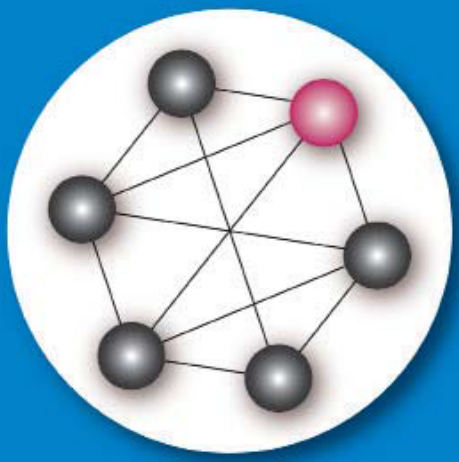


DBTechNet

DBTech VET

SQL Transactions

Teooria ja praktilised harjutused

Eesti keeles

Lifelong Learning Programme

Käesolev väljaanne on koostatud DBTech VET Teachers (DBTech VET) projekti raames.
Code: 2012-1-FI1-LEO05-09365.

DBTech VET on Euroopa Komisjoni ja partnerite poolt loodud Leonardo da Vinci koostööprogrammi projekt.

www.DBTechNet.org
DBTech VET

SQL andmetehingud – õppetöö juhend
Esimene trükk 2013

Autorid: Martti Laiho, Dimitris A. Dervos, Kari Silpiö
Väljaandja: DBTech VET Teachers project

ISBN 978-952-93-2420-0 (pehmes köites)
ISBN 978-952-93-2421-7 (PDF-formaadis)

SQL andmetehingud – õppetöö juhend

Eesmärgid

Usaldusväärne andmeliiklus baseerub ennekõike korralikult koostatud SQL süntaksil, kus vigaste andmete jõudmine andmete hulka on välistatud. Kui programmeerija ei tunne andmetehingute põhitõdesid, võivad tema töö tulemuseks olla puuduliku sisuga kirjed ning süsteemide üldise suutlikkuse aeglustumine või seiskumine. Nagu liikluses, kehtivad ka andmebaasidega suhtlemises kindlad reeglid, mida tuleb tunda ning järgida.

Käesoleva juhendi eesmärgiks on selgitada elementaarseid andmetehingute programmeerimise võtteid levinumates andmebaasi haldussüsteemides (ABHS). Lisatud on näiteid tüüpilistest lahendustest ning lihtsamatest optimeerimistest.

Sihtgrupid

Juhendi kasutajateks võiksid olla tehnika alaste kõrg- ja kutsekoolide õpetajad, praktikud ja õpilased. Samuti võib see huvi pakkuda tarkvaraarendajatele teiste levinumate andmebaaside haldussüsteemidega (ABHS) tutvumisel.

Eeldused

Kasutajal peaksid olema lihtsamad praktilised oskused mõnest SQL-põhisest andmebaaside haldussüsteemist (ABHS).

Õppemetoodika

Praktiliseks õpitu kinnistamiseks soovitatakse õppijal siinkohal tungivalt eksperimenteerida reaalses andmebaasi haldussüsteemi keskkonnas. Sel eesmärgil on kompileeritud tasuta virtuaalne keskkond näidis-skriptidega, mille leiate alajaotusest „**Lisalugemist, viiteid ja materjale**“.

Sisukord

1 SQL andmetehing kui loogiline tegevus

1.1 Andmetehingute (transaktsioonide) mõte

SQL andmetehingut ehk transaktsiooni võib vaadelda kui mõnda teist tavaelus teostatavat tehingut. Oma igapäevaelus teeme me samuti kõikvõimalikke äritehinguid, ostame erinevaid tooteid, muudame ja tühistame tellimusi, ostame kontserdipileteid, maksame arveid, kindlustusmakseid jne. Iga inimtegevus koosneb **loogiliste tööde jadast**, mis tähendab kas mingi tegevuse sooritamist terviklikult või kogu **tegevuse/andmetehingu** peatamist ja lõpetamist.

Peaaegu kõik infosüsteemid kasutavad mõnda andmebaasi haldussüsteemi (ABHS) andmehõiveks ja salvestamiseks. Kaasaegsed ABHS-id on juba piisavalt intelligentsed, et tagada hallatavate andmete terviklus ning võimaldada kiiret juurdepääsu mitmele samaaegselt pöörduvale kliendile. Need tooted pakuvad rakendustele usaldusväärseid teenuseid tagamaks andmete ja käskude järjepidevust, kuid neid tuleb osata kasutada õigesti. Selleks tulebki andmebaasi poole **pöördumised** üles ehitada loogiliste **andmetehingutena**.

Tarkvaras vale andmetehingute haldus ja juhtimine võivad viia soovimatute tulemusteni nagu näiteks:

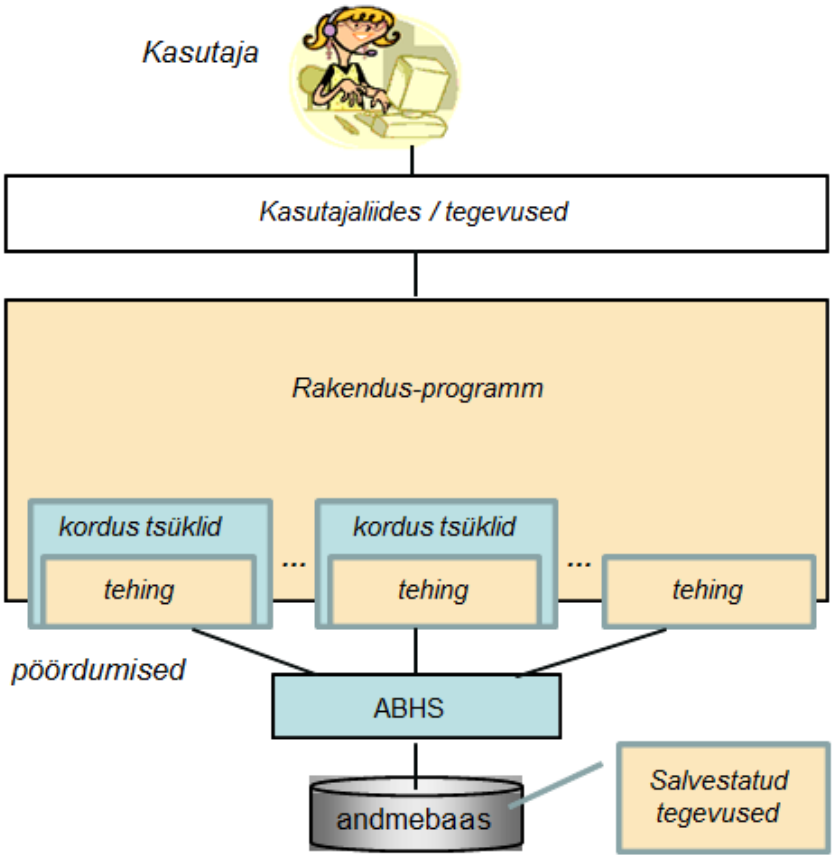
- netipoe süsteemis kaovad ära kliendi tellimused, maksete kinnitused, kauba saatelehed vms.
- vead istekohtade reserveeringutes või topelt reserveeringud bussi- või lennureisidel
- kadunud sissekanded häirekeskuste logides jne.

Eelpool kirjeldatud olukordi juhtub reaalses elus tihti, kuid neid enamasti avalikult ei tunnistata. DBTech VET projekti eesmärk on luua raamistik headest tavadest ja metoodikast, mis Aitaks neid olukordi vältida.

Siinkohal käsitletavad andmetehingud on andmete käitlemise mõttes **taastatavad tegevused**. Samuti tagavad need kogu andmebaasi taastatavuse, kui süsteem peaks kokku jooksma (nagu on näidatud **Lisas 3**) ning süsteemi järjepidevuse mitme samaaegse kasutajaga keskkonnas.

1.2 Kliendi-serveri kontseptsioon SQL keskkonnas

Selles juhendis keskendume andmete juurdepääsule SQL andmetehingu reeglite abil, kirjutades seda otse koodi, kuid peame silmas pidama, et vastav programmiline osa läheneb andmetele natukene teisest küljest (nagu on näidatud **Lisas 2**).

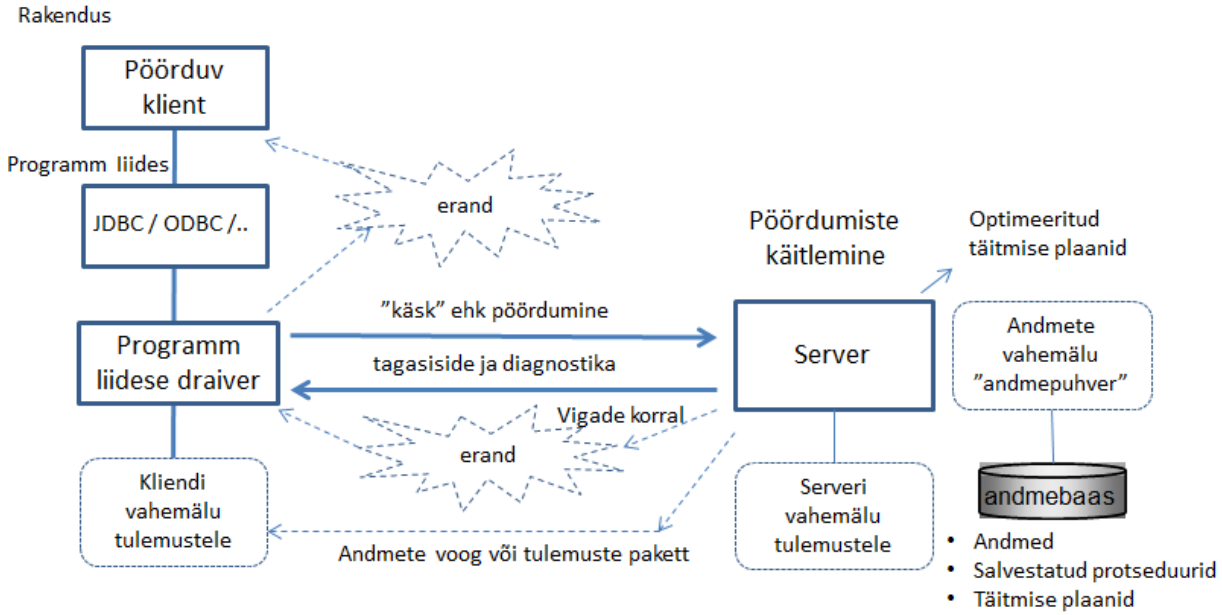


Joonis 1.1 SQL Andmetehingute asukoht rakenduse kihtides.

Joonisel 1.1 on kujutatud tüüpilist andmebaasi rakenduse arhitektuuri, kus andmetehing ise ei asu kasutajaliidesega samal tarkvarakihil. Kasutaja kõik tegevused, millega ta mõjutab rakenduse tööd on siin defineeritud kui kasutaja andmetehing. Seega võib ühe kasutaja andmevahetus sisaldada endas mitut SQL andmetehingut, mis omakorda sisaldavad andmete lugemist, töötlemist ning lõpuks kinnitust, millega uuendatakse andmebaasi kirjete sisu. Kordustsüklid kujutavad endast programmseid vahendeid, mille abil korratakse kasutajate samaaegse pöördumiste tõttu tekkinud nurjunud andmetehinguid, kuni need õnnestuvad.

Mõistmaks SQL andmetehingute mõtet, peame endale selgeks tegema kliendi ja server vahelise kätlus-dialoogi põhitõed. Ühenduse loomiseks peab rakendus algatama **andmebaasiühenduse**, mille sees omakorda avatakse **SQL sessioon**. Lihtsuse huvides nimetame rakendust edaspidi **SQL-kliendiks** ja andmebaasi serverit **SQL-serveriks**. Serveri poolt vaadatuna kasutab klient andmebaasi teenust n.ö. klient-server režiimis, saates **SQL käske** funktsioonide/meetodite parameetritena läbi programmeerimisliidese **(**API)[1]. Olenemata kasutatavast liidesest, toimub loogilisel tasandil dialoog SQL-keeles ning andmete õigsus tagatakse SQL andmetehingute reeglistiku täitmise abil**.**

[1] ODBC, JDBC, ADO.NET, LINQ, jne. sõltuvalt kasutatavast programmeerimiskeelest, nagu C++, C#, Java, PHP, jne.



Joonis 1.2 SQL käskude töötlemise selgitus.

Joonis 1.2 selgitab meile SQL käskude ringlust, alustades **kliendipoolsest pöördumisest**, mis salvestatakse töötlemiseks server puhvermällu ning lõpetades serveri poolse vastusega pöördumisele. SQL käsud võivad koosneda ühest või mitmest **SQL lausest**, mis esmalt sõelutakse (ingl. *parsing*), analüüsitakse vastavuses andmebaasi sisuga, optimeeritakse ning lõpuks täidetakse. Suhteliselt aeglase kettale salvestamise tõttu toimub kiiruse ja süsteemi võimekuse huvides kogu andmetöötlus muutmälus asuvas puhveralas, kuhu salvestatakse ka kõik hiljuti kasutatud kirjed.

Sisestatud SQL käskude täitmine serveris on nende õnnestumise seisukohalt **atomaarne** ehk jagamatu – kogu käsu sisu peab saama 100% täidetud, vastasel juhul tühistatakse kogu antud käsk ning taastatakse sellele eelnenud olukord. Vastuseks saadab server tagasiside, millest teavitatakse klienti käs(k)u(de) õnnestumis(t)est või selle nurjumis(t)est; viimased esitatakse kliendile erandite käsitluste jadana. Samas peame meeles pidama, et SQL laused UPDATE ja DELETE saavad alati positiivse tagasiside isegi, kui neile vastavaid ridu andmebaasist ei leitud. Rakenduse vaatenurgast on tõenäoliselt tegemist veaolukorraga aga käsu täitmise seisukohast on tegemist õnnestumisega. Sellest lähtuvalt peab rakenduse kood uurima serveri tagasisidet ning tuvastama, mitut rida antud käsk mõjutas.

SELECT käsuga saadud tulemus võetakse kliendi pool vastu rida realt, seda siis kas otse serverist või kliendiarvuti enda vahemälust.

1.3 SQL andmetehingud

Kui rakendus peab täitma SQL käskude jada jagamatult, peavad need käsud olema grupeeritud loogiliseks tegevuseks ehk SQL andmetehinguks, mis andmeid töödeldes viib andmebaasi sisu ühest kindlast olekust teise ning on seega **täideviidud tegevus**. Iga edukalt täidetud andmetehing lõpeb käsuga **COMMIT** samas, kui iga nurjunud tehing lõppeb käsuga **ROLLBACK**, millega taastatakse andmetehingule eelnenud olukord. Seega võib andmetehingut nimetada ka **taastatavaks tegevuseks**. ROLLBACK käsu eelis seisneb selles, et kui rakendusse programmeeritud andmetehingut pole mingil põhjusel võimalik lõpule viia, pole hiljem vaja

täidetud käske ükshaaval tagasi võtta vaid piisab ühest alati toimivast käsust, et endine olukord taastada. Kinnitamata (ingl. *uncommitted*) andmetehingud taastatakse automaatselt juhul, kui süsteem peaks mingil põhjusel kokku jooksma või ühendus kliendi ja server vahel kaduma. Samuti on mõned ABHS-id võimelised samaaegsete pöördumiste konflikti korral automaatselt andmebaase taastama nagu seda tehakse ühes alljärgnevatest näidetest.

Kokkuvõtvalt – andmetehingu sisu käsitletakse ühe tegevusena, kui selle täitmine nurjub, saab ühe käsuga taastada andmetehinguga senimaani tehtud tegevused ning ROLLBACK käsk toimib alati garanteeritult. Kui süsteem peaks andmetehingu töötlemise ajal kokku jooksma, taastatakse endine olukord koheselt peale taaskäivitust.

ISO SQL standardi järgi nagu ka näiteks DB2 ja Oracle andmebaasides alustab iga uus SQL käsk peale eelmist käsku või andmetehingut automaatselt uut iseseisvat SQL andmetehingut. Sellisel juhul saab antud olukorda nimetada andmetehingute **iseseisvaks alustamiseks** ABHS poolt.

Mõned ABHS-id nagu näiteks SQL Server, MySQL/InnoDB, PostgreSQL ja Pyrrho töötavad vaikimisi **AUTOCOMMIT** režiimis. See tähendab, et iga SQL käsk täidetakse koheselt ning meil **puudub automaatse taastamise võimalus**. Seega vigade tekkimisel peab kliendi poolne rakendus kasutama n.ö. pöördtegevust, et esialgne olukord taastada, kuid see võib osutuda võimatuks, kui mitu samaaegselt server poole pöördunud klienti on teinud erinevaid muudatusi. Samuti võib katkenud ühenduse korral jääda andmebaasi sisu lünklikuks, sisaldades poolikuid andmeid. Selleks, et saaksime kasutada õiget andmetehingu loogikat, tuleb iga uue andmetehingu algus ära märkida täiendava käsuga nagu: BEGIN WORK, BEGIN TRANSACTION või START TRANSACTION, olenevalt kasutatavast ABHS-ist.

ABHS-ides MySQL ja InnoDB saab töötavas SQL sessioonis valida, kas soovime kasutada andmetehingute iseseisvat algatamist või mitte, sisestades käsu:

SET AUTOCOMMIT = { 0 | 1 }

kus 0 viitab automaatsele andmetehingute algatamisele, 1 aga AUTOCOMMIT režiimile.

Mõned ABHS-d nagu näiteks Oracle, kinnitavad automaatselt andmetehingu mistahes andmekirjeldus käsu järel (s.t. CREATE, ALTER või DROP mõne andmebaasi objekti suhtes nagu TABLE, INDEX, VIEW, vms).

SQL serveris võib terve **haldurprogrammi**, sealhulgas tema andmebaasid, konfigureerida automaatselt tehinguid alustama; samuti võib käimasoleva **SQL sessiooni** (ühenduse) lülitada kas automaatse andmetehingu või tagasi AUTOCOMMIT režiimi, kasutades käsku:

SET IMPLICIT_TRANSACTIONS { ON | OFF }

Mõned abiprogrammid nagu käsurea rakendus (CLP) IBM-i DB2-s ja mõned andmebaasi pöördusliidesed nagu ODBC ja JDBC töötavad vaikimisi AUTOCOMMIT režiimis. Näiteks JDBC liideses tuleb iga uut tehingut alustada alltoodud meetodi väljakutsumisega ühenduse objektile:

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

Mõned andmetehingud võivad hariliku käskude jada asemel sisaldada ka keerukamat loogikat. Nii võib andmetehingu kood sisaldada tingimusi, milliste serveri poolt saadud andmete korral milliseid edasisi käske jagada. Isegi sellisel juhul nimetatakse seda SQL tehingut loogiliseks tegevuseks, mis õnnestub või määratakse taastamisele. Kuid andmetehingu nurjumine ei too alati automaatselt kaasa ROLLBACK[2] taastamise käsu käivitamist – vea peaks leidma andmebaasiga suhtlev rakendus (vt. „**SQL vigade diagnostika**“) ning käivitama oma koodis taastamise käsu.

1.4 Andmetehingute loogika

Vaatame järgnevat pangakontode tabelit.

