarkoitetaan riviä, jonka joku toinen
-- transaktio lisää transaktion aikana ja jota transaktio ei aluksi
-- ole huomannut, mutta voisi huomata vielä transaktion aikana.
--
-- 0. Taulun alustus kokeilua varten
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
DROP TABLE T2;
CREATE TABLE T2 (id INT);  -- transaction alkamisen testaamiseksi
COMMIT;

-- 1. Client A starts
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET NOCOUNT ON

-- T-SQL:n funktio, joka laskee sisäkkäisistä transaktioista transaktion tason:
SELECT @@trancount transaktiota
SELECT COUNT(*) dummy FROM T2;
SELECT @@trancount transaktiota

transaktiota
------------
0

dummy
-----------
0

transaktiota
------------
1
```

Huomaamme, että A:n transaktio alkoi vasta oikeaan tauluun tehdystä kyselystä!

```
-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
INSERT INTO Accounts (acctID, balance) VALUES (301,3000);
COMMIT;

-- 3. Client A continues
-- Accounts having balance > 1000 euros
SELECT * FROM Accounts WHERE balance > 1000;
acctID      balance
----------- -----------
202         2000
301         3000

-- 4. Client B continues
INSERT INTO Accounts (acctID,balance) VALUES (302,3000);
COMMIT;

-- 5. Client A continues
-- Let's see the results
SELECT * FROM Accounts WHERE balance > 1000;
acctID      balance
----------- -----------
202         2000
301         3000
```

```
302         3000

COMMIT;
```

Kysymys

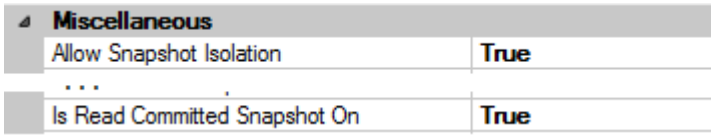
- Miten voisimme tässä estää phantomin?

```
-------------------------------------------------------------
-- SNAPSHOT-kokeiluja
-------------------------------------------------------------
```

SNAPSHOT-eristyvyystaso ei ole oletuksena käytettävissä SQL Server tietokannoissa, vaan tämä ominaisuus on konfiguroitava käyttöön tietokantakohtaisesti. SNAPSHOT-testejä varten luomme uuden tietokannnan

```
CREATE DATABASE SnapsDB;
```

ja konfiguroimme sen Management Studion avulla tukemaan rivien moniversiointia seuraavasti:



Vaihdamme sitten client A:n ja B:n SQLQuery-ikkunat käyttämään tätä SnapsDB-kantaa seuraavasti:

```
USE SnapsDB;
```

```
-----------------------------------------------------------------------
-- Harjoitus 2.7
-- SNAPSHOT-kokeilu erilaisten phantom ja ghoast –rivien tapauksessa
-----------------------------------------------------------------------
USE SnapsDB;

-- 0. Testiä varten vaihdetaan taulun T rakenne ja sisältö
DROP TABLE T;
GO

SET IMPLICIT_TRANSACTIONS ON;
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(30), i SMALLINT);
INSERT INTO T (id, s, i) VALUES (1, 'first',  1);
INSERT INTO T (id, s, i) VALUES (2, 'second', 2);
INSERT INTO T (id, s, i) VALUES (3, 'third', 1);
INSERT INTO T (id, s, i) VALUES (4, 'fourth', 2);
INSERT INTO T (id, s, i) VALUES (5, 'to be or not to be', 1);
COMMIT;

-- 1. Client A starts
USE SnapsDB;
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL SNAPSHOT ;
SELECT * FROM T WHERE i = 1;
id          s                              i
----------- ------------------------------ ------
1           first                          1
```

```
3           third                          1
5           to be or not to be             1
```

```
-- 2. Client B starts
USE SnapsDB;
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

INSERT INTO T (id, s, i) VALUES (6, 'Insert Phantom', 1);
UPDATE T SET s = 'Update Phantom', i = 1 WHERE id = 2;
DELETE FROM T WHERE id = 5;

SELECT * FROM T;
```

```
id          s                              i
----------- ------------------------------ ------
1           first                          1
2           Update Phantom                 1
3           third                          1
4           fourth                         2
6           Insert Phantom                 1
```

```
-- 3. Client A continues
-- Let's repeat the query and try some updates
SELECT * FROM T WHERE i = 1;
```

```
id          s                              i
----------- ------------------------------ ------
1           first                          1
3           third                          1
5           to be or not to be             1
```

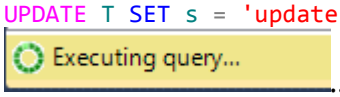
```
INSERT INTO T (id, s, i) VALUES (7, 'inserted by A', 1);
UPDATE T SET s = 'update by A inside snapshot' WHERE id = 3;
UPDATE T SET s = 'update by A outside snapshot' WHERE id = 4;
UPDATE T SET s = 'update by A after B' WHERE id = 1;

SELECT * FROM T WHERE i = 1;
```

```
id          s                              i
----------- ------------------------------ ------
1           update by A after B            1
3           update by A inside snapshot    1
5           to be or not to be             1
7           inserted by A                  1
```

```
-- 3.5 Client C in a new session starts and executes a query
USE SnapsDB;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM T;
```

```
id          s                              i
----------- ------------------------------ ------
1           update by A after B            1
2           Update Phantom                 1
3           update by A inside snapshot    1
4           update by A outside snapshot   2
6           Insert Phantom                 1
7           inserted by A                  1
```

```
-- 4. Client B continues
SELECT * FROM T;
id          s                             i
----------- ----------------------------- ------
1           first                         1
2           Update Phantom                1
3           third                         1
4           fourth                        2
6           Insert Phantom                1


-- 5. Client A continues
SELECT * FROM T WHERE i = 1;
id          s                             i
----------- ----------------------------- ------
1           update by A after B           1
3           update by A inside snapshot   1
5           to be or not to be            1
7           inserted by A                 1

UPDATE T SET s = 'update after delete?' WHERE id = 5;
```



... waiting for B ...