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

Tüüpilise SQL andmetehingu õpiku näitena kanname kindla summa raha (antud juhul 100 eurot) ühelt kontole teisele:

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

Süsteemi rikke või kliendi ja serveri vahelise ühenduse katkemise korral peale esimest kirje muutmist (UPDATE) garanteerib tehingu protokoll selle, et kontolt „101“ raha kaduma ei lähe ning andmetehingu eelne olukord taastatakse. Kuid eelpoolt näidatud andmetehing on veel päris kaugel usaldusväärsest lahendusest:

a) Juhul, kui kumbagi kontot ei eksisteeri, annab UPDATE käsk SQL protokolli kohaselt „õnnestunud“ tegevuse tagasiside. Sellepärast peaks kood kontrollima, mitut rida kumbki UPDATE käsk mõjutas.

b) Kui esimene UPDATE käsk saab konto „101“ seisu negatiivseks minnes tagasisideks veateate (rikutakse tabeli CHECK piirangut), teine UPDATE käsk aga täidetakse ning loetakse õnnestunuks, siis andmebaasi jäävad vigased andmed.

Sellest lihtsast näitest järeldub tõsiasi, et tarkvara arendaja peab teadma, kuidas erinevad ABHS-id käituvad ning kuidas SQL diagnostika toimib erinevates programmeerimisliidestes. Isegi siis on meil veel palju õppida ning kasulikke optimeerimisi, mida oma rakenduse koodis teha.

1.5 SQL vigade diagnostika

Erinevalt harilikust käskude jadast võivad mõned SQL andmetehingud sisaldada loogilisi tehteid ja tingimusi ning vastav loogika võib otsustada tegevuste täitmise ajal vastavalt saadud tagasisidele edasiste tegevuste üle. Kas sellisel juhul on antud SQL Andmetehingu näol tegemist jagamatu tegevusega, mille täitmine õnnestub või tühistatakse kogu andmetehingu täitmine. Andmetehingu nurjumine ei too aga alati kaasa automaatset andmetehingu tühistamist ja

[2] ABHS-id meie DebianDB virtuaallaboris eriravad PostgreSQL liideses tehingu nurjumise korral kõiki teisi käske peale ROLLBACK-i

taastamist (ROLLBACK)[3]. Kliendi rakendus peaks kontrollima iga serveri pöördumise kohta saadud tagasisidet ning otsustama vea korral kas see andmetehing tühistada või mitte.

Selle jaoks oli algselt ISO SQL-89 standardis ettenähtud täisarvu (*integer*) tüüpi indikaator **SQLCODE**, mis lisati iga tagasiside lõppu. 0 tähendas õnnestunud täitmist, 100 tähendas, et vastavaid ridu ei leitud, ülejäänud numbrid olid aga tootespetsiifilised (kajastusid konkreetse ABHS juhendis) – ülejäänud positiivsed väärtused näitasid erinevaid hoiatusi ning negatiivsed väärtused vigu.

ISO SQL-92 standardis asendati aegunud SQLCODE 5-tähelise string tüüpi indikaatoriga **SQLSTATE**, milles 2 esimest tähemärki näitasid SQL vea või hoiatuse klassi ning 3 järgmist tähemärki selle alamklassi. Kombinatsioon „00000“ indikaatoris SQLSTATE tähendas õnnestunud täitmist. Sajad erinevad vigade ja hoiatuste klassi standardiseeriti (näiteks SQL tabelite tingimuste rikkumist), kuid endiselt jäi palju kombinatsioone endiselt ABHS spetsiifiliseks. Kombinatsioon klassi koodiga „40“ tähendas katkenud andmetehingut mis oli põhjustatud samaaegsest kirjutamisest, veast salvestusalgoritmis, ühenduse katkemisest või serveri poolsest veast.

Selleks, et server saaks anda toimunud muudatustest kliendile paremat tagasisidet, täiendas X/Open grupp SQL keelt GET DIAGNOSTICS käsuga, mis võimaldab täpsustada tagasiside detaile ning käia läbi diagnostika logi leidmaks erinevaid vigu ja hoiatusi. See täiendus on viidud ISO SQL standardisse sisse alates SQL:1999 versioonist, kuid ainult osa sellest on tegelikult kasutusel ABHS-ides nagu DB2, Mimer ja MySQL 5.6. Järgneb MySQL 5.6 koodinäide detailse tagasiside päringu kohta:

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

Mõned süsteemid suudavad ära kasutada SQL keelele omaseid täiendavaid indikaatoreid ja funktsioone. Näiteks MS SQL Serveri Transact-SQL-is (vahel ka lühidalt T-SQL) on võimalik kasutada @@-muutujaid nagu näiteks @@ERROR lokaalsete veakoodide jaoks või siis @@ROWCOUNT viimati käideldud ridade arvu jaoks.

[3] PostgreSQL liides DBTech VET'i Debian VM rakenduses eirab sellise tehingu nurjumise korral kõiki teisi käske peale ROLLBACK-i

IBM-i DB2 SQL keeles kasutatakse ISO SQL diagnostika indikaatoreid SQLCODE ja SQLSTATE järgnevalt:

```
<SQL statement>                    // SQL käsk või andmetehing
IF (SQLSTATE <> '00000') THEN
     <error handling>              // tegevus vigade korral
END IF;
```

Oracle PL/SQL keeles kasutatakse BEGIN ja END plokkide vahelise koodi täitmisel vigade ehk erandite käsitlemiseks EXCEPTION viita:

```
BEGIN
     <processing>             // SQL käsk või andmetehing
EXCEPTION                     // erandid
WHEN <exception name> THEN    // konkreetne erand
     <exception handling>;    // vastav tegevus
...
WHEN OTHERS THEN              // ülejäänud erandid
     err_code := sqlcode;
     err_text := sqlerrm;
     <exception handling>;    // vastav tegevus
END;
```

Esimesed SQL diagnostika alaste rakenduste kirjeldused leiate JDBC meetodist ODBC ning erandite/vigade loeteludest SQLExceptions ja SQLWarnings. JDBC Java programmeerimise liideses tekitavad SQL vead automaatselt SQL erandi, mille töötlemiseks kasutatakse *try-catch* plokki (vaata ka **Lisa 2**):

```
... throws SQLexception {
...
try {                         // tegevus, mille erandeid hiljem
     ...                      //   allpool töödeldakse
     <JDBC statement(s)>      // SQL käsk või andmetehing
}
catch (SQLException ex) {     // erandi tuvastus ja linkimine
     <exception handling>     // vastav tegevus
}
```

JDBC-s tagastab käsku täitev meetod kliendile alati ka käideldud ridade arvu *rowcount*.

1.6 Praktilised harjutused

> Ärge kunagi uskuge kõike, mida loete! Usaldusväärse rakenduse looja peab **eksperimenteerima** ja **veenduma**, et tema kirjutatud kood töötab konkreetse ABHS peale nii nagu soovitud. Erinevad ABHS-id käituvad väga erinevalt isegi elementaarsete SQL andmetehingute käsitlemise osas.

Lisas 1 käsitleme automaatselt ning kliendi poolt alustatud SQL Andmetehingute, käskude COMMIT ja ROLLBACK ning andmetehingute loogika testimist MS SQL Serveri peal, kuid selle juhendi lugeja peaks neid kindlasti ise praktikas järele proovima, et veenduda SQL Serveri

käitumises näidetes Lisa 1.1 – Lisa 1.2 . Microsofti leheküljelt on harjutamiseks võimalik tasuta alla laadida SQL Server Express nimeline programm.

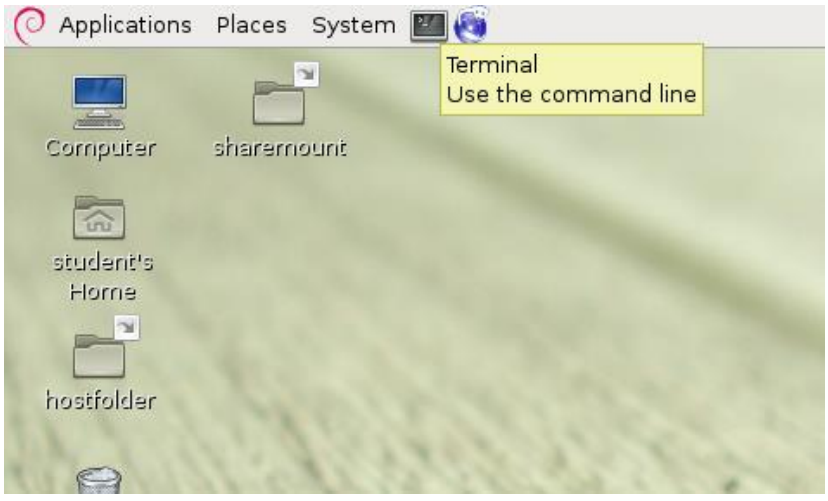
Praktiliste harjutuste esimeses peatükis tutvustatakse õppijale DBTechNet'i tasuta virtuaallabori DebianDB kasutamist, milles on mitmeid eelpaigaldatud ABHS-e nagu IBM DB2 Express-C, Oracle XE, MySQL GA, PostgreSQL ja Pyrrho. MySQL GA pole just kõige usaldusväärsem ABHS selles paketis, kuid tänu laialdasele kasutusele koolides, kasutame seda esmaste andmetehingu-põhiste andmetöötlusülesannete läbiviimiseks.

Alustame „Andmetehingute müsteeriumi" lahendamist MySQL-is. Olgu öeldud, et MySQL-i käitumine oleneb kasutatavast andmebaasimootorist. Algselt ei toetanud MySQL-i vaikemootor üldse andmetehinguid aga alates versioonist 5.1, kui uueks mootoriks sai InnoDB, olukord muutus. Sellegi poolest võivad mõned funktsioonid anda endiselt kummalisi tulemusi.

> Järgnevad katsetused on sarnaselt teostatavad kõigis DebianDB ABHS-ides ning koodinäited on skriptifailides **Lisas 1**. Need katsetused ei ole mõeldud näitamaks MySQL-i puudusi, kuid neid olukordi ei väldita kuna tarkvara arendajad peavad teadma erinevate toodete puudusi (ingl. *bug*) ning oskama neist hoiduda (ingl. *workaround*).

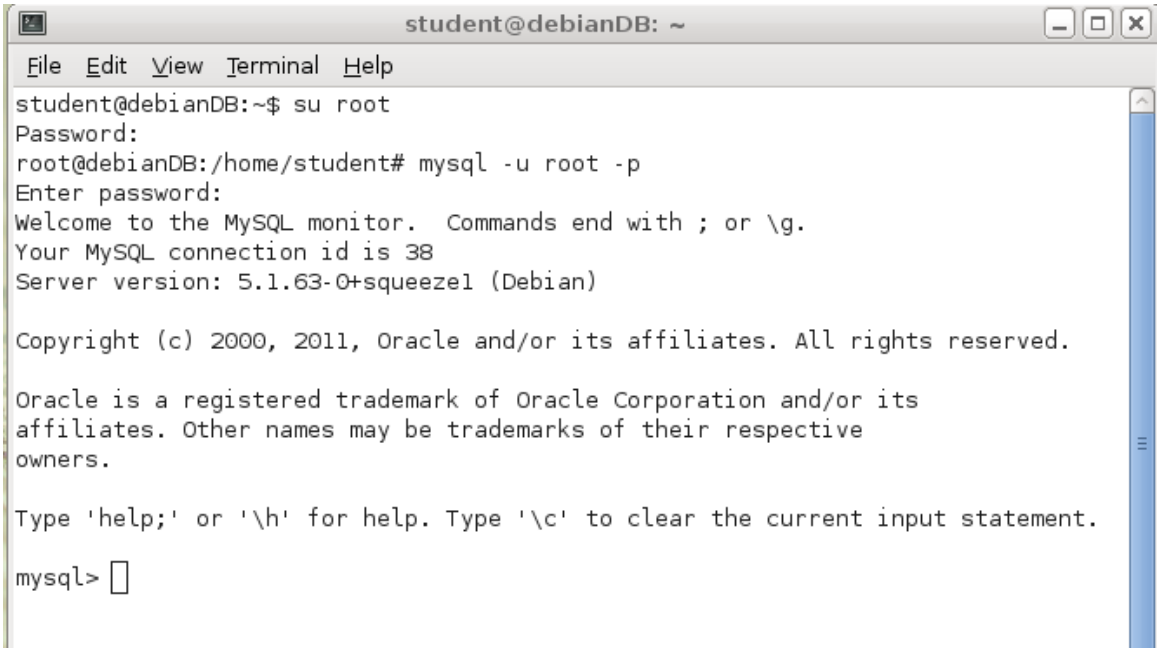
Eeldame, et lugeja on juba tutvunud DBTech VET väljaandega "Quick Start to the DebianDB Database Laboratory" (lühijuhend); tegemist on juhendiga, mis abistavad DebianDB paigaldamisel ja esmasel seadistamisel mõnes virtuaalkeskkonnas (tavaliselt Oracle VirtualBox).

Kui käesolevas juhendis viidatud DebianDB on seadistatud ja töötab, kasutame sisse logimiseks kasutajanime '**student**' parooliga '**Student1**'. Selleks, et luua esimene andmebaas, tuleb vahetada kasutajakontot, siseneda kasutaja '**root**' parooliga '**P4ssw0rd**' nagu on näidatud lühijuhendis. Selleks tuleb klikkida virtuaalmasinas ikoonil "Terminal/Use the command line" (joonis 1.3).



Joonis 1.3 Ikooni "Terminal / Use the command line" asukoht virtuaalmasina menüüs.

Sisestame järgmised Linux'i käsud terminali aknas käsureale, (joonis 1.4) et käivitada MySql kliendirakendus:



Joonis 1.4 MySQL-i sessiooni käivitamine 'root' juurkasutajana.

SQL käsk, millega luuakse uus “TestDB” nimeline andmebaas on järgmine:

```
-------------------------------------
CREATE DATABASE TestDB;
-------------------------------------
```

Selleks, et anda meie kasutajale juurdepääs andmebaasile TestDB koos kõikvõimalike õigustega, tuleb juurkasutaja ’root’ alt sisestada järgnev SQL käsk:

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

Nüüd on aeg väljuda juurkasutajaga ’root’ ning lõpetada see MySQL sessioon. Sisestades 2 järgnevat käsku, jõuame tagasi kasutaja ’student’ sessiooni:

```
-------------------------------------
EXIT;
exit
-------------------------------------
```

Kui ’root’ juurkasutaja sessioonist väljumise järel peaks DebianDB ekraan mustaks minema ja ilmuma parooli dialoog, siis käesolevas virtuaallaboris on kasutaja ’student’ parooliks ’Student1’.

Nüüd võime kasutajaga ’student’ käivitada MySQL kliendi ning siseneda andmebaasi TestDB:

```
-------------------------------------
mysql
use TestDB;
-------------------------------------
```

Nii algab uus MySQL sessioon.

Ülesanne 1.1

Loome andmebaasi tabeli „T“, milles on 3 tulpa: id (täisarv tüüpi, mittetühi, unikaalne väärtus), s (1 kuni 30 tähemärgist koosnev väärtus) ja si (väike täisarv):

```
-----------------------------------
mysql> CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(30), si
SMALLINT);

Query OK, 0 rows affected (0.27 sec)
-----------------------------------
```

MySQL klient kuvab peale igat SQL käsku jooksva diagnostika ehk tagasiside täitmise kohta.

Veendumaks, et loodud tabel on olemas ning soovitud struktuuriga, kasutame MySQL-i täiendavat DESCRIBE käsku:

```
-----------------------------------
DESCRIBE T;
-----------------------------------
```

> Linux-i MySQL on tõusutundetu, erandiks on andmebaaside ja tabelite nimed. See tähendab, et käsud "Describe T", "describe T", "create TaBle T ..." on samased aga "use testDB" ja "describe t" viitavad aga hoopis teistele andmebaasile ja tabelile, kui meie praktilistes harjutustes.

Lisame tabelisse mõned read:

```
-----------------------------------
INSERT INTO T (id, s) VALUES (1, 'first');
INSERT INTO T (id, s) VALUES (2, 'second');
INSERT INTO T (id, s) VALUES (3, 'third');
-----------------------------------
```

SQL käsk "SELECT * FROM T" abil saame kinnitust, et meie sisestatud 3 rida on tabelisse lisatud (nagu märkasite, sisestati ’si’ tulpa tühiväärtused ehk NULL).

> Veenduge alati, et sisestate iga käsu järel semikooloni (";") ning alles seejärel "enter".

Tuletades meelde, mida on seni kirjutatud andmetehingute kohta, proovime nüüd tühistada teostatud andmetehingu ning taastada endise olukorra järgneva käsuga:

```
-----------------------------------
ROLLBACK;
-----------------------------------
```

Esmalt tundub, et käsu täitmine õnnestus, kuid kasutades "SELECT * FROM T" käsku näeme, et meie sisestatud 3 rida on tabelis alles.

Sellele on ainult üks loogiline põhjendus – "AUTOCOMMIT". MySQL käivitub vaikimisi AUTOCOMMIT režiimis, kus andmetehingu alustamiseks tuleb kasutada "START TRANSACTION" käsku ning peale andmetehingu lõpetamist läheb MySQL tagasi AUTOCOMMIT režiimi. Selle tõestuseks proovime läbi järgnevad SQL käsud:

```
-----------------------------------
START TRANSACTION;

INSERT INTO T (id, s) VALUES (4, 'fourth');

SELECT * FROM T;
ROLLBACK;

SELECT * FROM T;
-----------------------------------
```

Küsimus

- Võrdle kahe viimase SELECT * FROM T käsuga saadud tulemusi.

Ülesanne 1.2

Sisestame järgnevad SQL käsud:

```
-----------------------------------
INSERT INTO T (id, s) VALUES (5, 'fifth');
ROLLBACK;
SELECT * FROM T;
-----------------------------------
```

Küsimused

- Milline on SELECT * FROM T käsuga saadud tulemus?
- Millis(t)ele järeldus(t)ele jõudsite seoses võimalike piirangutega START TRANSACTION käsu kasutamisel MySQL/InnoDB-s?

Ülesanne 1.3

Selle sessiooni AUTOCOMMIT režiim on välja lülitatud "SET AUTOCOMMIT" käsuga:

Esmalt kustutame kõik read peale ühe:

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

… ning lülitame AUTOCOMMIT režiimi välja:

```
-----------------------------------
SET AUTOCOMMIT = 0;
-----------------------------------
```

Sisestame jälle uued read:

```
-----------------------------------
INSERT INTO T (id, s) VALUES (2, 'second');
INSERT INTO T (id, s) VALUES (3, 'third');
SELECT * FROM T;
-----------------------------------
```

… ning seejärel ROLLBACK käsk:

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

Küsimus

- Millised eelised ja puudused on "SET TRANSACTION" käsu kasutamisel "SET AUTOCOMMIT" ees, kui soovime MySQL-i AUTOCOMMIT režiimi välja lülitada?

Kasutades MySQL-i Linux-i terminali/käsurida saab üles ning alla nooleklahvidega liikuda eelnevalt sisestatud käskude ajaloos. Teistes DebianDB virtuaallabori eelpaigaldatud ABHS keskkondades pole see alati võimalik.

Kahe järjestikuse kriipsu ("-") kasutamine SQL käsus tähendab kommentaari algust ehk kogu infot, mis jääb kriipsude ja 'enteri' vajutuse vahele ignoreeritakse SQL süntaksi analüsaatoris (ingl. *parser*).

SQL käsud jaguneva muuhulgas andmekirjelduskeele (DDL) käskudeks (näiteks CREATE TABLE, CREATE INDEX ja DROP TABLE) ja andmekäitluskeele (DML) käskudeks (näiteks SELECT, INSERT ja DELETE).

Ülesanne 1.4

Võttes arvesse eelnevat, uurime edasi ROLLBACK käsu ulatust:

```
-----------------------------------
INSERT INTO T (id, s) VALUES (9, 'will this be committed?');

CREATE TABLE T2 (id INT);
INSERT INTO T2 (id) VALUES (1);

SELECT * FROM T2;
ROLLBACK;

SELECT * FROM T;  -- What has happened to T ?
SELECT * FROM T2; -- What has happened to T2 ?
SELECT * FROM T3; -- Oops!

SHOW TABLES;
-----------------------------------
```

Küsimus

- Millis(t)ele järeldus(t)ele jõudsite?

Ülesanne 1.5

Tabeli „T" sisu viiakse sellisele kujule nagu see oli ülesandes 1.3:

```
-----------------------------------
```

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

Nüüd vaatame, kas viga MySQL-is kutsub automaatselt esile ROLLBACK-i. Sisestame järgnevad SQL käsud:

```
-----------------------------------
INSERT INTO T (id, s) VALUES (2, 'Error test starts here');
-- nulliga jagamine peaks tekitama vea
SELECT (1/0) AS dummy FROM DUAL;

-- proovime muuta rida, mida pole olemas
UPDATE T SET s = 'foo' WHERE id = 9999 ;

-- … ja kustutada rida, mida pole olemas
DELETE FROM T WHERE id = 7777 ;

--
INSERT INTO T (id, s) VALUES (2, 'Hi, I am a duplicate');
INSERT INTO T (id, s)
       VALUES (3, 'How about inserting too long of a string value?');
INSERT INTO T (id, s, si) VALUES (4, 'Smallint overflow for 32769?', 32769);
INSERT INTO T (id, s) VALUES (5, 'Transaction still active?');

SELECT * FROM T;
COMMIT;

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

Küsimused

a) Mida me saime teada automaatse taastamise kohta MySQL-i vigade korral?
b) Kas nulliga jagamine on viga?
c) Kas MySQL reageerib ületäituvustele?
d) Mida me saime teada järgnevatest näidetest:

```
-----------------------------------
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'
-----------------------------------
```

Antud näites on "23000" standardne SQLSTATE väärtus, mis näitab unikaalse väärtuse piirangu rikkumist ning 1062 on vastava vea kood MySQL-is.

Järgnev diagnostika näide on MySQL-is alates versioonist 5.6. Kõik „@“ märgiga algavad muutujad on MySQL-i lokaalsed tüübita muutujad SQL keeles.

```
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)

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

Ülesanne 1.6

Käesolev MySQL-i versioon ei toeta tulpade tasandil CHECK piiranguid, vaid ainult ridade tasandil. Aga olenemata sellest, ei kasuta see versioon CHECK piiranguid nagu selgub järgnevatest näidetest:

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

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

-- proovime juba tuttavat ülekannet
UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;
SELECT * FROM Accounts;
ROLLBACK;

-- proovime, kas CHECK piirang üldse töötab:
UPDATE Accounts SET balance = balance - 2000 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 2000 WHERE acctID = 202;
SELECT * FROM Accounts ;
ROLLBACK;
-----------------------------------
```

Järgmises SQL andmetehingus proovime kanda 500 eurot kontolt „101“ kontole acctID = 777, mida aga pole olemas:

```
-----------------------------------
```

```
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 777;
SELECT * FROM Accounts;
ROLLBACK;
-----------------------------------
```

Küsimused

a) Kas viimased kaks UPDATE käsku täideti olenemata sellest, et teisel juhul püüti omistada uut väärtust mitte eksisteerivale kontole/reale vastavas tabelis?

b) Kui ROLLBACK käsk oleks asendatud COMMIT-iga, kas sellisel juhul oleks käesolev andmetehing õnnestunud ning andmebaasi sisu oleks jäädavalt muudetud?

c) Kas käesolev SQL andmetehing rikub andmetervikluse nõuet? Kui jah, siis kas MySQL püüab takistada selle SQL andmetehingu teostamist või kas andmetehingu vältel saadetakse kliendi rakendusele vastava sisuga tagasisidet, et tuvastada viga ja tegutseda andmetervikluse taastamise nimel?

Ülesanne 1.7 – SQL andmetehing kui taastatav tegevus

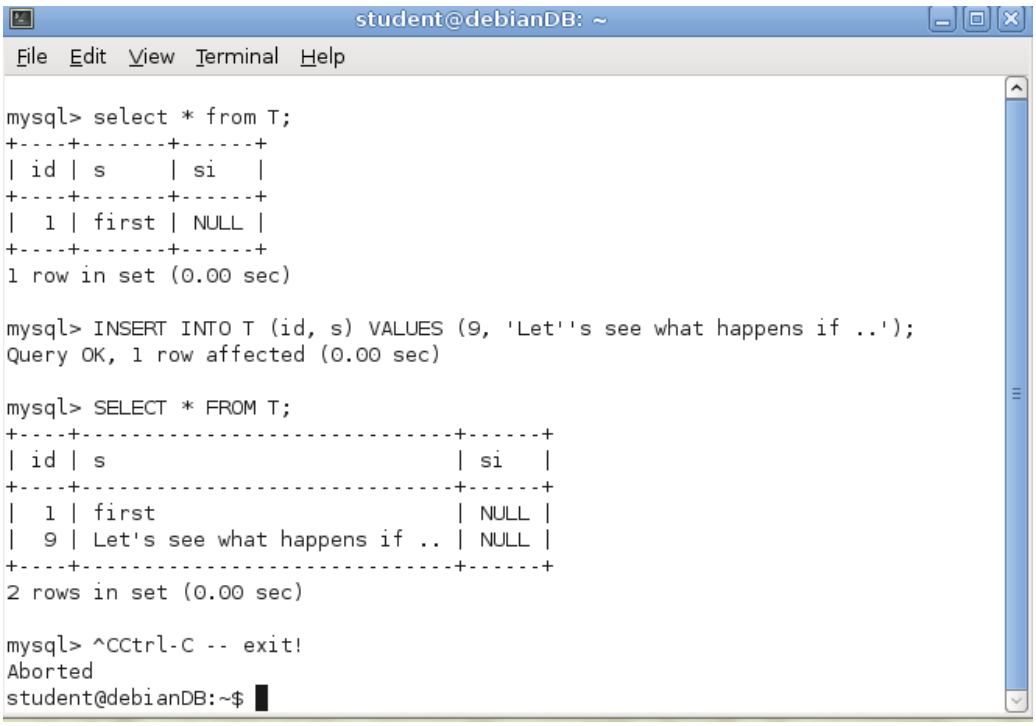
Katsetame **taastatava tegevuse** omadust kinnitamata SQL andmetehingute korral. Selleks peame alustama andmetehingu ning katkestama SQL ühenduse kliendi ja serveri vahel. Ühenduse taastamisel kontrollime, kas andmebaas sisaldab katkestuse eelseid kinnitamata muudatusi.

Vaatame, kuidas katkenud ühendus (mis Interneti rakenduste puhul pole mingi haruldane nähtus) mõjutab käimas olevat kinnitamata andmetehingut:

Tabelisse „T“ sisestatakse uus rida:

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

Ühendus katkestatakse tahtlikult kliendi poolt kasutades "Control C" (Ctrl-C) käsku (joonis 1.5):



Joonis 1.5 ABHS kokkujooksmise matkimine.

Sulgeme DebianDB akna, avame uue ning alustame uut MySQL sessiooni TestDB andmebaasiga:

```
-----------------------------------
mysql
USE TestDB;
SET AUTOCOMMIT = 0;
SELECT * FROM T;
COMMIT;
-----------------------------------
```

Küsimus

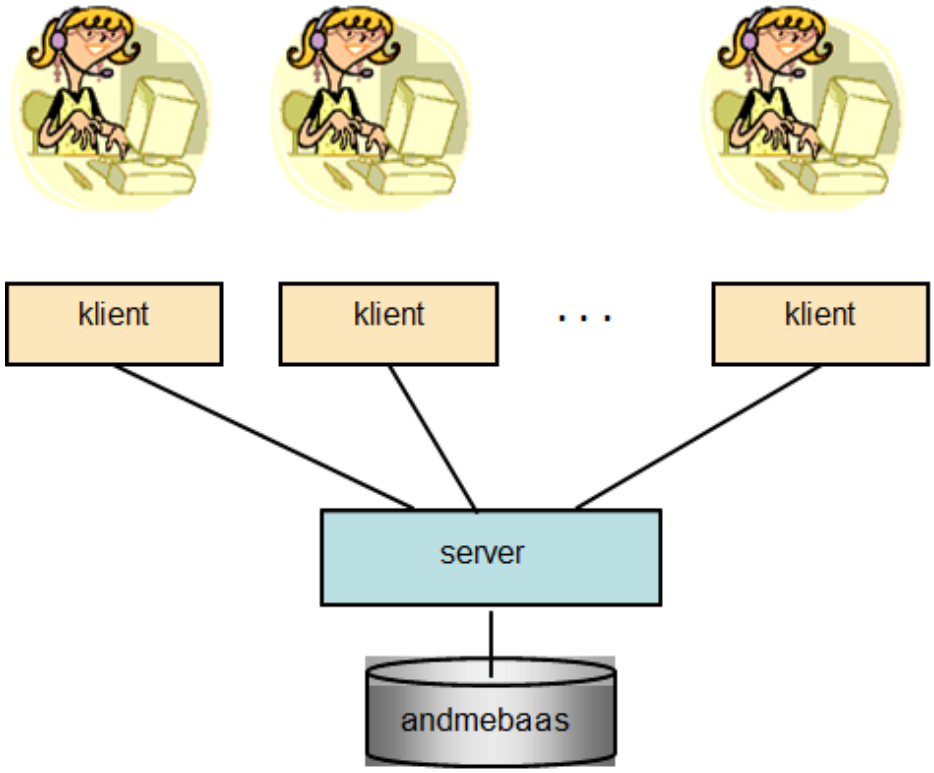
- Kas tabeli T ridades on näha muutusi?

Kõik andmebaasis toimunud muudatused on nähtavad **andmetehingute logifailis**. **Lisas 3** kirjeldatakse, kuidas on võimalik andmetehingute logi abil taastada andmebaas kuni viimase kinnitatud käsuni enne süsteemi kokkujooksmist. Seda saab proovida, kui ülesandes 1.7 ühenduse katkestamise asemel sulgeda MySQL serveri teenus.

2 Samaaegsed andmetehingud

Nõuanne!
Ärge uskuge seda, mida te loete või kuulete andmetehingute toest erinevates ABHS-ides! Selleks, et luua usaldusväärseid rakendusi, peate te ise nende süsteemide funktsioonid läbi proovima. Erinevused pole mitte ainult SQL andmetehingute toetamises, vaid ka selles, kuidas elementaarsete andmetehingute algatamist ja kinitamist käsitletakse.

Rakendus, mis töötab laitmatult ühe kasutajaga keskkonnas, ei pruugi aga alati mitme kasutaja samaaegseid pöördumisi ühiskasutus keskkonnas (joonis 2.1) korrektselt teenindada.



Joonis 2.1 Mitu klienti pöördub korraga sama andmebaasi poole (ühiskasutus keskkond).

2.1 Samaaegsuse probleemid – usaldusväärsed andmed

Korraliku samaaegse täitmise juhtimise puudumine või ABHS teenuste oskamatu kasutamine võib viia selleni, et **andmebaasi sisu** või meie **päringute vastused** on vigased ehk **mitte usaldusväärsed**.

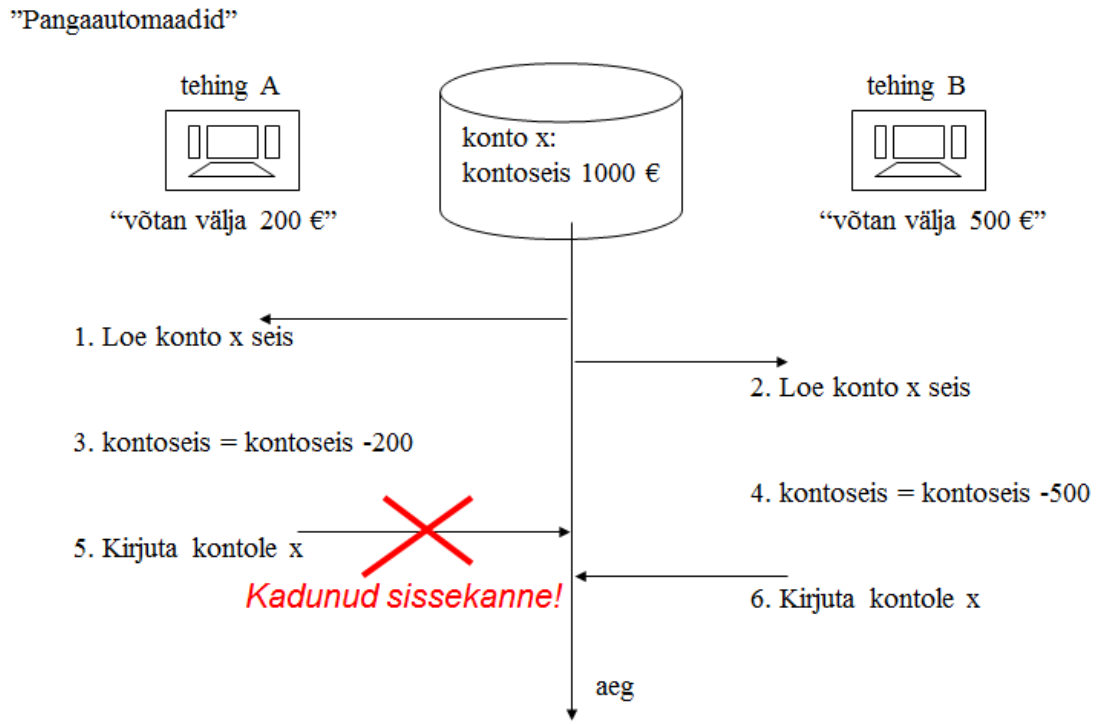
Vaatleme järgnevalt tüüpilisi samaaegse täitmise probleeme ehk anomaaliaid:

- Kadunud sissekande probleem;
- Poolikute ehk teiste kinnitamata andmetehingute andmete lugemise probleem;
- Hägusa lugemise ehk vahepeal teise andmetehingu poolt muudetud andmete lugemise probleem;
- Fantoomkirjete ehk vahepealse teise andmetehingu poolt lisatud kirjete lugemise probleem,

ning seejärel vaatame, kuidas võiks nimetatud probleeme ISO SQL standardi kohaselt reaalses ABHS-is lahendada.

2.1.1 Kadunud sissekande probleem

C. J. Date pakub järgnevas näites (joonis 2.2) välja olukorra, kus kaks klienti erinevate sularaha automaatide juures võtavad raha välja sama konto pealt, kus algselt on 1000 eurot.



Joonis 2.2 Kadunud sissekande probleem

Ilma samaaegse täitmise juhtimiseta läheks andmetehingu „A" poolt 5. sammul kirjutatav 800 eurot kaduma, sest andmetehing „B" kirjutakse selle arvutatud 500 euroga 6. sammul lihtsalt üle. Kui see juhtuks enne andmetehingu „A" kinnitamist, oleks tegemist **kadunud sissekandega**. Igal kaasaegsel ABHS-il on sisse ehitatud mingi elementaarne samaaegse täitmise juhtimine, mis kaitseb ühe andmetehingu kirjutatud andmeid teise samaaegse andmetehingu poolt ülekirjutamise eest enne esimese andmetehingu lõppu.

Kui see stsenaarium lahendataks "SELECT .. UPDATE" käskude jada abil ning kasutataks mõnda andmebaasi lukustuse skeemi, poleks enam tegemist kadunud sissekandega vaid juba ummikseisuga (DEADLOCK, andmetehingud jäävad teineteise lõppu ootama, selgitame hiljem), mille korral näiteks andmetehing „B" tühistatakse ABHS-i poolt ja andmetehing „A" kinnitatakse ning täidetakse.

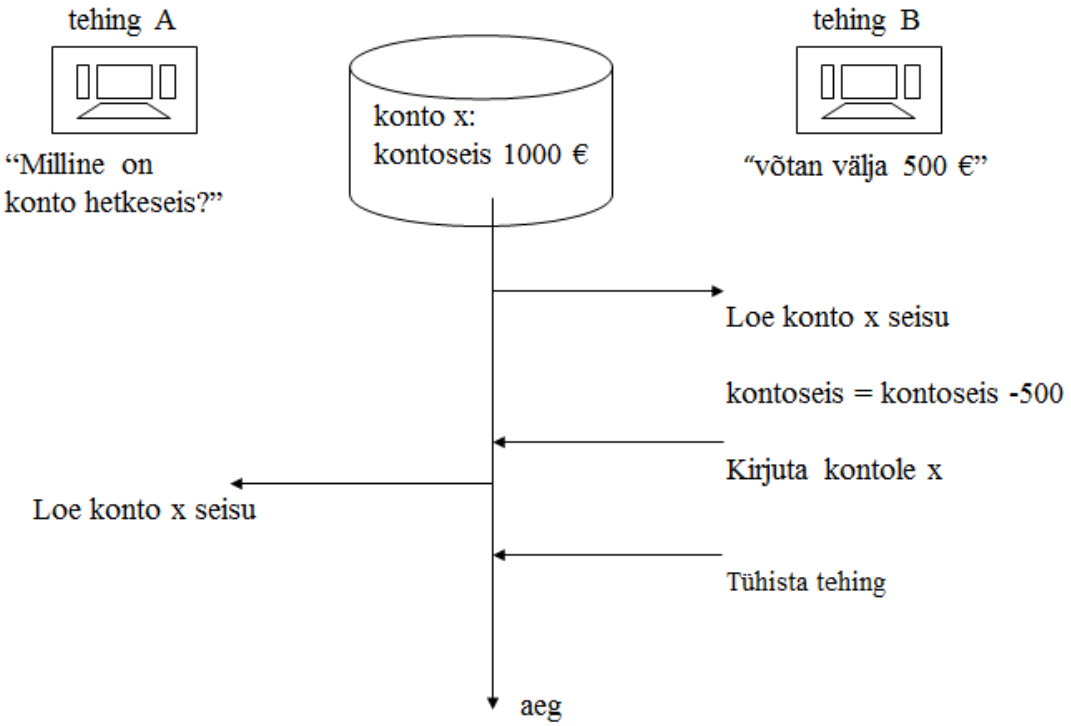
Kui sellel juhul rakendatakse n.ö. ***tundlikku uuendamist*** (vahetu loetud väärtuse muutmine) nagu näiteks

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

ja andmetehing kaitstakse lukustamisega, õnnestub antud tegevus.

2.1.2 Poolikute andmete lugemise probleem

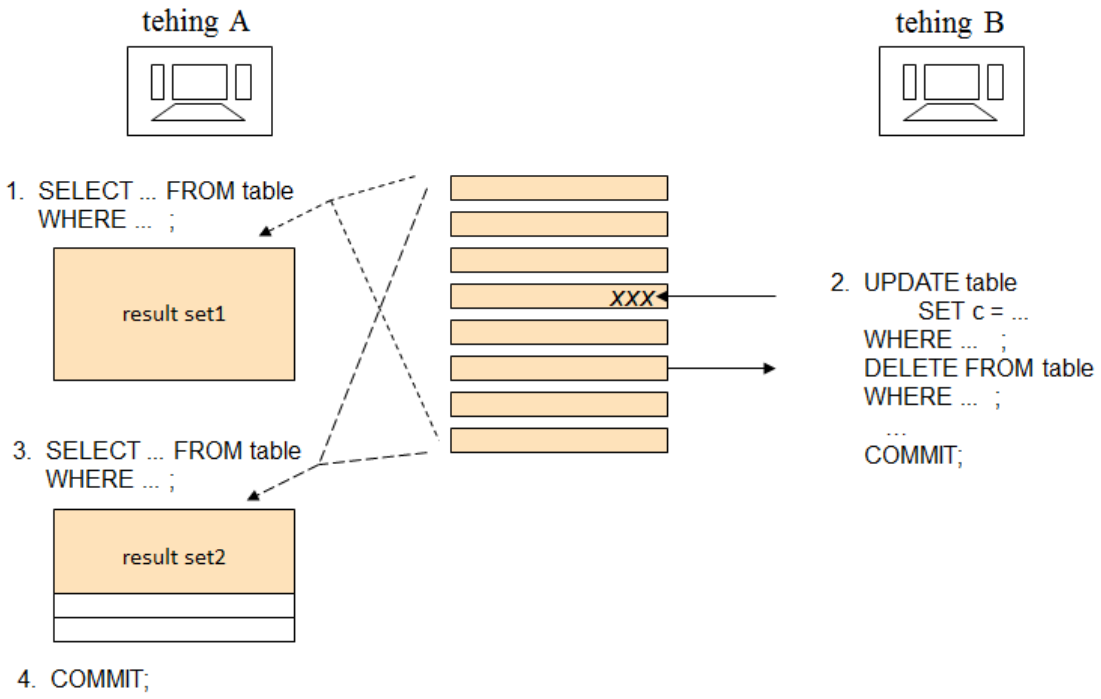
Poolikute andmete lugemise anomaalia (joonisel 2.3) puhul ei arvesta andmetehing selle riskiga, et loetud andmete hulka võivad sattuda samaaegse andmetehingu poolt kirjutatud, kuid kinnitamata andmetehingu kirjed, mis võivad enne meie andmetehingu kinnitamist tühistatud saada. Sellised andmetehingud ei tohiks teha andmebaasis ühtegi muudatust ning üldiselt on igasuguse kinnitamata andmetehingu andmete lugemine ja kasutamine seotud riskiga, kus tulemuseks on tõenäoliselt vigased andmed või valed otsused.



Joonis 2.3 Poolikute andmete lugemise näide.

2.1.3 Hägusa lugemise probleem

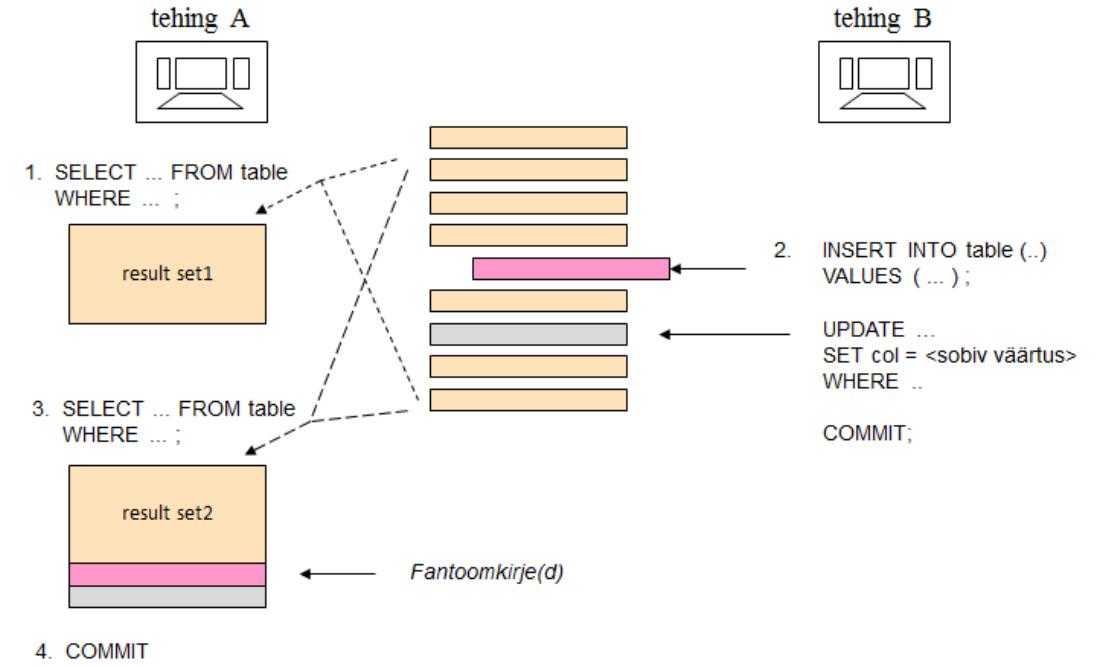
Hägusa lugemise anomaalia (joonisel 2.4) väljendub korduvalt loetavate ridade erinevuses s.t. sama rea järgmise lugemise eel on samaaegne andmetehing muutnud selle rea väärtusi. Samuti ei pruugi mõned eelnevalt loetud read järgmisel lugemisel enam eksisteerida ning välista ei saa ka uute ridade tekkimist andmetehingu poolt loetavate andmete hulka (vt. **fantoomkirje probleemi**).



Joonis 2.4 Hägusa lugemise anomaalia andmetehingu A seisukohast.

2.1.4 Fantoomkirjete probleem

Joonis 2.5 kirjeldab fantoomkirjete probleemi, mille korral sisaldab järgmise lugemise tulemuste hulk uusi ridu. Siinkohal võib olla tegemist nii uute ridadega kui ka ridadega, mille tulpade väärtusi on muudetud nii, et nad vastaksid päringu tingimustele.



Joonis 2.5 Näide fantoomkirjete probleemist.

2.2 SQL andmetehingute ACID printsiip

Aastal 1983 „ACM Computing Survey“-s Theo Härder’i ja Andreas Reuter’i poolt avaldatud ACID printsiip defineerib usaldusväärse SQL andmetehingu ühiskasutus keskkonnas. ACID on lühend neljast olulisest inglise keelsest SQL andmetehingu omadusest:

Atomic (atomaarne, jagamatu). Andmetehing peab olema jagamatu (põhimõttel „kõik või mitte midagi“) tegevuste jada, mis õnnestub ja täidetakse või nurjub ning tühistatakse täielikult, taastades andmetehingu eelse olukorra andmebaasis.

Consistent (kooskõlaline, täielik). Tegevuste jada peab viima andmebaasi sisu ühest täielikust olekust teise. See tähendab, et hiljemalt andmetehingu kinnitamise hetkeks pole andmetehing rikkunud ühtegi andmebaasi piirangut (primaarvõtmeid ehk peavõtmeid, unikaalseid võtmeid ehk unikaalseid tunnuseid, välisvõtmeid, piiranguid).

Enamik ABHS-e rakendavad piiranguid jooksvalt tegevuste käigus. Siin esitatud kooskõlalisuse nõue eeldab tarkvara arendajalt korrektset rakenduste loogikat, koodi, korralikku andmetehingute ülesehitust ning nende käitumise ja erandite käsitlemise testimist.

Isolated (eraldatud, peidetud, isoleeritud). „Andmetehingu sisesed tegevused peavad olema varjatud teiste samaaegselt toimuvate andmetehingute eest“ (Härder, Reuter). Enamik ABHS-e ei järgi seda reeglit täielikult nagu on kirjeldatud meie „samaaegsete andmetehingute“ materjalis. Kaasaegsed ABHS-id kasutavad erinevaid samaaegsete andmetehingute juhtimise tehnoloogiaid kaitsmaks andmetehinguid samaaegse täitmise kõrvalnähtuste eest ning tarkvara arendajad peavad neid teadma ja oskama õigesti kasutada.

Durable (kestev, püsiv, pikaajaline). Teostatud tegevused püsivad andmebaasis isegi võimalike süsteemi rikete korral.

ACID printsiibi kohaselt pole lubatud kinnitada andmetehingut, mis ei vasta eelpool loetletud omadustele. Kliendi rakendus või andmebaasi server peavad head seisma selle eest, et selline andmetehing saaks tühistatud.

2.3 Andmetehingu eraldatuse tasemed

Eraldatuse omadus ACID printsiibi juures on paras pähkel. Samaaegsete andmetehingute juhtimise mehhanismide vale kasutamine võib viia konfliktideni või liialt pikkade ooteaegadeni päringute töötlemisel, mis teevad andmebaaside töö ebaefektiivseks.

ISO SQL standard ei täpsusta, kuidas samaaegsuse juhtimist peaks rakendama, kuid lähtuvalt eelpool toodud anomaaliatest, defineerib see eraldatuse tasemed, mis peaksid erinevaid olukordi lahendama (tabel 2.1). Eraldatuse tasemetest tulenevate lugemise piirangute kirjeldused leiame tabelist 2.2. Mõned neist tasemeist on väiksemate piirangutega, teised aga rangemad, pakkudes paremat kaitset, kuid seda sageli andmebaasimootori jõudluse arvelt.

Tasub silmas pidada, et eraldatuse tasemed ei käsitle ühtegi andmete kirjutamise piirangut. Andmete kirjutamisel tuleb tüüpiliselt rakendada mõnda kaitselukustust ning õnnestunud kirjutamine on alati andmetehingu lõpuni kaitstud mistahes ülekirjutamise eest.

Tabel 2.1 ISO SQL Andmetehingute eraldatuse tasemed vs. samaaegsuse anomaaliad.

Anomaaliad: / Eraldatuse tase:	Kadunud sissekanne	Poolikud andmed	Hägus lugemine	Fantoom kirjed
Kinnitama tehingute lugemine (READ UNCOMMITTED)	võimatu	võimalik !	võimalik !	võimalik !
Kinnitatud tehingute lugemine (READ COMMITTED)	võimatu	võimatu	võimalik !	võimalik !
Korratav lugemine (REPEATABLE READ)	võimatu	võimatu	võimatu	võimalik !
Järjepidev (SERIALIZABLE)	võimatu	võimatu	võimatu	võimatu

Tabel 2.2 ISO SQL (s.h. DB2) Andmetehingute eraldatuse tasemete selgitused.

ISO SQL eraldatuse tase	**DB2** erald. tase	**Piirangute kirjeldus**
Read Uncommitted	UR	Kinnitamata andmetehingute lugemine. Võimaldab samaaegsete andmetehingute poolt kirjutatud kinnitamata poolikute andmete lugemist.
Read Committed	CS (CC)	Kinnitatud andmetehingute lugemine. Poolikute andmete lugemine pole võimalik. Oracle ja DB2 (versioonist 9.7 ja edasi) loevad andmete viimase kehtiva versiooni, (DB2-s nimetatakse seda taset "Currently Committed", CC) mõned ABHS-id ootavad vaikimisi kuni samaaegne andmetehing kinnitatakse.
Repeatable Read	RS	Korratav lugemine. Võimaldab lugeda ainult kehtivaid andmeid, mis on korduvalt loetavad ning samaaegsete andmetehingute UPDATE ja DELETE käsud neid ridu muuta ei saa.
Serializable	RR	Järjepidev. Võimaldab lugeda ainult kehtivaid andmeid, mis on korduvalt loetavad ning samaaegsete andmetehingute UPDATE, DELETE ja INSERT käsud neid ridu ja tabeleid muuta ei saa.

Pange tähele erinevusi ISO SQL ja DB2 eraldatuse tasemete nimetustes. DB2 defineeris algselt vaid 2 taset: CS, „Cursor Stability“ ehk sama, mis kinnitatud tegevuste lugemine ja RR, „Repeatable Read“ korratava lugemise kohta. Neid nimetusi pole muudetud olenemata asjaolust, et ISO SQL-is defineeriti 4 taset ning tasemele RR anti teistsugune sisu

Lisaks ISO SQL eraldatuse tasemetele on nii Oracle'is kui ka SQL Server'is kasutusel „Snapshot" nimeline tase. Sellisel juhul näeb klient kogu andmetehingu vältel „mälupilti" andmebaasi olekust nagu see oli andmetehingu alguses, kuid ei näe vahepeal samaaegse andmetehingu poolt muudetud andmeid. **Teadmiseks**, Oracle-is nimetatakse seda nimetusega SERIALIZABLE.

Hiljem vaatame täpsemalt, milliseid eraldatuse tasemeid kasutavad meie poolt käsitletavad ABHS-id ning kuidas neid rakendada. Sõltuvalt ABHS-ist võib eraldatuse taseme määrata vaikimisi tervele andmebaasile, Andmetehingule või sessioonile selle alguses ning mõne konkreetse süsteemi puhul isegi tegevuse või tabeli juures vahetult. Parima tava ning ISO SQL-i järgi soovitatakse andmete eraldatuse tase määrata iga andmetehingu alguses eraldi vastavalt sellele, millist eraldust konkreetse tegevuse juures vaja läheb (kaitse vs. jõudlus). ISO SQL-i kohaselt määratakse Oracle'i ja SQL Server'i eraldatus järgneva käsu abil,

```
SET TRANSACTION ISOLATION LEVEL <eraldatuse tase>
```

kuid näiteks DB2-s hoopis järgnevalt:

```
SET CURRENT ISOLATION = <eraldatuse tase>
```

Olenemata süntaksi ja tasemete nimetuste erinevusest ABHS-ides, ODBC ja JDBC programmeerimisliides tunnistavad vaid ISO SQL eraldatuse tasemete nimetusi. JDBC-pöörduse puhul määratakse eraldatuse tase ühenduse objekti meetodi *setTransactionIsolation* parameetrina:

```
<ühendus>.setTransactionIsolation(Connection.<Andmetehingu eraldatus>);
```

kus `<Andmetehingu eraldatus>` defineeritakse programmeerimiskeele vastavate võtmesõnade abil, näiteks TRANSACTION_SERIALIZABLE tähendab järjepidevat eraldust. Võtmesõna seotakse vastava eraldatuse tasemega ning saadetakse ABHS-ile JDBC draiveri abil.

2.4 Samaaegse täitmise juhtimise mehhanismid

Kaasaegsed ABHS-id kasutavad põhiliselt järgnevaid samaaegse täitmise juhtimise mehhanisme (SJ, inglisekeelse lühendina CC) töödeldavate andmete eraldamiseks:

- Mitmeosaline lukustusskeem[4] (**ML**, Multi-Granular Locking scheme, MGL või LSCC)
- Mitme versiooniline samaaegsuse juhtimine (**MV**, Multi-Versioning Concurrency Control, MVCC)
- Optimistlik samaaegsuse juhtimine (**OJ**, Optimistic Concurrency Control, OCC).

2.4.1 Lukustamisega samaaegse täitmise juhtimine (ML)

Lihtsad lukustusskeemid, mille abil andmebaasi serveri ***lukustuse haldur*** lugemise ja kirjutamise korral automaatselt andmete täielikkust kaitseb, defineerib tabel 2.3. Kuna ühe tabeli rea kohta saab korraga toimuda vaid üks kirjutamine, püüab lukustuse haldur kirjutatavale reale (INSERT, UPDATE või DELETE käsud) seada ekslusiivset ehk monopoolset lukustust (**X-lukustus**). X-lukustus antakse ainult kirjutamiseks kui samale reale või ridade hulgale pole eelnevalt seatud

[4] Kirjanduses kasutavad mitmed autorid lukustusskeemi kohta nimetust „pessimistlik samaaegsuse juhtimine" (PSJ) ja mitmeversioonilise kohta „optimistlik samaaegsuse juhtimine", kuigi tegelikul OJ-l on teistsugune sisu

ühtegi teist lukustust (nagu on näidatud tabelis 2.3) ning **seadmise hetkest jääb X-lukustus kehtima andmetehingu lõpuni**.

Loetavaid ridu (SELECT käsu puhul) kaitseb lukustuse haldur jagatud **S-lukustusega**, mida saab seada samaaegselt mitme erineva kliendi loetavatele ridadele kuna nad ei sega üksteist, kuigi see oleneb andmetehingu eraldatuse tasemest. **`READ UNCOMMITTED`** eraldatuse tase ei vaja lugemise kaitseks S-lukustust, kuid teiste tasemete jaoks on S-lukustus lugemisel vajalik ning siis seatakse see juhul, kui neil ridadel pole ühegi teise andmetehingu poolt lukustust.

Tabel 2.3 S-lukustuse ja X-lukustuse omavahelised seosed.

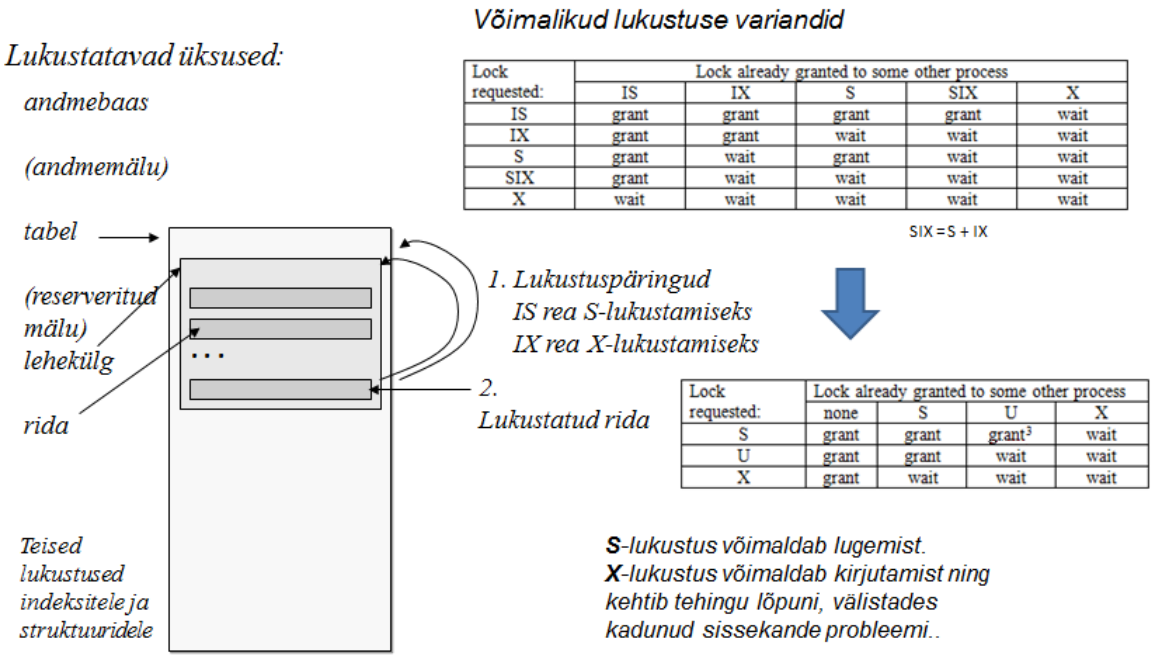
Andmetehing soovib seada reale vastavat lukustust	Teine andmetehing on lukustanud sama rea vastava lukustusega		Ükski teine andmetehing pole seda rida lukustanud
	S-lock	**X-lock**	
S-lock	lukustus seatud	oota vabanemist	lukustus seatud
X-lock	oota vabanemist	oota vabanemist	lukustus seatud

`READ COMMITTED` eralduse korral vabastatakse rida S-lukustusest kohe peale lugemist, kuid **`REPEATABLE READ`** ja **`SERIALIZABLE`** eraldatuse tasemetel jääb S-lukustus kehtima andmetehingu lõpuni. Read vabastatakse automaatselt kõigist lukustustest peale andmetehingu lõppu[5] ning kinnitamist või tühistamist.

> Mõne ABHS-i SQL dialekt sisaldab eraldi seatavat lukustuskäsku LOCK TABLE, kuid ka need lukustused vabastatakse automaatselt peale andmetehingu lõppu ning S-lukustused READ COMMITED eralduse korral varem. Mingit eraldi seatavat vabastuskäsku UNLOCK TABLE SQL-keeles pole, välja arvatud MySQL/InnoDB-s.

Tegelik ABHS-i lukustusskeem on tunduvalt keerukam, rakendades lukustust erinevatele osistele nagu rida, lehekülg, tabel, indeksite vahemik, struktuur jne. ning lisaks S- ja X-lukustusele veel teisigi lukustusrežiime. Realukustuse puhul püüab lukustuse haldur esmalt rakendada soovitud lukustust kõrgema taseme üksustele, et selgitada **mitmeosalise lukustusskeemi (ML)** korral sobivust võimalike teiste andmetehingute poolt eelnevalt lukustatud üksustega nagu on kirjeldatud joonise 2.6 abil. Inglisekeelsetes tabelites on esimese tulbas lukustuse tüüp, mida haldur soovib rakendada ning järgmistes tulpades eelnevalt samale ressursile rakendatud lukustus. Kokkusobivus tabelis viitab „grant“ lukustuse lubamisele ning „wait“ kohustusele oodata, kuni eelnev lukustus on eemaldatud.

[5] Mõnes süsteemis erandiks on WITH HOLD funktsiooniga valitud read, näiteks DB2-s.

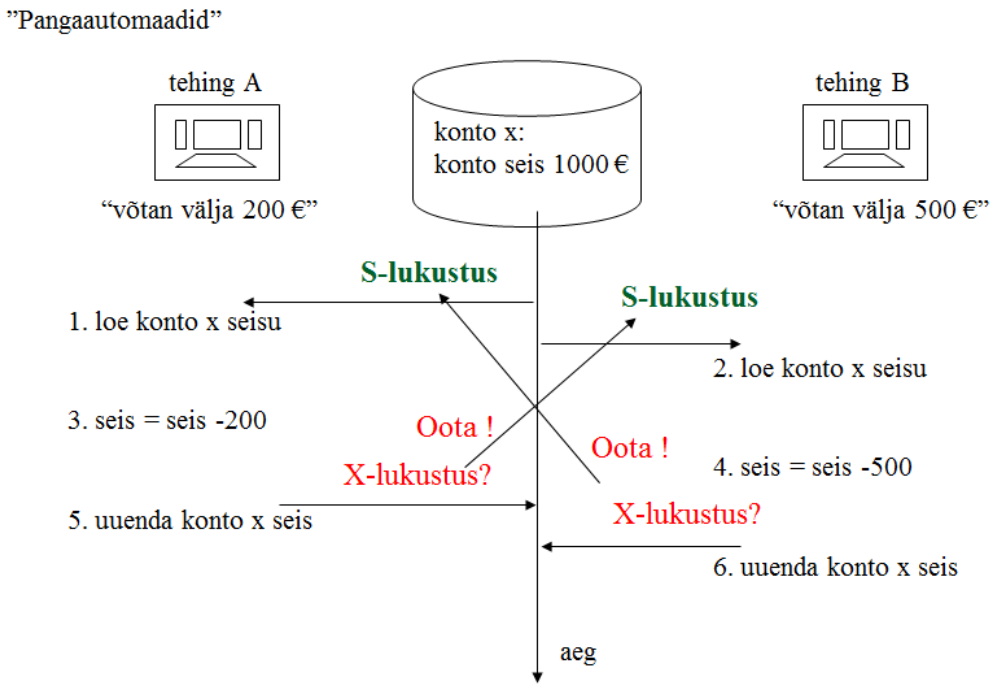


Lock requested:	Lock already granted to some other process				
	IS	IX	S	SIX	X
IS	grant	grant	grant	grant	wait
IX	grant	grant	wait	wait	wait
S	grant	wait	grant	wait	wait
SIX	grant	wait	wait	wait	wait
X	wait	wait	wait	wait	wait

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

Joonis 2.6 Erineva taseme üksuste lukustuste kokkusobivus.

Lukustusprotokoll välistab küll kadunud sissekande probleemi aga kui samaaegne andmetehing kasutab eraldatust, mille puhul S-lukustust hoitakse ridadel andmetehingu lõpuni, viib see probleemini, mida kirjeldab joonis 2.7. Mõlemad andmetehingud jäävad selles olukorras ootama teineteise lõppu ning tekib nn. surnud ring ehk **ummikseis**. Esimestel andmebaasidel oli see põhiline probleem, kuid kaasaegsetel ABHS-idel on sisse ehitatud **ummiku detektor** hargprogramm, mis käivitub korra iga 2 sekundi järel ning otsib süsteemist ummikseise. Leides täidetavate harude seast kirjeldatud olukorra, valib detektor välja ühe ootava andmetehingu ning tühistab selle sunniviisiliselt, võimaldades teisel andmetehingu täitmisega edasi minna.



Joonis 2.7 ML välistab kadunud sissekanded, kuid tekitab ummikseisu.

Tühistatud andmetehingu kliendi rakendus saab serverilt ummikseisu erandi teate ning peaks seetõttu lühikese juhusliku ajahetke pärast uuesti andmetehingu täitmist proovima. Vaata Pangaandmetehingu näites Java koodi „kordus tsükli“ osa **Lisas 2**.

> Tasub meeles pidada, et **ükski ABHS ei saa ise automaatselt korrata sunniviisiliselt tühistatud andmetehingut** ummikseisu korral – see on kliendi rakenduse või vastavat rakendus jooksutava serveri tarkvara ülesanne. Oluline on mõista, et ummikseis ei ole viga ning ühe andmetehingu sunniviisiline peatamine on serveri poolne teenus, mis hoolitseb serveri jõudluse eest selles olukorras.

2.4.2 Mitmeversiooniline samaaegse täitmise juhtimine (MV)

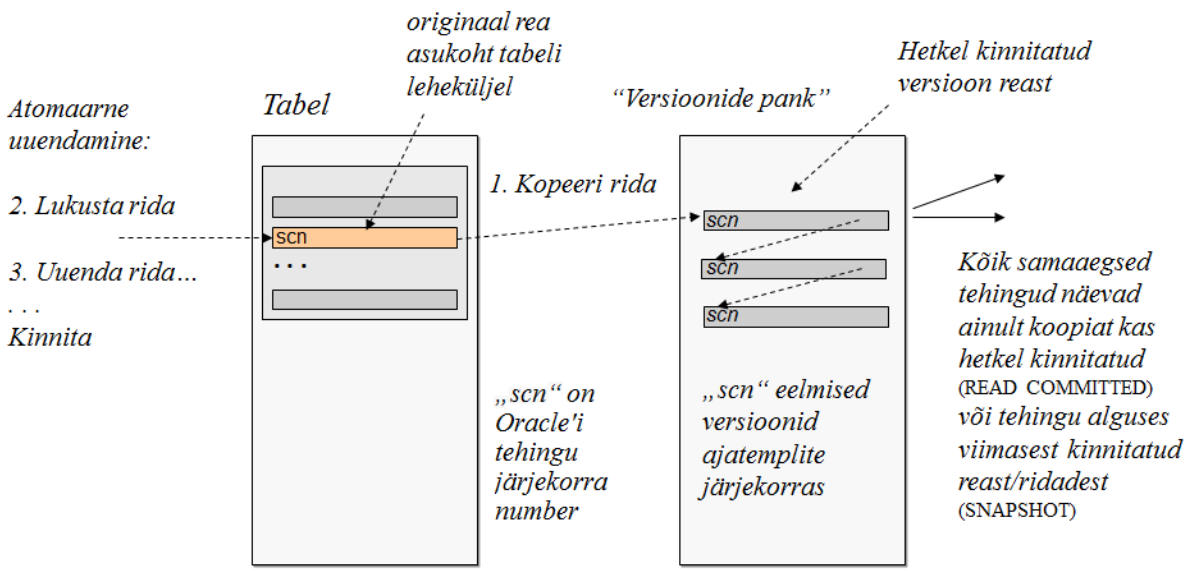
MV puhul säilitab server jada erineva ajatempliga uuendatud verisoone kõigist andmeridadest, seega leidub iga samaaegse andmetehingu jaoks kinnitatud andmete versioon selle andmetehingu algushetkest. See samaaegse täitmise juhtimistehnika välistab ooteajad lugemisel ning pakub kahte eraldatuse taset – kõik **`READ COMMITTED`** tasemega andmetehingud loevad ridadest **viimast kehtivat versiooni** ja **`SNAPSHOT`** tasemega andmetehingud näevad ridade lugemisel **viimast kehtivat versiooni andmetehingu algushetkel** (või enda poolt kirjutatud ridu). „Mälupilt“ ehk „mälukaader“ eraldatuse korral ei saa andmetehing kunagi näha fantoomkirjeid ega ka midagi nende tekkimise vastu ette võtta; samas võimaldab `SERIALIZABLE` eraldus ML mehhanismis fantoomridade kirjutamist välistada. Tegelikult näeb andmetehing selles „mälupildis“ ka neid ridu, mille teine samaaegne andmetehing on vahepeal andmebaasist kustutanud.

Olenemata eraldatuse tasemest on kirjutamine ka MV süsteemides kaitstud mingisuguse realukustusega[6]. Kui proovida uuendada ridu, mille sisu on vahepeal teiste samaaegsete andmetehingute poolt muudetud, saate vastuseks mõne „mälukaader on liiga vana“-tüüpi veateate.

MV rakendamine

Joonis 2.8 kirjeldab MV mehhanismi rakendamist Oracle andmebaasis. Oracle nimetab „mälupilt“ eraldust nimetusega järjepidev (`SERIALIZABLE`).

[6] SolidDB on ABHS, kus MV ei lukusta eelnevalt kirjutamisi; sellel juhul esimene kirjutaja võidab.

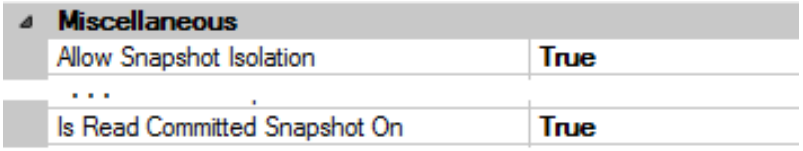


Joonis 2.8 Mitme versiooniline (MV) haldamine põhineb kinnitamiste ajalool.

Oracle'i MV korral seatakse esimese kirjutava (INSERT, UPDATE või DELETE käsud) andmetehingu jaoks reale/ridadele teatav lukustus ning see andmetehing saab end lõpule viia, teised kirjutavad andmetehingud jäetakse ootele. Lukustus seatakse reale süsteemi muudatuse numbri (SCN) abil, mis seotakse vastava andmetehingu numbriga[7] ridade ajaloos. Rida on lukustatud kuni sama numbriga andmetehing on aktiivne. Kirjutamise lukustamine võimaldab ummikseisude teket, kui sunniviisilise tühistamise asemel leiab Oracle koheselt realukustuse, mis ummiku põhjustas, saadab klindi rakendusele vastava erandi teate ning ootab, et ummiku põhjustanud rakendus ise andmetehingu tühistaks ja seda kordaks.

Oracle'i samaaegse täitmise juhtimist nimetatakse samuti hübriidseks, sest lisaks MV-le automaatse realukustusega pakub see eraldi „LOCK TABLE" käsku mis koos „SELECT .. FOR UPDATE" päringuga võimaldab ka ridu lukustada ning välistab samaaegselt fantoomkirjete kirjutamise antud tabelisse. Lisaks saab Oracle'is kasutada **kirjutuskaitsega** (ingl. *read only*) **andmetehinguid**.

Ka Microsofti arendajad on märganud MV eeliseid ning alates 2005 aasta versioonist on SQL Serveris võimalik kasutada ridade versioonilist haldamist, andmebaasi seadistamisel Transact-SQL käskude abil ning alates 2012 aasta versioonist on andmebaasi omadused sellised nagu joonisel 2.9.



Joonis 2.9 SQL Server 2012 „mälupiltide" toetuse seadistamine

MySQL/InnoDB samaaegse täitmise juhtimise mehhanism on oma olemuselt tõeline hübriid-samaaegsuse juhtimisega, pakkudes 4 eraldatuse taset lugemistele:

[7] Tehingu järjekorra number XSN SQL Server MV rakenduses (Delaney 2012)

- READ UNCOMMITTED - kinnitamata andmete lugemine ilma S-lukustuse ja MV-ta;
- READ COMMITTED - viimati kinnitatud andmete lugemine MV-ga;
- REPEATABLE READ - korratav lugemine (tegelikult „mälupilt“) MV-ga;
- SERIALIZABLE - järjepidev lugemine ML-ga kasutades S-lukustust, mis väldib fantoomridu.

2.4.3 Optimistlik samaaegse täitmise juhtimine (OJ)

Algses OJ-s hoiti kõik muudatused andmebaasist lahus kuni kinnitamise hetkeni. Sellist juhtimist on rakendatud näiteks Lääne-Šoti Ülikooli (UWS) poolt loodud Pyrrho ABHS-is. Ainuke automaatne eraldatuse tase Pyrrho ABHS-is on järjepidev (SERIALIZABLE) tase (http://**www.pyrrhodb.com**).

2.4.4 Kokkuvõte

ISO SQL standard on välja töötatud ANSI poolt ning põhineb algselt DB2 SQL-keele versioonil, mille õigused IBM ANSI-le andis. DB2 samaaegse täitmise juhtimismehhanismina kasutatakse mitmeosalist lukustamist, millel tol ajal oli vaid 2 taset: kinnitatud andmete lugemine (CS) ja korratav lugemine (RR). See asjaolu on tinginud uute eraldatuse tasemete (vt. tabel 2.2) defineerimise ANSI/ISO SQL-is, mida võiks terminoloogiliselt nimetada jagatud lukustamiseks.

Samas ei ütle SQL standard midagi eraldatuse tasemete sisemiste mahhanismide kohta, nii on mitmeid erinevaid samaaegse täitmise juhtimise algoritme koondatud sama nimetuse alla, kuigi nad käsitlevad juhitavaid andmeid mõnevõrra erinevalt, nagu oleme erinevatest näidetest juba tähele pannud. Järgnevas tabelis 2.4 on alampealkirja "Isolation Levels" all võrreldud ANSI/ISO SQL-i eraldatuse tasemete nimetusi ja erinevate ABHS-ide vastavaid tasemete rakendusi (tasemed – viimati kinnitatud lugemine (*read latest committed*) ja „mälupilt“ (*snapshot*) on lisatud tabeli autori poolt), mis aga omavad sageli teisi segadust tekitavaid ning omavahel nihkes või vahetuses olevaid nimetusi. Artiklis "A Critique of ANSI SQL Isolation Levels" (Berenson et al, 1995) tunnistas ANSI/ISO SQL töögrupp seda kui tõsist probleemi.

Praktiliste harjutuste ülesanne 2.7 näitab mõningaid probleeme erinevate ABHS-ide juures seoses „mälupilt“ eraldatusega kirjutamisel, sest kirjutamine rikub „mälupildi“ sisu. Seega saab mälupilti turvaliselt kasutada vaid hetkeliste raportide koostamisel. Tabel 2.4 koondab endasse ka teisi ABHS-ide vahelisi erinevusi.

Tabel 2.4 ISO SQL-i ja erinevate ABHS-ide poolt toetatud funktsionaalsus.

	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	

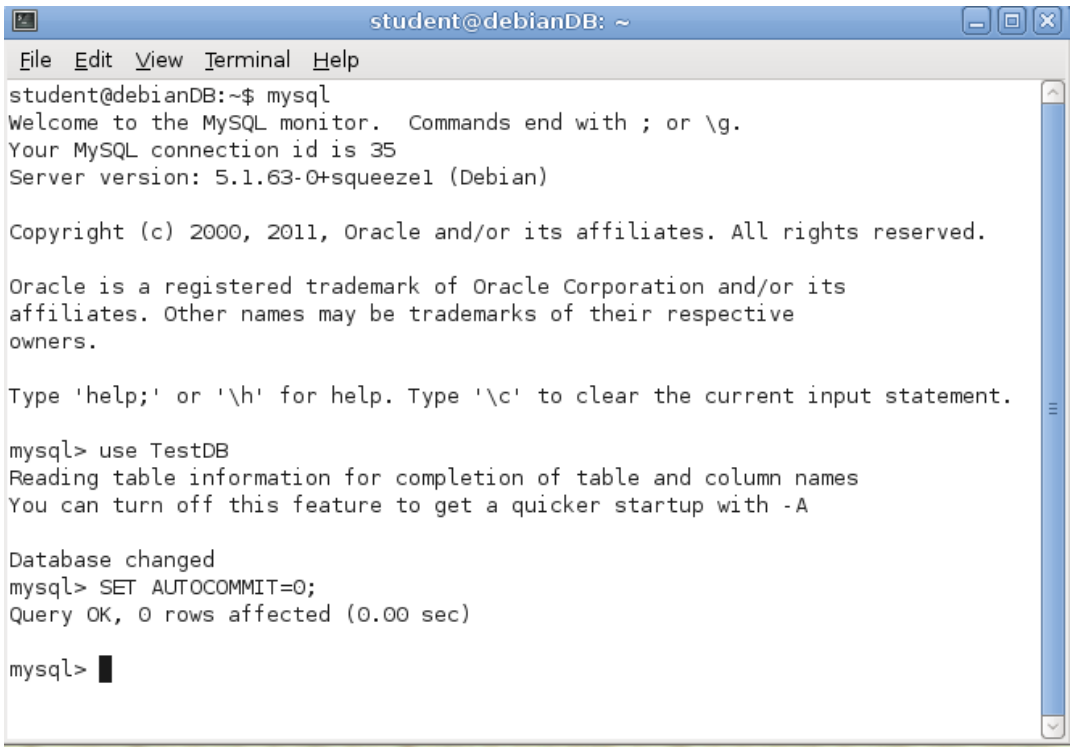
Mälupilt (*snapshot*) tähendab täielikku andmebaasi „vaadet“ andmetehingu alguse seisuga. Sellepärast sobibki see ideaalselt ainult lugemise käske sisaldavate andmetehingute jaoks, kuna nii ei blokeerita teiste samaaegsete andmetehingute täitmist.

Mälupildi kasutamine ei välista fantoomkirjete olemasolu, sest need lihtsalt ei sisaldu selles „pildis“. MV-d kasutavas ABHS-is ennetatakse fantoomkirjete esinemist SQL järjepideva eralduse tabeli-taseme lukustuse rakendamisega (näiteks Oracle ja PostgreSQL). Mälupilt eraldatuse korral lubatakse kõigis ABHS-ides INSERT käsu kasutamist, kuid tegevus teiste kirjutamiste korral on erinev.

Tabelist 2.4 näeme, et DB2 on meie DebianDB-s ainuke ABHS, mis ei toeta mälupilt-tüüpi eraldatuse taset.

2.5 Praktilised harjutused

Uut MySQL sessiooni alustatakse tavalisel moel (joonis 2.10):

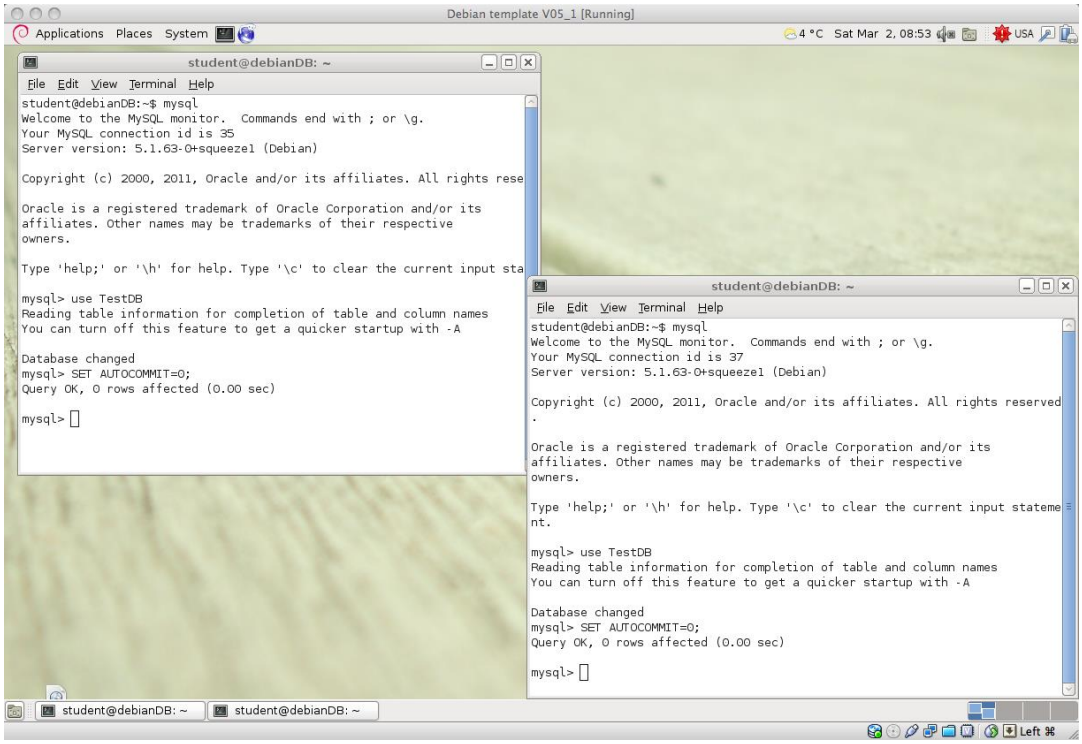


Joonis 2.10 MySQL sessiooni alustamine

Ülesanne 2.0

Samaaegse täitmise juhtimise katsetamiseks tuleb käivitada veel üks MySQL sessioon teises terminali aknas (joonis 2.11). Olgu eksraanil kõrvuti asuvatest akendest vasakpoolne sessioon A ja parempoolne sessioon B. Mõlema sessiooniga siseneme andmebaasi TestDB ja lülitame välja AUTOCOMMIT režiimi:

```
-----------------------------------
use TestDB
SET AUTOCOMMIT = 0;
-----------------------------------
```



Joonis 2.11 Samaaegsed MySQL sessioonid DBTechNet virtuaallaboris

Nagu näha, võivad samaaegsed andmetehingud üksteist blokeerida, sellepärast peavad koostatud andmetehingud olema nii lühikesed ja konkreetsed kui võimalik. Andmetehingu jooksul lõppkasutaja sekkumist nõudvad dialoogid võivad kaasa tuua lõputud ooteajad seega **ei tohi SQL andmetehing anda juhtimist üle kasutajaliidesele enne andmetehingu lõppu**.

Ükski ABHS ei korda automaatselt andmetehingut kui see on ummikseisu olukorras tühistatud sunniviisiliselt. Andmetehingu kordamise organiseerimine on kliendi rakenduse või seda jooksutava serveri ülesanne. Oluline on mõista, et ummikseis ei ole viga ning ühe andmetehingu sunniviisiline peatamine on serveri poolne teenus, mis hoolitseb serveri jõudluse eest selles olukorras.

Soovitus!

Niipea, kui arvame, et ummikseisud ja mehhanismid, mille abil ABHS neid töötleb, aitavad meil võidelda pikkade ooteaegade peamise põhjusega …

On selge, et samaaegselt täidetavad andmetehingud võivad üksteist blokeerida, seepärast peab andmetehingud koostama nii lühikeste ja konkreetsetena, kui võimalik.
Lõppkasutaja kaasamine andmetehingu dialoogi toob kaasa lõputult pikad ooteajad, seepärast **ei tohi SQL andmetehingu juhimist anda üle kasutajaliidesele**.
Rakenduse programne loogika peab tagama nendest keeldudest kinni pidamise SQL andmetehingute täitmisel.

Kontrollime süsteemis kehtivat (vaikimisi) eraldatuse taset SELECT käsu abil:

```
-----------------------------------
SELECT @@GLOBAL.tx_isolation, @@tx_isolation;
-----------------------------------
```

MySQL on vaikimisi nii globaalselt, kui ka sessiooni põhiselt korratava lugemise (REPEATABLE READ) tasemel.

Kindluse mõttes kustutame eelmise kontode tabeli ning loome uue, millesse sisestame 2 rida andmeid:

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

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

Ülesanne 2.1

Nagu eelnevalt selgitasime, ilmneb **kadunud sissekande** probleem, kui samaaegselt täidetav andmetehing kirjutab üle esimese andmetehingu andmed enne selle lõppu. Kõik kaasaegsed ABHS-id suudavad selle olukorra samaaegse täitmise juhtimise abil välistada. Kuid peale esimese andmetehingu kinnitamist võib teine hooletult koostatud andmetehing selle lugemata ülekirjutada. Seda kutsutakse **pimedaks ülekirjutamiseks** ning seda on äärmiselt lihtne esile kutsuda.

Pime ülekirjutamine toimub näiteks siis, kui rakendus loeb andmebaasis väärtuse, uuendab oma mälus seda ja kirjutab uue väärtuse andmebaasi tagasi. Järgnevas tabelis simuleerime lokaalseid muutujaid kasutavat rakendust. Lokaalne muutuja on lihtsalt märgend käimasoleva sessiooni mälus, millele viitab @-märk muutujanime ees ning seda võib kasutata SQL käskude sees kuni ta omab skalaarset ehk ühemõõtmelist väärtust.

Tabel 2.5 sisaldab katset samaaegsete sessioonidega (A ja B, kumbki eraldi tulbas). Et katse „õnnestuks“ sisaldab esimene tulp tegevuste järjekorda. Eesmärgiks on kadunud sissekande simuleerimine (joonis 2.2): sessiooni A andmetehing võtab kontolt „101“ välja 200 eurot, sessiooni B Andmetehing võtab samalt kontolt 500 eurot ning kirjutab konto seisu üle (info eelmise andmetehingu kohta läheb kaduma). Seega pimeda ülekirjutamise tulemus on samaväärne kadunud sissekandega.

> 4. sammul tuleb meeles pidada, et MySQL-i vaikimisi lukustus kestab 90 sekundit. Seega andmetehing A peaks jätkama 5. sammuga koheselt peale Andmetehingu B 4. sammu sooritamist.

Esmalt taastame tabeli Accounts esialgse sisu:

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

```
COMMIT;
-----------------------------------
```

Tabel 2.5 Pimeda ülekirjutamise simuleerimine, kasutatakse lokaalseid muutujaid

	Sessioon A	Sessioon B

1	`SET AUTOCOMMIT = 0;`
	`SET TRANSACTION`
	`ISOLATION LEVEL READ COMMITTED;`
	
	`-- välja võetava raha hulk`
	`SET @amountA = 200;`
	`SET @balanceA = 0;  -- algväärtus`
	
	`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;`
		
		`-- välja võetava raha hulk`
		`SET @amountB = 500;`
		`SET @balanceB = 0;  -- algväärtus`
		
		`SELECT balance INTO @balanceB`
		`FROM Accounts WHERE acctID = 101;`
		
		`SET @balanceB = @balanceB - @amountB;`

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

4		`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;`

Samaaegse (siin õigemini vahelduva) täitmise korral andmetehingute A ja B poolt tabelis 2.4, A valmistub 1. sammul välja võtma 200 eurot (andmebaasi sisu pole veel muudetud), B valmistub 2. sammul välja võtma 500 eurot, A uuendab 3. sammul andmebaasi sisu, B uuendab 4. Sammul andmebaasi sama rea sisu, A kontrollib enne kinnitamist 5. sammul, kas konto seis on selline nagu ta peaks olema ning B teeb 6. sammul sama.

Küsimused

- Kas süsteem käitus nii nagu oli ette arvata?
- Kas on märke kadunud andmetest?

Kõik ABHS-id rakendavad samaaegse täitmise juhtimist nii, et ühelgi eraldatuse tasemel ei tekiks kadunud sissekande anomaaliat. Samas on alati võimalus, et hooletult kirjutatud kood võib põhjustada pimeda ülekirjutamise ning tulemuseks on samuti vigased andmed baasis. Praktikas oleks tegemist justkui näiliselt turvalise programmiga, mille turvaaugust pääseb sisse see, mida me iga hinna eest eemal hoida oleme püüdnud.

Ülesanne 2.2a

Kordame ülesannet 2.1, kuid seekord järjepideva eraldusega.

Esmalt taastame tabeli Accounts esialgse sisu:

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

Tabel 2.6a Ülesande 2.1 stsenaarium S-lukustusega

	Sessioon A	Sessioon B

1	`SET AUTOCOMMIT = 0;`
	`SET TRANSACTION`
	`  ISOLATION LEVEL SERIALIZABLE;`
	
	`-- välja võetava raha hulk`
	`SET @amountA = 200;`
	`SET @balanceA = 0;  -- algväärtus`
	
	`SELECT balance INTO @balanceA`
	`FROM Accounts WHERE acctID = 101;`
	
	`SET @balanceA = @balanceA - @amountA;`
	
	`SELECT @balanceA;`	

2		`SET AUTOCOMMIT = 0;`
		`SET TRANSACTION`
		`  ISOLATION LEVEL SERIALIZABLE;`
		
		`-- välja võetava raha hulk`
		`SET @amountB = 500;`
		`SET @balanceB = 0;  -- algväärtus`
		
		`SELECT balance INTO @balanceB`
		`FROM Accounts WHERE acctID = 101;`
		
		`SET @balanceB = @balanceB - @amountB;`

3	UPDATE Accounts SET balance = @balanceA WHERE acctID = 101;	
4		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;**

Küsimused

a) Millis(t)ele järeldus(t)ele jõudsite?
b) Mis oleks juhtunud, kui järjepidev eraldus oleks mõlema andmetehingu puhul asendatud korratava lugemise eraldusega?

MySQL rakendab korratava lugemise taset MV kaudu ja järjepidevat taset ML kaudu.

Ülesanne 2.2b

Kordame ülesannet 2.2a, kuid nüüd kasutame „tundlikku uuendust“ SELECT – UPDATE käskude abil ilma lokaalsete muutujateta.

Esmalt taastame tabeli Accounts esialgse sisu:

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

Table 2.6b SELECT – UPDATE võidujooks koos „tundlike uuendustega“

	Sessioon A	**Sessioon 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		UPDATE Accounts SET balance = balance – 500 WHERE acctID = 101;
5	SELECT acctID, balance FROM Accounts WHERE acctID = 101; **COMMIT;**	
6		SELECT acctID, balance FROM Accounts WHERE acctID = 101; **COMMIT;**

Küsimus

- Millis(t)ele järeldus(t)ele jõudsite?

Ülesanne 2.3

UPDATE – UPDATE võidujooks kahe erineva konto juures ning erinevas järjekorras.

Esmalt taastame tabeli Accounts esialgse sisu:

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

Tabel 2.7 UPDATE - UPDATE stsenaarium

	Sessioon A	**Sessioon 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;**

Küsimus

- Millis(t)ele järeldus(t)ele jõudsite?

Andmetehingu eraldatuse tasemel pole mingit mõju viimasele stsenaariumile, kui alati on kasulik defineerida iga andmetehingu alguses sobiv tase. Alati on võimalik ka varjatud protsesside olemasolu näiteks võõrvõtmete kontrollimised või trigerid, mis kasutavad lugemist. Trigerite loomine on aga andmebaasi administraatorite problem ning ei kuulu selles juhendis arutamisele.

Ülesanne 2.4

Andmetehingute anomaaliate teema jätkuks proovime esile kutsuda poolikute andmete lugemist. Andmetehing A töötab MySQL vaikimisi korratava lugemise tasemel, andmetehing B töötab kinnitamata andmete lugemise tasemel:

Esmalt taastame tabeli Accounts esialgse sisu:

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

Tabel 2.8 Poolikute andmete lugemise probleem

	Sessioon A	Sessioon 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;**	

Küsimused

a) Millis(t)ele järeldus(t)ele jõudsite?

b) Mis oleks juhtunud, kui kinnitamata andmete lugemise tase oleks asendatud kinnitatud andmete lugemise tasemega andmetehingus B?

c) Mis oleks juhtunud, kui kinnitamata andmete lugemise tase oleks asendatud korratava lugemise tasemega andmetehingus B?

d) Mis oleks juhtunud, kui kinnitamata andmete lugemise tase oleks asendatud järjepideva tasemega andmetehingus B?

Ülesanne 2.5

Järgmisena vaatame hägusa lugemise anomaaliat:

Esmalt taastame tabeli Accounts esialgse sisu:

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

Tabel 2.9 Hägusa lugemise probleem

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

2		SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL REPEATABLE READ;** UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101; UPDATE Accounts SET balance = balance + 500 WHERE acctID = 202; **COMMIT;**
3	-- Kordame sama päringut SELECT * FROM Accounts WHERE balance > 500; **COMMIT;**	

Küsimused

a) Kas andmetehing A loeb 1. ja 3. sammul sama tulemuse?

b) Mis juhtub, kui seada andmetehing A korratava lugemise eraldatuse tasemele?

Ülesanne 2.6

Proovime tekitada klassikalist „raamatu näidet“ fantoomkirjete sisestamisest:

Esmalt taastame tabeli Accounts esialgse sisu:

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

Tabel 2.10 Fantoomkirje probleem

	Sessioon A	Sessioon B
1	SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL REPEATABLE READ;** SELECT * FROM Accounts WHERE balance > 1000;	
2		SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL READ COMMITTED;**

		INSERT INTO Accounts (acctID, balance) VALUES (303,3000); **COMMIT;**
3	-- Kas me näeme uut kontot 303? SELECT * FROM Accounts WHERE balance > 1000; **COMMIT;**	

Küsimused

a) Kas andmetehing B peab ootama andmetehingu A järel?

b) Kas andmetehingu B poolt sisestatud acctID = 303 konto on andmetehingule A nähtav?

c) Kas sammude 2 ja 3 järjekorra muutmine muudab loetud tulemust?

d) MySQL/InnoDB kasutab korratava lugemise eraldatuse tasemel mitmeversioonilist juhtimist, kuid mis on sellise versioonimise mõte?

e) Mida same teada järgnevast testist?

```
-----------------------------------
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ;
ERROR 1568 (25001): Transaction characteristics can't be changed while a
transaction is in progress

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

mysql> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ;
Query OK, 0 rows affected (0.00 sec)

mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)
-----------------------------------
```

Ülesanne 2.7

Uurime “mälupildi” eraldatuse taset ja teist tüüpi fantoomkirjeid

Esmalt loome uue tabeli:

```
-----------------------------------
DROP TABLE T;

-- uus testimise tabel
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, 'forth', 2);
INSERT INTO T (id, s, i) VALUES (5, 'to be or not to be', 1);

COMMIT;
-----------------------------------
```

Tabel 2.11 Sisestamise ja uuendamise fantoomkirjed, kustutatud rea uuendamine

	Sessioon A	**Sessioon 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;`
		
		`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	`-- Korda päringut ja uuenda`
	`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 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;`	

4		**`COMMIT;`**

5	`SELECT * FROM T WHERE i = 1;`
	
	`UPDATE T`
	`SET s = '`**`updated after delete?`**`'`
	`WHERE id = 5;`	

	SELECT * FROM T WHERE i = 1;	
6	**COMMIT;** SELECT * FROM T;	

Küsimused

a) Kas andmetehingu B poolt tehtud INSERT ja UPDATE on andmetehingu A poolt nähtavad?

b) Mis juhtub, kui andmetehing A proovib uuendada andmetehingu B poolt uuendatud rida 2?

c) Mis juhtub, kui andmetehing A proovib uuendada andmetehingu B poolt kustutatud rida 5?

d) Võrdle tulemusi sarnaste stsenaariumitega SQL Serveri Oracle „mälupiltide“ korral.

e) Kuidas oleks võimalik välistada fantoomkirjete tekkimist kui andmetehing A on aktiivne?

SELECT käsk tekitab MySQL/InnoDB-s korratava lugemise tasemel töötava andmetehingu korral täieliku „mälupildi“. Kui aga andmetehing muuda tabeli ridu, pole see „mälupilt“ enam täielik.

Nimekiri 2.1

Näidistulemused ülesandest 2.7