```
-- 6. Client B continues without waiting for A
COMMIT;


-- 7. Client A continues
Msg 3960, Level 16, State 2, Line 1
Snapshot isolation transaction aborted due to update conflict. You cannot use
snapshot isolation to access table 'dbo.T' directly or indirectly in database
'SnapsDB' to update, delete, or insert the row that has been modified or
deleted by another transaction. Retry the transaction or change the isolation
level for the update/delete statement.

-- 8. Client B continues
SELECT * FROM T;
id          s                             i
----------- ----------------------------- ------
1           first                         1
2           Update Phantom                1
3           third                         1
4           fourth                        2
6           Insert Phantom                1
```

Kysymys

- Selitä mitä tässä kokeilussa tapahtui. Miksi vaiheiden 3.5 ja 4 kyselyjen tulosjoukot ovat erilaisia?

--

Lisää mielenkiintoisia tietoja SQL Server'in transaktioista löytyy kirjasta
Kalen Delaney (2012), "SQL Server Concurrency, Locking, Blocking and Row Versioning"
ISBN 978-1-906434-90-8

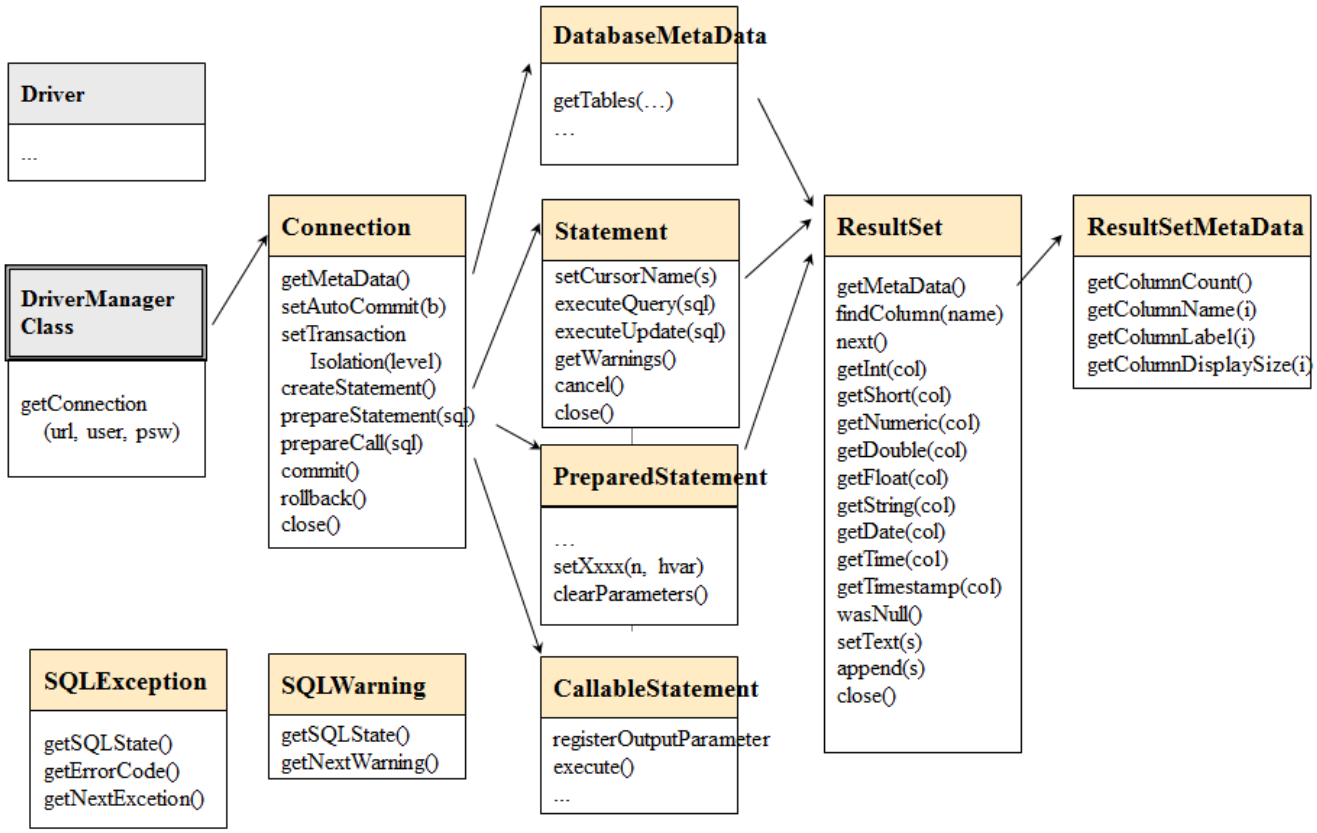
Liite 2. Java/JDBC -transaktio-ohjelmointia

Edellä olemme kokeilleet DBMS-järjestelmien transaktiopalveluita käyttäen järjestelmien vuorovaikutteisen SQL-murteita ns. SQL-editor-ohjelmilla. Kuitenkin sovellusohjelmien tarvitsemia tietokantakäsittelyjä ei tehdä pelkällä SQL-kielellä vaan käyttäen ao. ohjelmointikielelle sovitettua tietokantaliittymäkirjastoa (application programming interface, API). Matalimmalla API-tasolla tietokantaliittymät ovat DBMS:n funktiokirjastoja, joille on SQL-standardissa oma osansa: Call Level Interface eli SQL/CLI, ja esimerkiksi DB2:n nykyinen funktiokirjasto on jokseenkin SQL/CLI:n mukainen. SQL/CLI:ssä SQL-kielen lauseet välitetään palvelimelle kutsufunktioiden parametreina. SQL/CLI:tä vastaava de facto standardi on Microsoftin ODBC API, ”universaalia funktiokirjasto”, jonka kutsut sovitetaan DBMS-järjestelmäkohtaisella ajuri-ohjelmistolla (driver) järjestelmän natiiville funktiokirjastolle. Käytetty SQL-kieli ja myös transaktioprotokolla ovat SQL/CLI:ssä yleisempää ja ajuri voi tehdä joitakin sovituksia DBMS:n murteelle.

Java-kielelle SQL/CLI on sovitettu helppokäyttöisempänä JDBC API –luokkakirjastona. Oletamme, että Java-kielen perusteet ja mahdollisesti JDBC API ovat lukijalle ennestään jossakin määrin tuttuja. Tarvittaessa näistä molemmista löytyy runsaasti tutoriaaleja Internetistä, ja täydellisimmät kuvaukset kielen nykyisen kehittäjän Oraclen sivuilta

http://docs.oracle.com/javase/tutorial/

Kuva A2.1 esittää kaaviona JDBC-luokkien keskeiset metodit ja vuorovaikutussuhteet. Pienenä esimerkkinä SQL-transaktioiden toteutuksesta sovellusohjelmissa käytämme tilisiirtoharjoituksemme toteutusta listauksen A.1 Java/JDBC-ohjelmana.



Kuva A2.1 JDBC API:n luokat ja keskeiset metodit

BankTransfer.java

Tilisiirto-ohjelma BankTransfer siirtää 100 euroa yhdeltä pankkitililtä “fromAcct” toiselle tilille ”toAcct”. Ohjelma on parametroitu siten, että lähdeohjelmaa muuttamatta sillä voidaan käyttää yhtä hyvin Oracle, DB2, MySQL, PostgreSQL –tietokannan kanssa tietokantalaboratoriomme tai Windows-alustalla myös SQL Server –tietokannan kanssa. Tilitietojen lisäksi sille syötetään tietokantapalvelimen JDBC-ajurin nimi, tietokannan URL sekä käyttäjän nimi ja salasana ohjelman komentoriviparametreina ja komentoriville ne ladotaan ympäristömuuttujista.

Listaus A2.1 Tilisiirron Java/JDBC-ohjelma BankTransfer.java

```
/*  DBTechNet Concurrency Lab 15.5.2008 Martti Laiho
Save the java program as BankTransfer.java and compile as follows
  javac BankTransfer.java
See BankTransferScript.txt for the test scripts applied to SQL Server, Oracle and DB2
Updates:
2.0 2008-05-26 ML preventing rollback by application after SQL Server deadlock
2.1 2012-09-24 ML restructured for presenting the Retry Wrapper block
2.2 2012-11-04 ML exception on non-existing accounts
2.3 2014-03-09 ML TransferTransaction returns 1 for retry, 0 for OK, < 0 for error
**************************************************************/
import java.io.*;
import java.sql.*;
public class BankTransfer {
   public static void main (String args[]) throws Exception
   {
      System.out.println("BankTransfer version 2.3");
      if (args.length != 6) {
         System.out.println("Usage:" +
         "BankTransfer %driver% %URL% %user% %password% %fromAcct% %toAcct%");
         System.exit(-1);
      }
      java.sql.Connection conn = null;
      boolean sqlServer = false;
      int counter = 0;
      int retry = 0;
      String URL = args[1];
      String user = args[2];
      String password = args[3];
      int amount = 100;
      int fromAcct = Integer.parseInt(args[4]);
      int toAcct = Integer.parseInt(args[5]);
      // SQL Server's explicit transactions will require special treatment
      if (URL.substring(5,14).equals("sqlserver")) {
        sqlServer = true;
      }
            // register the JDBC driver and open connection
      try {
         Class.forName(args[0]);
         conn = java.sql.DriverManager.getConnection(URL,user,password);
      }
```

```
    catch (SQLException ex) {
        System.out.println("URL: " + URL);
        System.out.println("** Connection failure: " + ex.getMessage() +
                           "\n SQLSTATE: " + ex.getSQLState() +
                           " SQLcode: " + ex.getErrorCode());
        System.exit(-1);
    }
    do {
    // Retry wrapper block of TransaferTransaction ----------
       if (counter++ > 0) {
          System.out.println("retry #" + counter);
          if (sqlServer) {
            conn.close();
            System.out.println("Connection closed");
            conn = java.sql.DriverManager.getConnection(URL,user,password);
            conn.setAutoCommit(true);
          }
       }
       retry = TransferTransaction (conn, fromAcct, toAcct, amount, sqlServer);
       if (retry == 1) {
          long pause = (long) (Math.random () * 1000); // max 1 sec.
          System.out.println("Waiting for "+pause+ " mseconds before retry"); // just for testing
          Thread.sleep(pause);
       } else
       if (retry < 0) System.out.println (" Error code: " + retry + ", cannot re-try.");
    } while (retry == 1 && counter < 10);  // max 10 retries
    // end of the Retry wrapper block -------------------------------