```
mysql> -- 5. samm
mysql> 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 |
+----+-----------------------------+------+
4 rows in set (0.01 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 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> -- 6. samm
mysql> COMMIT;
Query OK, 0 rows affected (0.02 sec)

mysql> 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 |
+----+------------------------------+------+
6 rows in set (0.00 sec)
```

Lisa 1 lõpuks on meil ülesande 2.7 SQL Server 2012 variant võrdlemiseks.

3 Näitelahendusi

Kasutaja tegevuse täitmiseks on sageli vaja korduvalt andmebaasi poole pöörduda. Mõned andmetehingud on mõeldud selle tegevuse jaoks andmebaasist ainult andmeid koguma, viimane samm kasutajaliideses on tavaliselt mingi salvestusnupu vajutus, mis käivitab andmebaasi sisu uuendava andmetehingu.

SQL andmetehing võib sama kasutaja tegevuse kosseisus omada erinevaid usaldusväärsus ja eraldatuse nõudeid. Sellest lähtuvalt tuleks alati andmetehingu alguses defineerida selle eraldatuse tase.

SQL standardi kohaselt võib kinnitamata andmete lugemise (READ UNCOMMITTED) isoleerituse tase kasutada vaid lugemise (READ ONLY) andmetehingutes (Melton and Simon 2002), kuid ABHS-id seda nõuet ei kontrolli.

Kuna ABHS-id erinevad üksteisest samaaegse täitmise juhtimise ja andmetehinguhalduri käitumise poolest, peab tarkvara arendaja usaldusväärsuse ja hea jõudluse tagamiseks tundma kasutatava ABHS-i käitumist.

Usaldusväärsus on prioriteet number 1, seejärel tuleb jõudlus jt. näitajad, kuid vaikimisi rakendatav eraldatuse tase on tavaliselt valitud just jõudlust silmas pidades! Sobiv eraldatuse tase tuleb hoolikalt valida, kui programmeerija ei suuda otsustada, milline tase piisavat usaldusväärsust võiks tagada, oleks järjepidev eraldus üldiselt kindla peale minek. Oluline on mõista, et „mälupilt“ eraldatus garanteerib vaid täielikud tulemused päringule, kuid ei säilita andmebaasi sisu. Kui loodavas rakenduses ei saa lubada fantoomkirjete esinemist, kuid ABHS toetab vaid „mälupilti“, tuleb uurida eraldi lukustamise võimalusi.

SQL andmetehing ei tohiks sisaldada ühtegi dialoogi lõppkasutajaga, kuna see mõjub drastiliselt andmetöötluskiirusele. Kuna SQL andmetehing on oma täitmise ajal tühistatav, ei tohiks see mõjutada midagi peale andmebaasi. Samuti peaksid SQL andmetehingud olema **nii lühikesed, kui võimalik**, et vähendada samaaegse täitmise võidujookse ning blokeerimisi erinevate andmetehingute vahel.

Vältige andmekirjelduskeele käske (näiteks CREATE ja DROP) andmetehingutes, sest nende automaatsed täitmised võivad kaasa tuua täiendavaid soovimatuid andmetehinguid.

Igal SQL andmetehingul peaks olema täpselt defineeritud eesmärk ning algus ja lõpp samas rakenduse osas. Selles juhendis me ei käsitle SQL andmetehinguid **salvestatud protseduurides**, sest need on igas ABHS-is erinevad. Mõned ABHS-id meie DebianDB-s ei luba COMMIT käsu kasutamist salvestatud protseduurides, kuid andmetehingud võivad saada sunniviisiliselt tühistatud ka salvestatud protseduuride täitmisel ning selle käitlemise eest peab hoolitsema rakenduse kood.

Tehnilises kontekstis on SQL andmetehing üksik andmebaasi ühendus. Kui andmetehing nurjub samaaegse täitmise konflikti tõttu, peaks rakenduse koodis olema **kordus tsükkel** piiratud umbes 10 korrale. Kui aga andmetehing sõltub mõne varasema päringu tulemustest, kuid andmebaasi sisu on vahepeal muudetud ning andmetehing ei suuda vastavat infot uuendada, ei tohiks kordus tsüklit olla ning kogu tegevuse kordamise võimalus tuleks anda kasutajale otsustada. Seda teemat käsitleme meie „RVV avaldises“.

Kui võrguühendus saab häiritud, tuleb SQL andmetehingu kordamiseks kindlasti alustada uus dialoog andmebaasiga.

Lisalugemist, viiteid ja materjale

Kasutatud allikad

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.
Tasuta alla laetav paljudelt lehekülgedelt.

Delaney, K., “SQL Server Concurrency – Locking, Blocking and Row Versioning”, Simple Talk Publishing, juuli 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. Saadaval aadressilt **http://www.opengroup.org/onlinepubs/9695959099/toc.pdf**

Valitud DBTechNet avaldised

Rohkem infot DB2, Oracle’i ja SQL Serveri samaaegse täitmise juhtimise kohta leiate "samaaegse täitmise avaldisest" aadressilt

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

Lisa infot SQL Andmetehingute ja optimistliku lukustuse (RVV ehk ”optimistic locking”) kohta erinevates pöördustes leiate "RVV avaldisest“ aadressilt

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

Virtuallabor ja alla laetavad materjalid

1. Virtuaallabori "DebianDB" OVA fail

   http://www.dbtechnet.org/download/DebianDBVM05.zip (3.9 GB, MySQL 5.1)

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

2. "Kiirjuhend" virtuaallabori ABHS-ide kasutamiseks

   http://www.dbtechnet.org/download/QuickStartToDebianDB.pdf

3. Skriptid Lisa 1 DB2, Oracle’i, MySQLi ja PostgreSQLi katsetamisteks

   http://www.dbtechnet.org/download/SQL_Transactions_Appendix1.zip

Lisa 1 SQL Serveri andmetehingute katsetamine

Lahendame edasi seda „Andmetehingute müsteeriumi" oma lemmik ABHS-is ning katsetame ja vaatame, kas selles juhendis selgitatu on meile selgeks saanud. ABHS-id käituvad erinevalt andmetehingute täitmisel, see võib üllatada programmeerijat, kuid ka tema kliente, kui ta pole kursis nende erinevustega. Kui teil on aega ja peale hakkamist, et katsetada rohkem kui ühte ABHS-i, saate kindlasti laiema pildi turul olevatest süsteemidest.

Virtuaallabori DebianDB jaoks loodud skriptid leiate aadressilt:

http://www.dbtechnet.org/download/SQL_Transactions_Appendix1.zip

Selles lisas esitleme oma katsetusi SQL Server Express 2012 keskkonnas. Kuna SQL Server töötab vaid Window platvormil ning pole saadaval DebianDB keskkonnas, näitame oma ülesannete tulemusi, mida saate võrrelda oma tulemustega. Kui soovite meie tulemusi kontrollida, saate Microsofti leheküljelt tasuta alla laadida SQL Server Express 2012 tarkvarapaketi, mis töötab Windows 7 või uuemate versioonide peal.

Järgnevates katsetustes kasutame SQL Server Management Studio (SSMS) programmi. Esmalt loome uue andmebaasi „TestDB", kasutades vaikeseadeid ning alustame USE käsuga selle andmebaasi kasutamist oma SQL sessioonis.

```
-----------------------------------
CREATE DATABASE TestDB;
USE TestDB;
-----------------------------------
```

1.osa Üksiku Andmetehingu katsed

SQL Server on vaikimisi AUTOCOMMIT režiimis ja kasutab eraldi viitamisega andmetehinguid, seega same kasutada mitmest käsust koosnevaid andmetehinguid. Samas saab serverit seada ka automaatsete andmetehingute režiimi. Üksikut SQL sessiooni saab samuti automaatsete andmetehingute režiimi viia järgneva käsuga

```
SET IMPLICIT_TRANSACTIONS ON;
```

mis kehtib kuni sessiooni lõpuni ning mida saab välja lülitada järgneva käsuga:

```
SET IMPLICIT_TRANSACTIONS OFF;
```

Alustame katsetamist ühe andmetehinguga. Esialgu näiteme ka lihtsamaid tulemusi:

```
-----------------------------------------------------------------
-- Ülesanne 1.1
-----------------------------------------------------------------
--  Autocommit režiim
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; -- mis juhtub?
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;  -- eraldi deklareeritud Andmetehingu algus

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)
```

Deklareeritud andmetehingus ROLLBACK käsk toimis, kuid nüüd oleme tagasi AUTOCOMMIT režiimis!

```
-----------------------------------------------------------------
-- Ülesanne 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
3           third                          NULL
(2 row(s) affected)

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


-----------------------------------------------------------------
-- Ülesanne 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)


-----------------------------------------------------------------
-- Ülesanne 1.4
-- DDL tähendab andmekirjelduskeelt. Sellised käsud nagu
-- CREATE, ALTER ja DROP on DDL käsud.
-- Katsetame DDL käskude kasutamist Andmetehingus!
-----------------------------------------------------------------
SET IMPLICIT_TRANSACTIONS ON;
INSERT INTO T (id, s) VALUES (2, 'will this be committed?');
CREATE TABLE T2 (id INT);  -- Katsetame DDL käsu kasutamist Andmetehingus!
INSERT INTO T2 (id) VALUES (1);
SELECT * FROM T2;
ROLLBACK;
GO  -- GO märgib serverile saadetava SQL käskude paketi lõppu
(1 row(s) affected)

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

SELECT * FROM T;    -- Mis juhtus tabeliga T ?
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
(1 row(s) affected)

SELECT * FROM T2;   -- Mis juhtus tabeliga T2 ?
Msg 208, Level 16, State 1, Line 2
Invalid object name 'T2'.
```

```
-----------------------------------------------------------------
-- Ülesanne 1.5a
-----------------------------------------------------------------
DELETE FROM T WHERE id > 1;
COMMIT;
-----------------------------------------------------------------
-- Kontrollime, kas viga toob SQL Serveris kaasa automaatse tühistamise.
--   @@ERROR on SQLCode indikaator Transact-SQL-is ja
--   @@ROWCOUNT on viimati muudetud ridade indikaator
-----------------------------------------------------------------
INSERT INTO T (id, s) VALUES (2, 'The test starts by this');
(1 row(s) affected)

SELECT 1/0 AS dummy;   -- nulliga jagamine peaks nurjuma!
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;  -- uuendame olematut rida
(0 row(s) affected)

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

DELETE FROM T WHERE id = 7777;   -- proovime kustutada olematut rida
(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?')
```

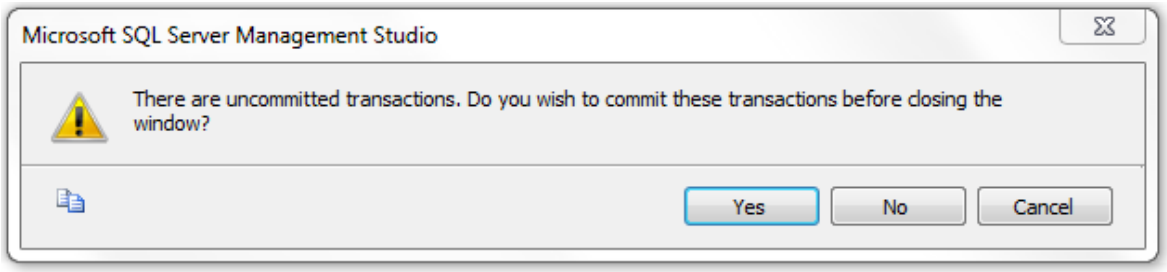
```
INSERT INTO T (id, s, si) VALUES (4, 'Smallint overflow for 32769?', 32769);
INSERT INTO T (id, s) VALUES (5, 'Is the transaction still active?');
SELECT * FROM T;
COMMIT;
GO

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;


-----------------------------------------------------------------------
-- Ülesanne 1.5b
-- See on spetsiaalne näide ainult SQL Serverile!
-----------------------------------------------------------------------
SET XACT_ABORT ON;  -- režiim, milles viga põhjustab automaatse tühistamise
SET IMPLICIT_TRANSACTIONS ON;

SELECT 1/0 AS dummy;   -- nulliga jagamine

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; -- selles režiimis ei põhjusta viga automaatset tühistamist

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

```
2           The test starts by this        NULL
(2 row(s) affected)

-- mis juhtus andmetehinguga?


-----------------------------------------------------------------
-- Ülesanne 1.6   Katsetame andmetehingute loogikat
-----------------------------------------------------------------
SET NOCOUNT ON;  -- lülitame "n row(s) affected" teated välja
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;

-- proovime panga ülekande näidet
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;

-- Testime, kas CHECK piirang üldse töötab:
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;

-- kasutame Transact-SQL-i IF lauset
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.

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;

-- Kui prooviks kasutada olematut kontot?
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;

-- Parandame olukorra Transact-SQL-i IF lausega
SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000
```

```
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


-----------------------------------------------------------------------
-- Ülesanne 1.7   Katsetame andmebaasi taastamist
-----------------------------------------------------------------------
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
```

Kui proovime praegu **väljuda SQL Server Management Studio-st**, saame järgmise hoiatuse:



ning meie katse huvides valime "No" variandi.

Taaskäivitades Management Studio ja ühendudes andmebaasiga TestDB, saame kontrollida meie viimase kinnitamata jäänud andmetehingu tulemust, pärides tabeli T sisu.

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

2.osa Katsetused samaaegsete andmetehingutega

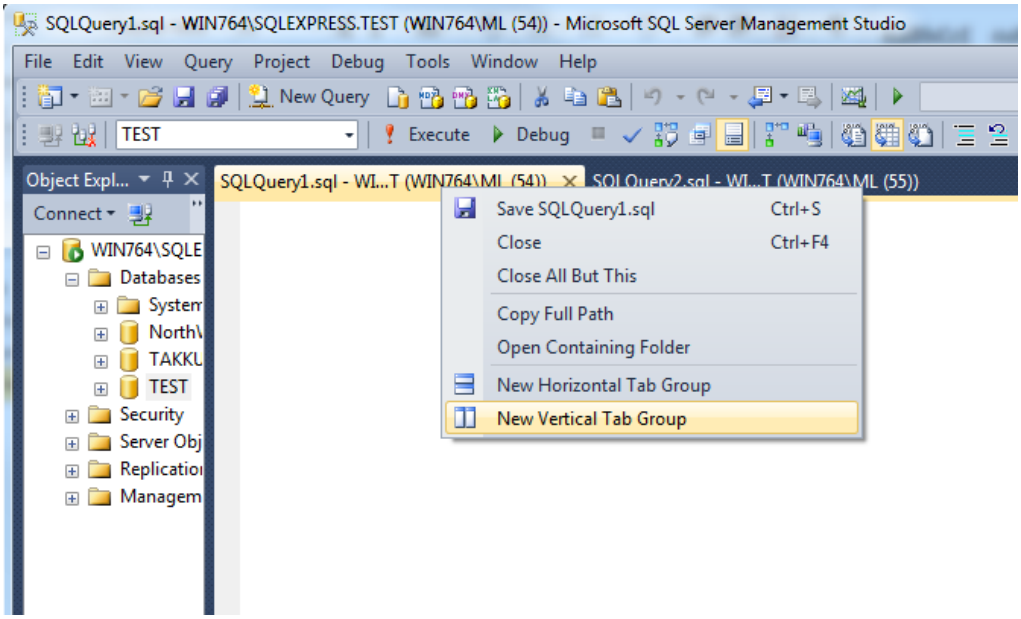
Samaaegse täitmise katsetamiseks avame kaks paralleelset SQL sessiooni „klient A“ ja „klient B“ ning ühendume sama andmebaasiga TestDB. Mõlemas sessioonis määrame tulemused teksti kujul esitatavateks järgnevast menüüst

Query > Results To > Results to Text

ning määrame mõlemad sessioonid automaatselt andmetehinguid algatama

```
SET IMPLICIT_TRANSACTIONS ON;
```

Kasutamaks paremini ära Management Studio visuaalset poolt, asetame paralleelsed SQLQuery aknad vertikaalselt kõrvuti, klõpsates parema klahviga päringu akna päisel ning valides "New Vertical Tab Group" vaate (vt. joonis L1.1).



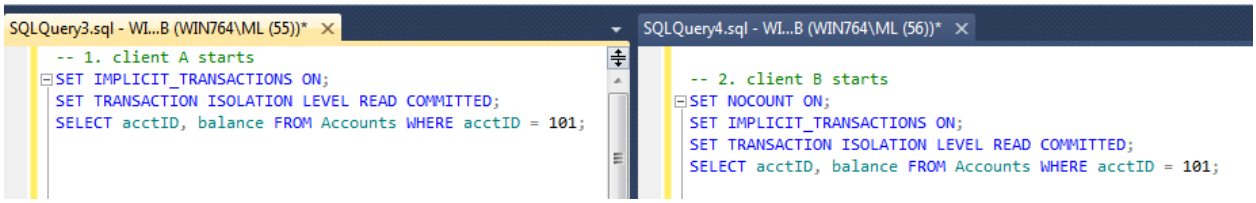
Joonis L1.1 Päringu akende kõrvuti seadmine.

```
-----------------------------------------------------------------
-- Ülesanne 2.1   Kadunud sissekande simuleerimine
-----------------------------------------------------------------
-- 0. Viime tabeli algseisu
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;
```

Kadunud sissekande probleem kui mõni sisestatud või uuendatud rida uuendatakse või kustutatakse teise samaaegse andmetehingu poolt enne esimese andmetehingu lõppu. Selline olukord võib olla võimalik mõnes faili-põhises „mitte-SQL“ lahenduses, kuid kaasaegsed ABHS-id välistavad selle tekkimise. Aga peale esimese andmetehingu kinnitamist võib ükskõik milline samaaegne andmetehing need read ikkagi üle kirjutada.