    conn.close();
    System.out.println("\n End of Program. ");
  }

  static int TransferTransaction (Connection conn,
      int fromAcct, int toAcct, int amount,
      boolean sqlServer
      )
      throws Exception {
      String SQLState = "*****";
      String errMsg = "";
      int retry =  0;
      try {
          conn.setAutoCommit(false);   // transaction begins
          conn.setTransactionIsolation(
              Connection.TRANSACTION_SERIALIZABLE);
          errMsg = "";
          retry = 0; //"N";
          // a parameterized SQL command
          PreparedStatement pstmt1 = conn.prepareStatement(
            "UPDATE Accounts SET balance = balance + ? WHERE acctID = ?");
          // setting the parameter values
          pstmt1.setInt(1, -amount);  // how much money to withdraw
          pstmt1.setInt(2, fromAcct); // from which account
          int count1 = pstmt1.executeUpdate();
```

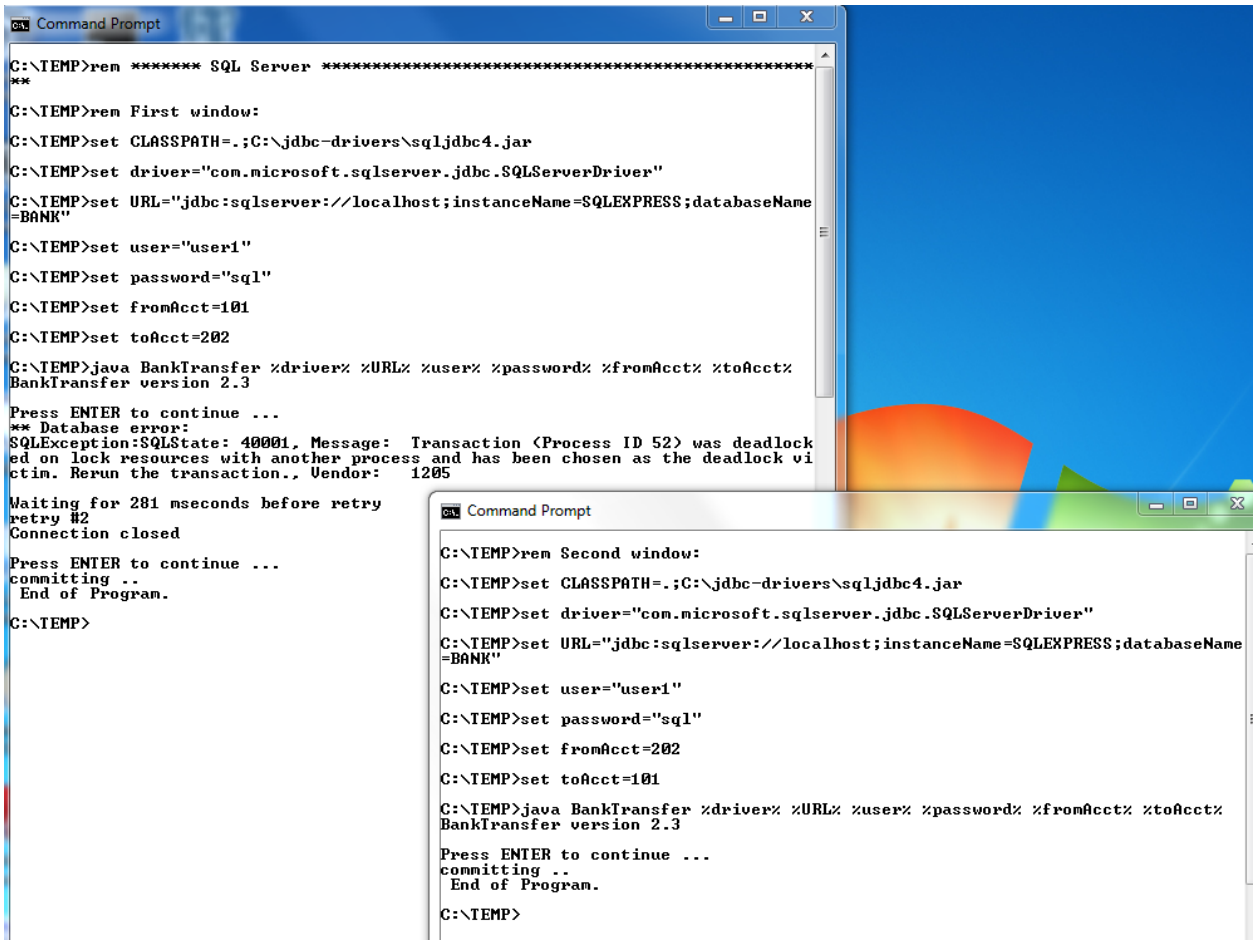
```
      if (count1 != 1) throw new Exception ("Account "+fromAcct + " is missing!");

      // ******  interactive pause just for concurrency testing ******
      // In the following we arrange the transaction to wait
      // until the user presses ENTER key so that another client
      // can proceed with a conflicting transaction.
      // This is just for concurrency testing, so don't apply this
      // user interaction in real applications!!!
        System.out.print("\nPress ENTER to continue ...");
        BufferedReader reader = new BufferedReader(
                new InputStreamReader(System.in));
        String s = reader.readLine();
      // ****** end of waiting ****************************

      pstmt1.setInt(1, amount);  // how much money to add
      pstmt1.setInt(2, toAcct);  // to which account
      int count2 = pstmt1.executeUpdate();
      if (count2 != 1) throw new Exception  ("Account "+toAcct + " is missing!");
      System.out.print("committing ..");
      conn.commit();  // end of transaction
      pstmt1.close();
    }
    catch (SQLException ex) {
      try {
        errMsg = "\nSQLException:";
        while (ex != null) {
            SQLState = ex.getSQLState();
            // is it a concurrency conflict?
            if ((SQLState.equals("40001")   // Solid, DB2, SQL Server,...
              || SQLState.equals("61000")   // Oracle ORA-00060: deadlock detected
              || SQLState.equals("72000"))) // Oracle ORA-08177: can't serialize access
                retry = 1; //"Y";
            errMsg = errMsg + "SQLState: "+ SQLState;
            errMsg = errMsg + ", Message:  "+ ex.getMessage();
            errMsg = errMsg + ", Vendor:   "+ ex.getErrorCode() + "\n";
            ex = ex.getNextException();
        }
        // SQL Server does not allow rollback after deadlock !
        if (sqlServer == false) {
           conn.rollback();  // explicit rollback needed for Oracle
                        // and the extra rollback does not harm DB2
        }
        // println for testing purposes
        System.out.println("** Database error: " + errMsg);
      }
      catch (Exception e) { // In case of possible problems in SQLException handling
        System.out.println("SQLException handling error: " + e);
        conn.rollback();  // Current transaction is rolled back
        retry = -1; // This is reserved for potential exception handling
      }
    } // SQLException
    catch (Exception e) {
      System.out.println("Some java error: " + e);
```

```
            conn.rollback();  // Current transaction is rolled back also in this case
            retry = -1; // This is reserved for potential other exception handling
         } // other exceptions
         finally { return retry; }
   }
}
```

Ohjelma ei pyri opettamaan Java-ohjelmointia, mutta siinä kannattaa kiinnittää huomiota

- Tietokantayhteyden connection-olion luontiin
- Retry Wrapper'in rakenteeseen ja deadlock-uhrin pause-ratkaisuun
- Transaktion aloitukseen
- Transaktion isolation level –asetukseen
- Parametroidun SQL-komennon suoritukseen PreparedStatement-oliolla ja parametriarvojen sidontaan
- SQLSTATE ja SQLCode-indikaattoreiden arvojen selvittämiseenl SQLException-olioiden metodeilla
- Samanaikaisuuskonfliktin erilaisiin SQLSTATE-arvoihin eri järjestelmissä.

Ohjelmaa on tarkoitus ajaa kahdessa rinnakkaisessa terminal-ikkunassa samaa tietokantaa vasten siten että tilisiirrot näissä ajoissa tehdään vastakkaisilla UPDATE–UPDATE skenaarioilla ja ohjelmaan koodatulla ylimääräisellä käyttäjän kontrollidialogilla (näkyy listauksen jälkimmäisessä laatikossa) ajot synkronoidaan siten, että voidaan testata deadlock-tilannetta. Tästä toivutaan transaktiolle ohjelmoidulla ReTry-patternilla, mikä on rajattu listauksessa ensimmäiseen laatikkoon.



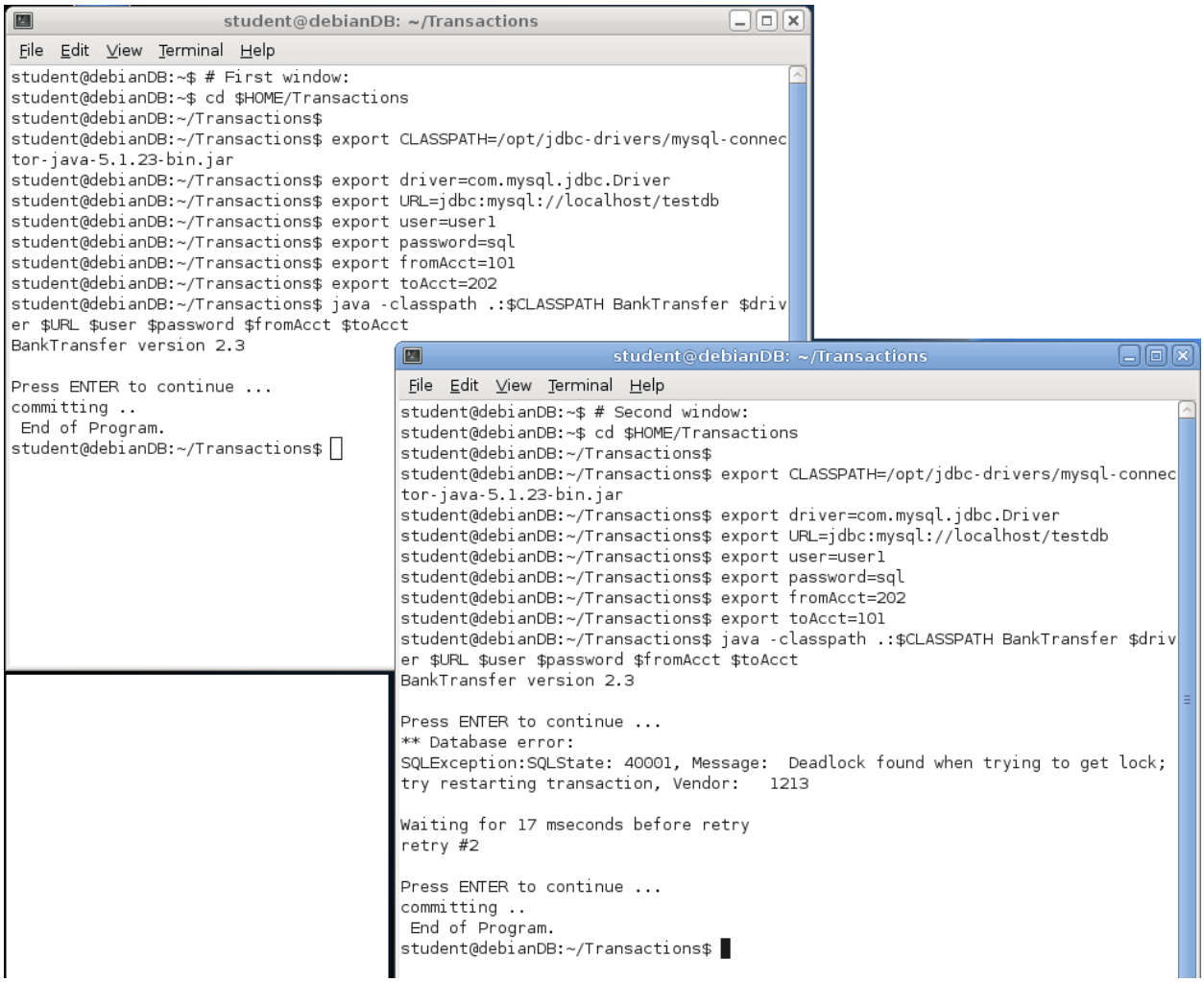
Kuva A2.2 BankTransfer-testi Windows-ympäristössä

Kuvan A2.2 testi kahdessa kilpailevasssa Command Prompt –ikkunassa on tehty Windows-alustalla SQL Server Express –järjestelmällä. SQL Serverin JDBC-ajuri sqljdbc4.jar on tässä asennettu hakemistoon C:\jdbc-drivers. Vasemmanpuoleinen client siirtää 100 euroa tililtä 101 tilile 202 ja oikeanpuoleinen client tililtä 202 tilille 101. Nostettuaan 100 euroa ensimmäiseltä tililtä molemmat ajot pysähtyvät odottamaan käyttäjältä jatkolupaa. Käyttäjän annettua ENTER-painalluksella molemmille jatkoluvan ajot törmäävät deadlock-tilanteeseen ja vasemmanpuoleinen joutuu deadlockin uhrina peruutetuksi. Oikeanpuoleinen pääsee jatkamaan ja saa transaktion committoitua. ajettavalle testille ja kuva A2.2 esittää testiajon tulokset kilpailevissa ikkunoissa.

Linux-asennuksissa ei ole sovittu yleistä ratkaisua JDBC-ajurien paikalle, joten olemme ratkaisseet asian siten, tarvittavat JDBC-ajurit on koottu tietokantalaboratoriossamme hakemistoon /opt/jdbc-drivers. Listaus A2.2 esittää ensin miten tämä JDBC-ajurien hakemisto on asennettu ja sitten testiajojen scriptit Linux-alustalla sovitettuna MySQL-järjestelmälle. Script-tiedostot ja BankTransfer-ohjelman lähdekoodi ja valmiiksi käännetty ajoluokka on talletettu student-käyttäjän hakemistoon Transactions.

Listaus A2.2 Scripts for MySQL on Linux:

```
# Creating directory /opt/jdbc-drivers for JDBC drivers
cd /opt
mkdir jdbc-drivers
chmod +r+r+r jdbc-drivers
# copying the MySQL jdbc driver to /opt/jdb-drivers
cd /opt/jdbc-drivers
cp $HOME/mysql-connector-java-5.1.23-bin.jar
# allow read access to the driver to everyone
chmod +r+r+r mysql-connector-java-5.1.23-bin.jar


#******* BankTransfer using MySQL/InnoDB **********************************

# First window:
cd $HOME/Transactions
export CLASSPATH=/opt/jdbc-drivers/mysql-connector-java-5.1.23-bin.jar
export driver=com.mysql.jdbc.Driver
export URL=jdbc:mysql://localhost/Testdb
export user=user1
export password=sql
export fromAcct=101
export toAcct=202
java -classpath .:$CLASSPATH BankTransfer $driver $URL $user $password
$fromAcct $toAcct

# Second window:
cd $HOME/Transactions
export CLASSPATH=/opt/jdbc-drivers/mysql-connector-java-5.1.23-bin.jar
export driver=com.mysql.jdbc.Driver
export URL=jdbc:mysql://localhost/Testdb
export user=user1
export password=sql
export fromAcct=202
export toAcct=101
java -classpath .:$CLASSPATH BankTransfer $driver $URL $user $password
$fromAcct $toAcct
#*************************************************************************
```



Kuva A2.3 BankTransfer-testi tietokantalaboratorion Debian Linux -ympäristössä

Kuvan A2.3 testissä DebianDB-virtuaalilaboratoriossa on avattu kilpailevat Linux terminal ikkunat ja niissä BankTransfer-ajot ottavat tietokantyhteyden paikalliseen MySQL/InnoDB-tietokantaan testdb. Testi etenee vastaavasti kuin Windows-alustalla ajettu testi edellä. Tällä kertaa deadlock-tilanteen uhriksi joutuu oikeanpuoleinen client ja lyhyen taun jälkeen sen uusintayritys onnistuu, koska kilpailija on ehtinyt saada transaktionsa jo aikaisemmin committoitua.

Liite 3. Transaktio – toipumisen perusyksikkö (unit of recovery)

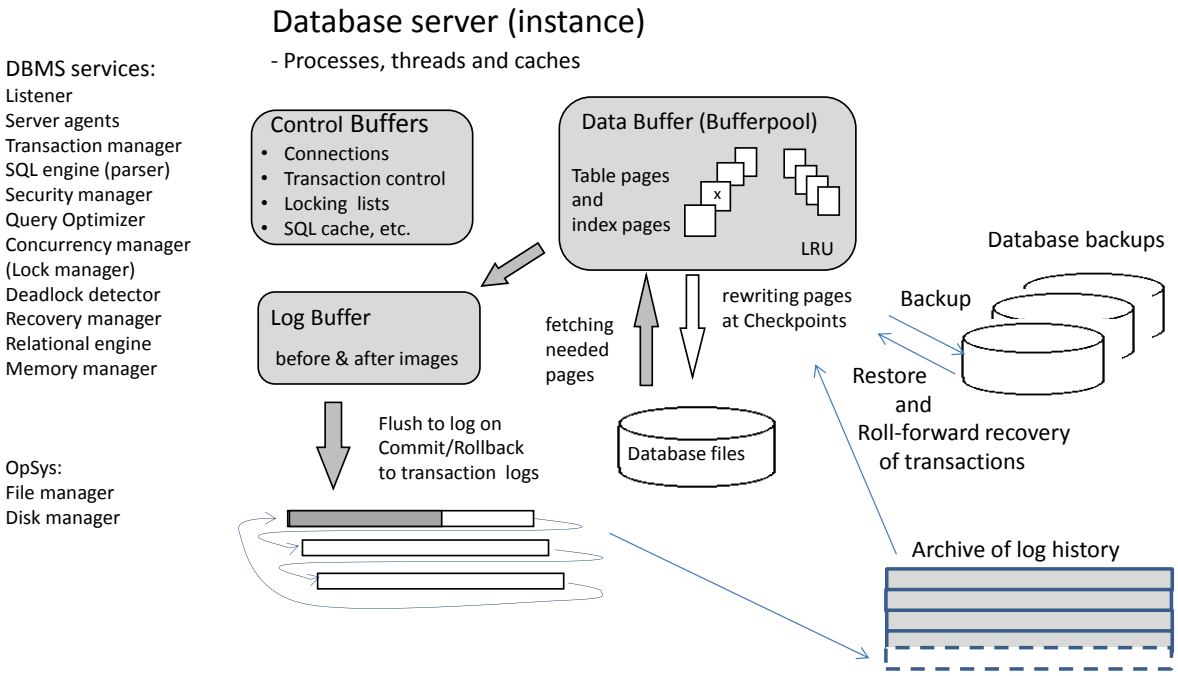
Transaktio on tietokannan tietosisällön eheyden ja virheistä toipumisen perusyksikkö (unit of recovery). Tämä koskee yksittäistä transaktiota, yksittäistä tietokantaa ja koko tietokantapalvelimen laitteistorikkoa.

Ongelmiin joutunut transaktio voidaan peruuttaa ROLLBACK-operaatiolla ja tietokantayhteyden menettänyt transaktio peruuntuu automaattisesti järjestelmän toimesta. COMMIT-operaatiollla valmiiksi kuitatut transaktiot puolestaan ovat oikein hoidetussa tietokannassa turvassa vaikka tietokantapalvelin tai sen fyysinen palvelin ”kaatuisivat”.

Tietokantapalvelimen toipuminen viimeiseen committoituun transaktioon asti, vain käynnistämällä palvelin uudelleen ”soft crash”-tilanteen jälkeen, esimerkiksi sähkökatkoksen tai palvelin/käyttöjärjestelmän kaatumisen, perustuu transaktiohistorian erilliseen talletukseen **transaktiolokiin** ja sen huolella suunniteltuun hoitoon.

Jopa fyysisen palvelimen tuhoutumisesta (”hard crash”) voidaan oikein hoidettujen tietokantavarmistusten (database backup) ja niiden jälkeisten täydellisen transaktiolokihistorian ulkoisen varmistus-arkistoinnin ja palautusohjelmiston avulla palauttaa tai luoda uudelleen jopa toiselle laitteistolle viimeisen commitoidun transaktion tasolle.

Kuva A3.1 esittää yleiskuvan tietokantapalvelimen osista ja toiminnoista transaktiokäsittelyn (transaction processing internals). Kuvaa täydentää seuraava kuvasarja
http://www.dbtechnet.org/papers/BasicsOfSqlTransactions.pdf
ja perusteellisemmin tietokantahoidon tutorialimme osoitteessa
http://www.dbtechnet.org/labs/dba_lab/DBALabs.pdf



Kuva A3.1 Yleiskuva tietokantapalvelimen transaktiokäsittelyn toiminnoista

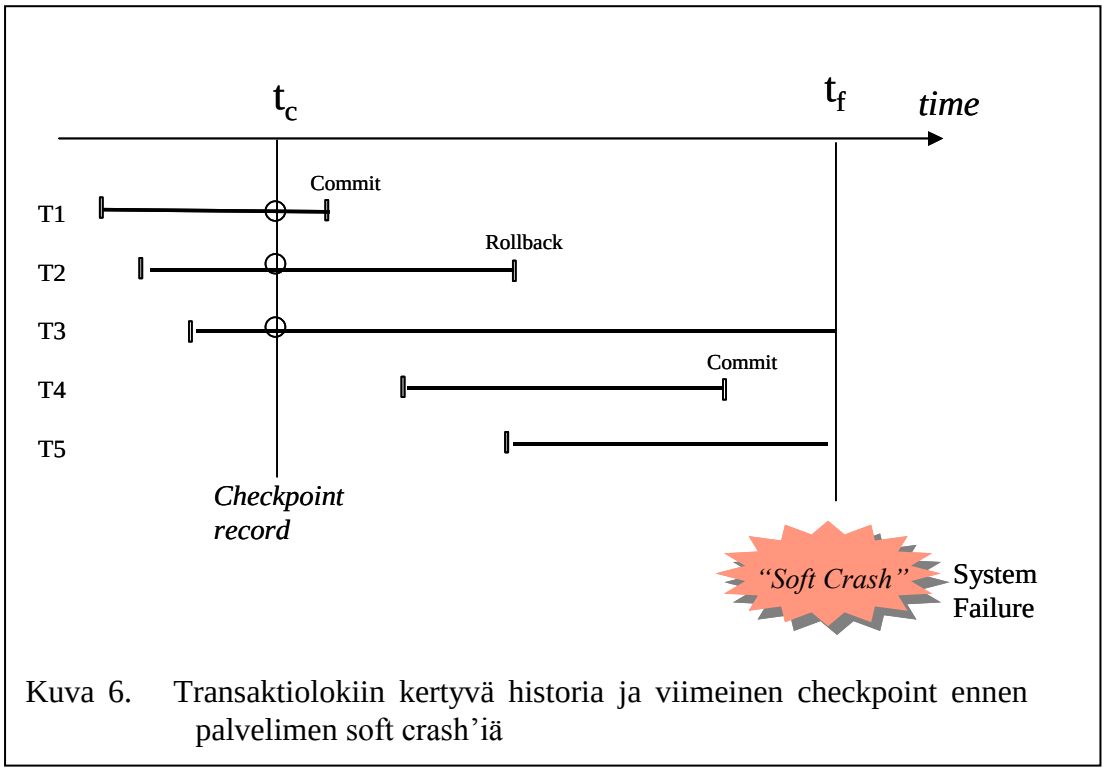
DBMS-järjestelmät ovat kehittyneimpiä ja monimutkaisimpia ohjelmistoja. Niiden suoritustehoa on opimoitu siten että käsittely tapahtuu mahdollisuuksien mukaan keskusmuistista varatuissa puskureissa, minimoiden hakuja ja kirjoitusta eli levyhakuja tietokantatiedostoihin.

Aktiivisimmat tietokannan osat pyritään säilyttämään palvelimen keskusmuistiin varatussa puskurialtaassa, jossa kaikki tietojen käsittely tapahtuu.
Tietokantapalvelin numeroi juoksevasti tietokannan alkaneet transaktiot ja kirjoittaa jokaisesta tietokannassa muutetusta rivistä asianomaisella transaktionumerolla merkityn tietueen **transaktiolokiin**, johon tulee rivin **alkukuva** (before image) ennen käsittelyä ja **loppukuva** (after image) käsittelyn jälkeen. INSERT-komennon tapauksessa alkukuva on tyhjä ja DELETE-komennon tapauksessa loppukuva on tyhjä. COMMIT ja ROLLBACK-komennoista kirjoitetaan lokiin myös tietueet ja transaktion lokitietueet kirjoitetaan puskureista transaktiolokiin viimeistään tässä vaiheessa. ROLLBACK-operaatiossa kaikki transaktion tekemät tietokantamuutokset peruutetaan palauttamalla transaktiolokin puskurista ao. rivien alkukuvat rivien sivuille puskurialtaassa.

Ajoittain DBMS tekee CHECKPOINT-operaation, jossa keskusmuistissa olevan puskurialtaan muuttuneet sivut kirjoitetaan tietokantatiedostoihin ja transaktiolokiin kirjoitetaan checkpoint-tietue (itseasiassa tietuerypäs), jossa on lista kaikista käynnissä olevien transaktioiden numeroista.

Huom: Jos tietokannanhoitaja sulkee tietokantainstanssin hallitusti eli ei salli kantaan uusia istuntoja ja odottaa että kaikki käynnissä olevat istunnot ovat päättyneet, kirjoittaa DBMS lopuksi transaktiolokiin checkpoint-tietueen, jossa on tyhjä transaktio-numerolista.

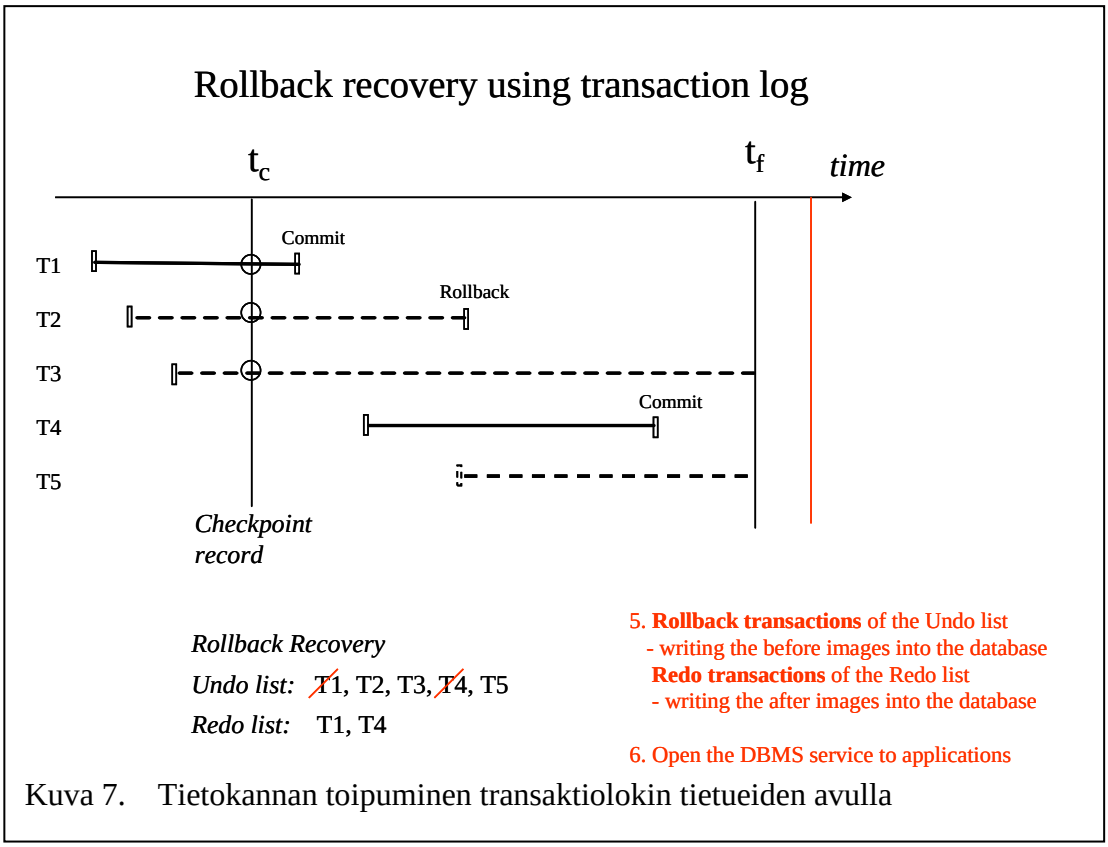
Tarkastelemme nyt tilannetta, jossa tietokantainstanssi tai koko palvelin kaatuu esimerkiksi sähkökatkoksen vuoksi. Tietokannan kannalta kaikki levyillä olevat tiedostot säilyvät, mutta kaikki puskurialtaan tiedot menetetään (ns. Soft Crash tilanne).



Kuva 6. Transaktiolokiin kertyvä historia ja viimeinen checkpoint ennen palvelimen soft crash'iä

Kun DBMS seuraavan kerran käynnistetään, se etsii ensitöikseen transaktiolokin viimeisen checkpoint-tietueen. Jos tämä osoittaa, että DBMS on suljettu hallitusti, käynnistetään sovellusten palvelu, muussa tapauksessa DBMS yrittää ensin toipua Soft Crash –tilanteesta Rollback Recovery –operaatiolla seuraavasti:

Checkpoint-tietueen transaktionumerot kirjataan peruutettavien transaktioiden ns. UNDO-listalle ja uudelleen kirjoitettavien transaktioiden ns. REDO-lista on aluksi tyhjä. DBMS käy transaktiolokin läpi checkpoint-tietueesta lokin loppuun asti kirjaten kaikki alkaneet transaktionumerot UNDO-listalle ja siirtäen UNDO-listalta kaikki lokin perusteella committoitujen transaktioiden numerot REDO-listalle. Lopuksi DBMS käy transaktiolokin läpi lopusta alkuun päin kirjoittaen UNDO-listan transaktioiden rivien alkukuvat tietokantaan ja vastaavasti kirjaten tietokantaan REDO-listan transaktioiden rivien loppukuvat transaktiolokin mukaisessa aikajärjestyksessä.



Kuva 7. Tietokannan toipuminen transaktiolokin tietueiden avulla

DBMS voi nyt avata viimeiseen committoituun transaktioon asti toipuneen tietokannan sovellusten käyttöön.

Huom. ARIES-protokollan mukaan niitä committoitujen transaktioiden tietoja, jotka on kirjoitettu tietokantaan checkpointin jälkeen (ns lazy writerin toimesta) ei kirjoiteta tietokantaan uudelleen. Tämä nopeuttaa tietokannan toipumista.

INDEX

Lukijoiden kommentteja “SQL Transactions – Theory and Hands-on Labs” -kirjasta:

“That is an amazing work of yours. It is perfect elearning material, together with online tools. I would never believe something like this is possible in connection with transaction management. Debian platform is welcome”.
- Ferenc Kruzslicz, University of Pécs, Hungary

“This is the only publication we know to have both solid theoretical foundation and practical exercises with so many different database products.”
A DBTech VET partner