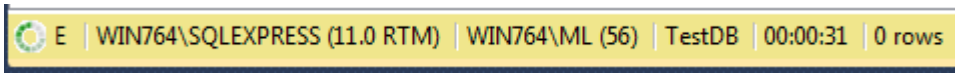
Järgnevalt simuleerime kadunud sissekande stsenaariumi kasutades kinnitatud andmete lugemise eraldatuse taset, kus S-lukustust ei säilitata. Kõigepealt loeb kliendi rakendus kontoseisud vabastades S-lukustused.

```
-- 1. Klient A alustab tegevust
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;

acctID      balance
----------- -----------
101         1000

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



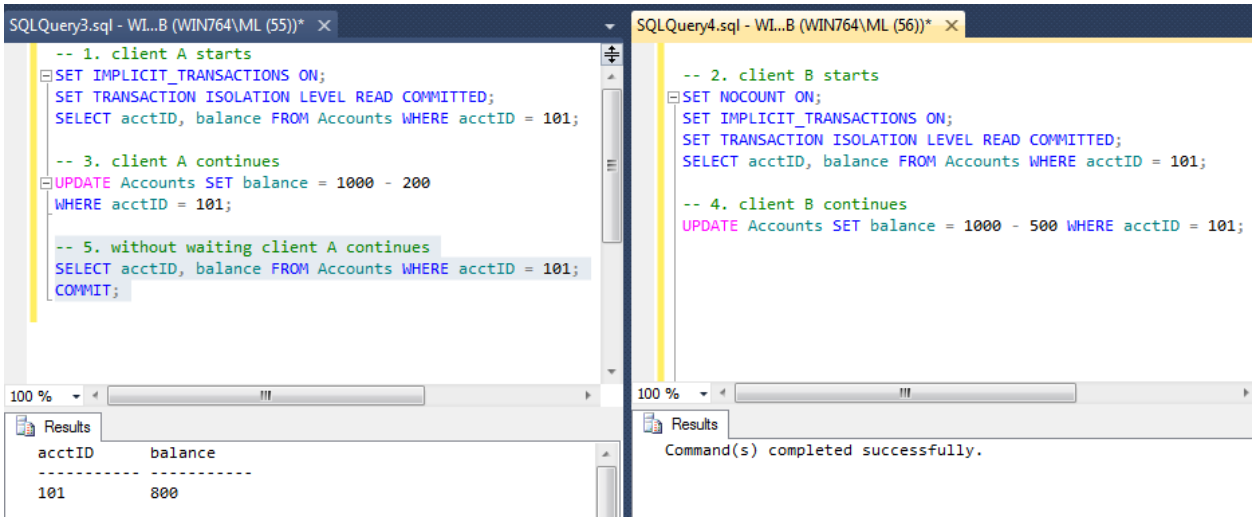
Joonis L1.2 Võistlevad Andmetehingud kõrvuti akendes.

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

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

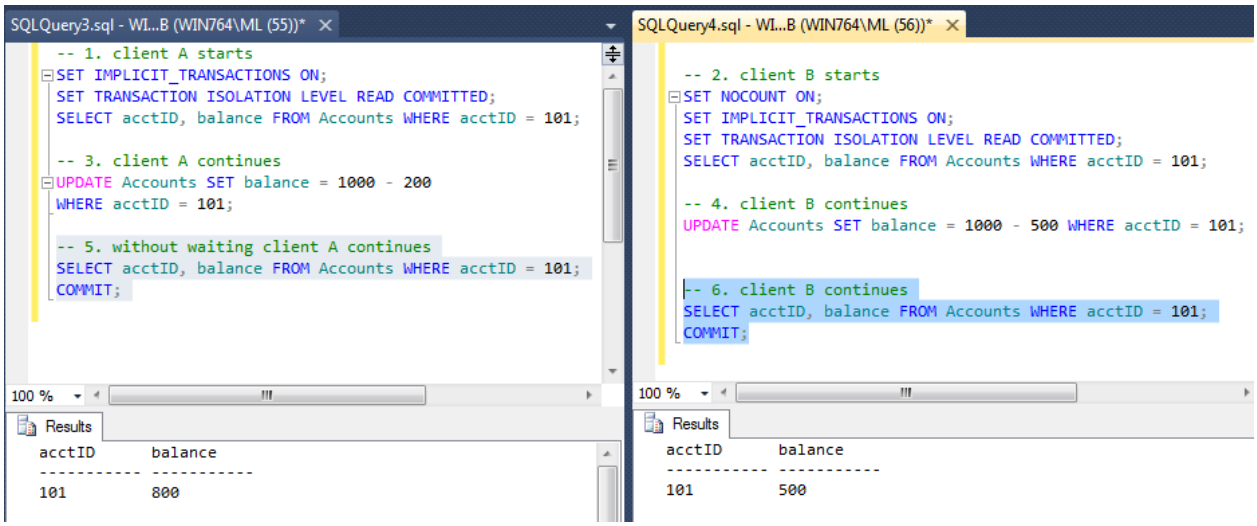


... ootame klienti A ...

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



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

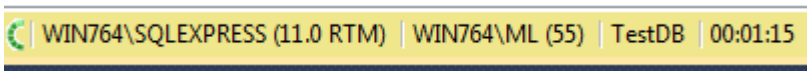


Lõpptulemus on vigane!

> Selles katses ei kasutanud me õiget kadunud sissekannet, kuid peale kliendi A andmetehingu kinnitamist, kirjutab klient B vastava rea üle – seda olukorda kutsutakse "**pimedaks ülekirjutamiseks**". Lahenduseks on "**tundlike uuenduste**" kasutamine:
>
> ```
> SET balance = balance - 500
> ```

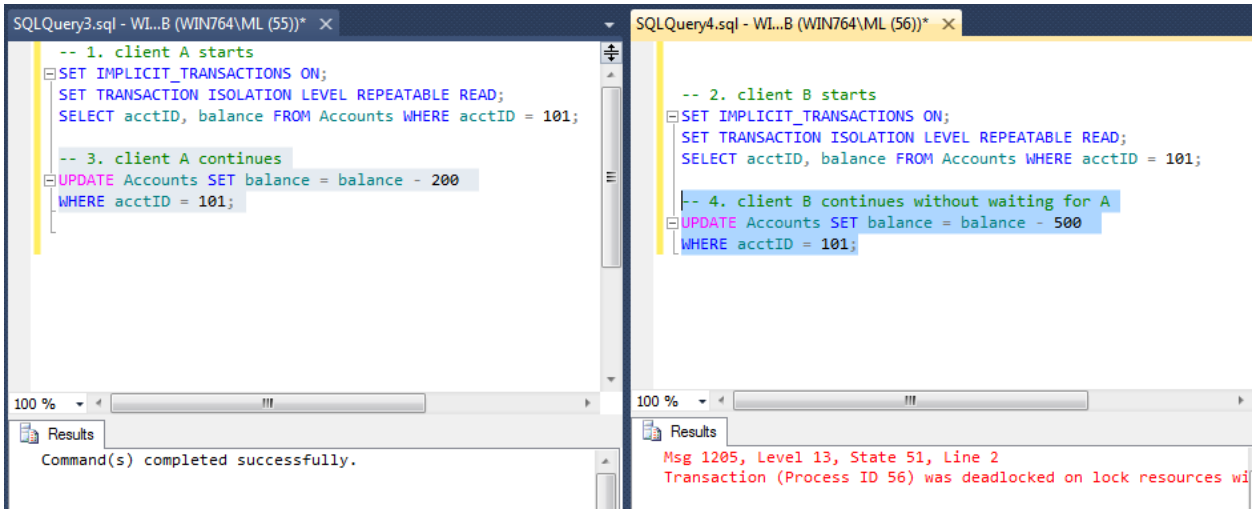
```
-----------------------------------------------------------------
-- Ülesanne 2.2   Kadunud sissekande probleemi lahendamine lukustuse abil
-----------------------------------------------------------------
--      Võidujooks sama ressursi (rida/read) üle
--      kasutades SELECT .. UPDATE käsustikku püüavad mõlemad
--      kliendid samalt kontolt raha välja võtta.
--
-- 0. Viime tabeli algseisu kliendi A sessioonilt
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. Klient A alustab tegevust
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;

-- 2. Klient B alustab tegevust
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
```

```
-- 3. Klient A jätkab
UPDATE Accounts SET balance = balance - 200
WHERE acctID = 101;
```



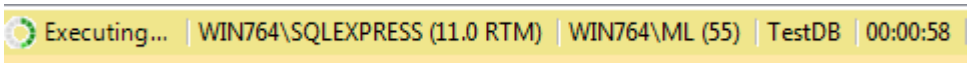
... ootame klienti B ...

```
-- 4. Klient B jätkab, ootamata klient A
UPDATE Accounts SET balance = balance - 500
WHERE acctID = 101;
```



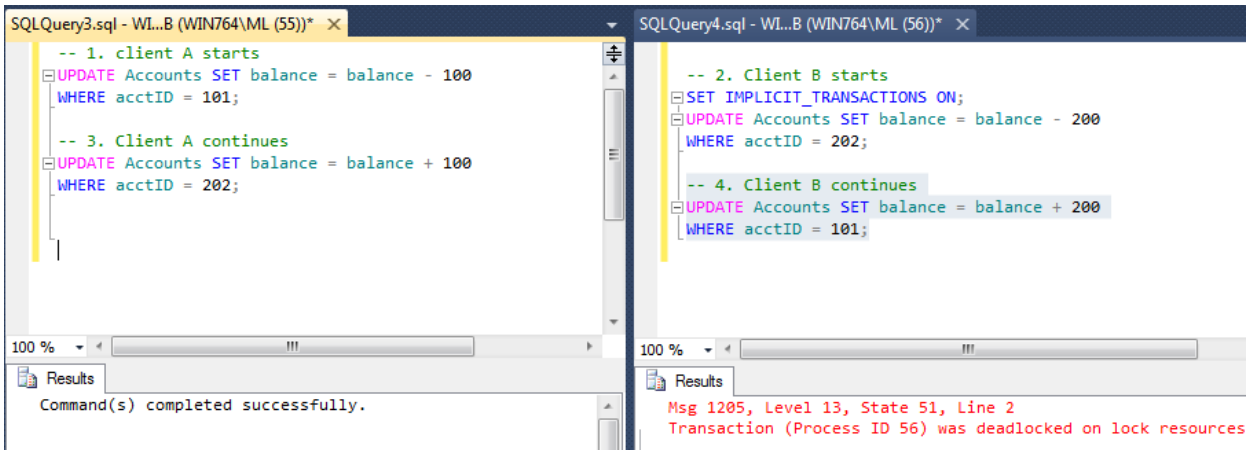

```
-- 5. Ummikseisu lahendamisel võitnud klient kinnitab Andmetehingu
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
acctID      balance
----------- -----------
101         800

COMMIT;


-----------------------------------------------------------------------
-- Ülesanne 2.3  Võidujooks kahele ressursile erinevas järjekorras
--               kasutades UPDATE-UPDATE käsustikku
-----------------------------------------------------------------------
-- klient A kannab 100 eurot kontolt 101 kontole 202
-- klient B kannab 200 eurot kontolt 202 kontole 101
--
-- 0. Viime tabeli algseisu kliendi A sessioonilt
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. Klient A alustab tegevust
UPDATE Accounts SET balance = balance - 100
WHERE acctID = 101;

-- 2. Klient B alustab tegevust
SET IMPLICIT_TRANSACTIONS ON;
UPDATE Accounts SET balance = balance - 200
```

```
WHERE acctID = 202;

-- 3. Klient A jätkab
UPDATE Accounts SET balance = balance + 100
WHERE acctID = 202;

COMMIT;
```



... ootame B-d ...

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




```
-- 5. Klient A jätkab, kui saab ...
COMMIT;
```

Ülesannetes 2.4 – 2.7 katsetame ISO standardis tuntud **samaaegse täitmise anomaaliate** ehk usaldusväärse andmetöötluse riskidega. Kas me suudame neid tuvastada? Kuidas me neid lahendada saame?

```
-----------------------------------------------------------------
-- Ülesanne 2.4   Poolikute andmete lugemine?
-----------------------------------------------------------------
--
-- 0. Viime tabeli algseisu kliendi A sessioonilt
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. Klient A alustab tegevust
SET IMPLICIT_TRANSACTIONS ON;

UPDATE Accounts SET balance = balance - 100
WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 100
WHERE acctID = 202;
```

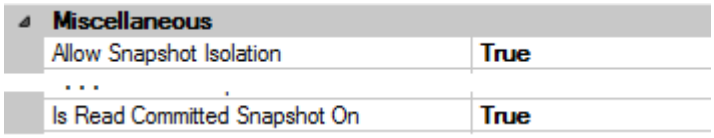
```
-- 2. Klient B alustab tegevust
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         900
202         2100
COMMIT;


-- 3. Klient A jätkab
ROLLBACK;

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

COMMIT;


-----------------------------------------------------------------
-- Ülesanne 2.5   Hägus lugemine?
-----------------------------------------------------------------
-- Hägusa lugemise anomaalia korral ei pruugi mõned loetud read
-- korduval lugemisel selle andmetehingu tulemustes enne andmetehingu
-- lõppu enam esineda.

-- 0. Viime tabeli algseisu kliendi A sessioonilt
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. Klient A alustab tegevust
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

--  Loeme üles kõik kontod, kus balance > 500 eurot
SELECT * FROM Accounts WHERE balance > 500;
acctID      balance
----------- -----------
101         1000
202         2000


-- 2. Klient B alustab tegevust
SET IMPLICIT_TRANSACTIONS ON;
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 202;
COMMIT;
```

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

COMMIT;


-----------------------------------------------------------------
-- Ülesanne 2.6   Sisestatud fantoomkirjed?
-----------------------------------------------------------------
-- Sisestatud fantoomkirjed on teise samaaegse andmetehingu poolt sisestatud
-- read, mida käesolev andmetehing võib enne lõppu näha.
--
-- 0. Viime tabeli algseisu kliendi A sessioonilt
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. Klient A alustab tegevust
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

-- kontod, kus balance > 1000 eurot
SELECT * FROM Accounts WHERE balance > 1000;
acctID      balance
----------- -----------
101         1000
202         2000


-- 2. Klient B alustab tegevust
SET IMPLICIT_TRANSACTIONS ON;
INSERT INTO Accounts (acctID,balance) VALUES (303,3000);
COMMIT;


-- 3. Klient A jätkab
-- vaatame tulemusi
SELECT * FROM Accounts WHERE balance > 1000;
acctID      balance
----------- -----------
202         2000
303         3000

COMMIT;
```

Küsimus

- Kuidas hoida ära fantoomkirjete esinemist?

```
-------------------------------------------------------------
-- "Mälupildi" uurimine
-------------------------------------------------------------
-- Andmebaasi seadetes tuleb võimaldada mälupilt eraldatuse tase.
-- Selleks loome uue andmebaasi

CREATE DATABASE SnapsDB;

-- Muudame uue andmebaasi seadeid vastavalt
```



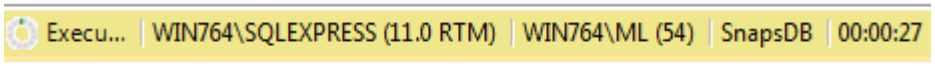

```
-- Edasi suuname mõlemad kliendid SnapsDB andmebaasi kasutama:

USE SnapsDB;


-------------------------------------------------------------------
-- Ülesanne 2.7   Erinevate fantoomkirjete katsed "mälupildi" korral
-------------------------------------------------------------------
USE SnapsDB;

-- 0. Alustame katsetustega ning loome uue tabeli T koos 5 reaga
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, 'forth', 2);
INSERT INTO T (id, s, i) VALUES (5, 'to be or not to be', 1);
COMMIT;

-- 1. Klient A alustab tegevust
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. Klient A alustab tegevust
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           forth                          2
6           Insert Phantom                 1


-- 3. Klient A jätkab
-- Kordame päringut ja proovime andmeid uuendada
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 klient C uuel sessioonil saadab päringu
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
(6 row(s) affected)
```

```
-- 4. Klient B jätkab
SELECT * FROM T;
id          s                              i
----------- ------------------------------ ------
```

```
1           first                          1
2           Update Phantom                 1
3           third                          1
4           forth                          2
6           Insert Phantom                 1


-- 5. Klient A jätkab
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;
```



... ootame klienti B ...

```
-- 6. Klient B jätkab ootamata klient A
COMMIT;

-- 7. Klient A jätkab
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. Klient B jätkab
SELECT * FROM T;
id          s                              i
----------- ------------------------------ ------
1           first                          1
2           Update Phantom                 1
3           third                          1
4           forth                          2
6           Insert Phantom                 1

(5 row(s) affected)
```

Küsimused

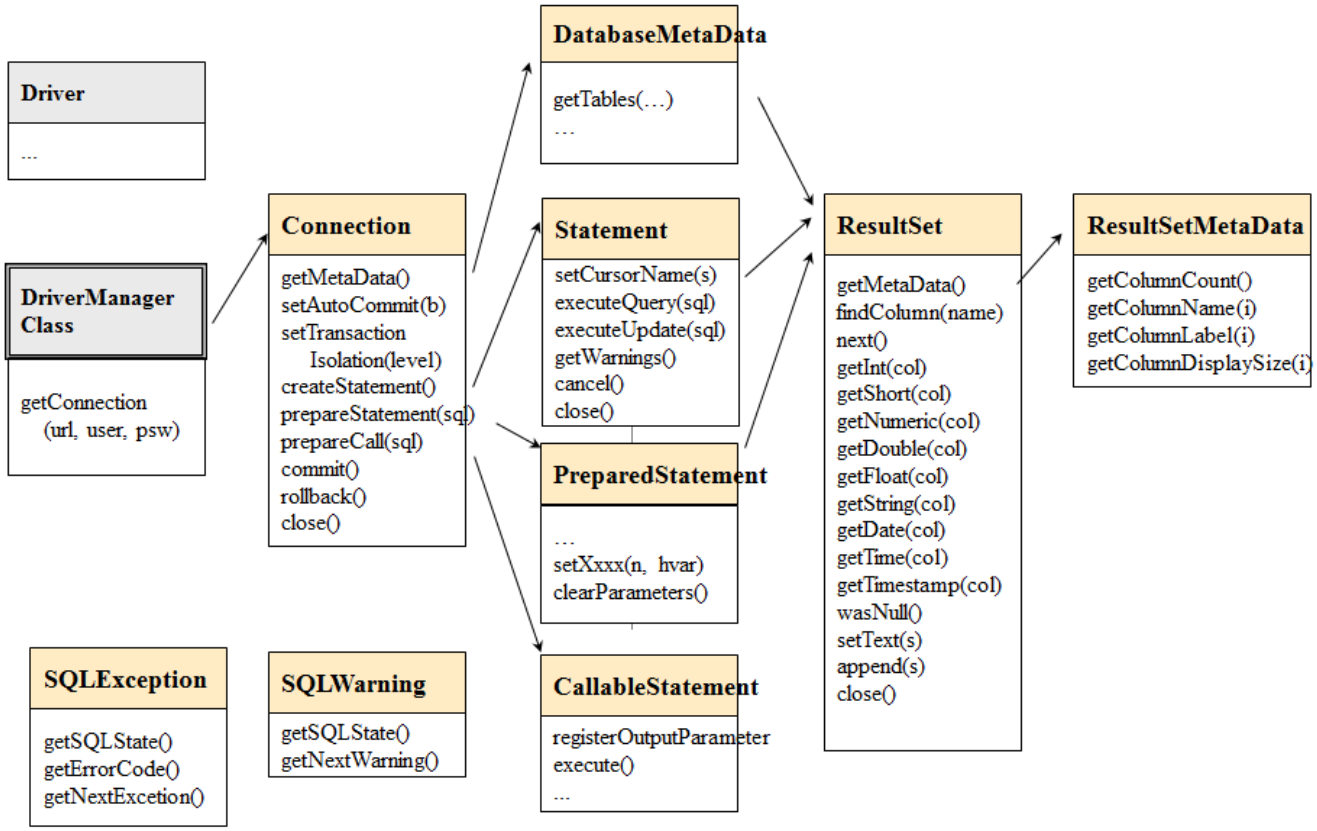
- Selgitage, kuidas see katse jätkus.
- Selgitage sammude 3.5 ja 4 tulemuste erinevust.

--

Põhjalikumalt SQL Serveri Andmetehingutest võite lugeda:
Kalen Delaney (2012), “SQL Server Concurrency, Locking, Blocking and Row Versioning”
ISBN 978-1-906434-90-8

Lisa 2 Andmetehingud Java-keeles programmeerimisel

Virtuaallaboris saame kasutada erinevate ABHS-ide spetsiifilisi ning SQL standardit järgivaid andmetehinguid ning neid interaktiivselt SQL redaktoris muuta. Kuid andmebaasidega suhtlevad rakendused on tavaliselt valmis programmeeritud mõne programmeerimisliidese abil. Järgmine näide on kirjutatud Java-keeles, kasutades JDBC programmeerimisliidest. Eeldame, et selle materjali lugeja on tuttav Java ja JDBC liidesega ning joonis L2.1 on vaid visuaalseks abi vahendiks mälu värskendamisel, kuidas Java põhilised objektid ja meetodid omavahel suhtlevad.



Joonis L2.1 JDBC põhikomponentide omavahelise suhtluse skeem.

Java koodinäide: BankTransfer

BankTransfer programm kannab 100 eurot ühelt (fromAcct) kontolt teisele (toAcct). Draiveri nime, andmebaasi aadressi, kasutajanime, parooli, fromAcct ja toAcct parameetrid sisestame käsurealt programmi parameetritena.

Testi läbiviimiseks peaks kasutaja avama kaks paralleelset ühendus – käsurea (Windows) või terminali (Linux) akent ning jooksutama programmi samamaegselt mõlemas aknas nii, et kord kantakse raha esimeselt kontolt teisele ja seejärel vastupidi (vt. järgnevaid skripte).

Lähtekonto debiteerimise järel jääb programm ootama ENTER-i vajutust, et täitmist sünkroniseerida ja samaaegsuse probleemi esile tuua. Programm demostreerib lisaks veel ka **kordus tsüklite** tööd.

Nimekiri L2.1 JDBC-s kirjutatud „BankTransfer“ Java kood

```
/*  DBTechNet Samaaegsuse labor 15.5.2008 Martti Laiho

   BankTransfer.java

Salvesta programm BankTransfer.java nimega ja kompileeri:

javac BankTransfer.java

Vaata BankTransferScript.txt failist SQL Serveri, Oracle ja DB2 koodinäiteid

Versiooni uuendused:
2.0 2008-05-26 ML takistab automaatset tühistamist SQL Serveri uumikseisu tekkimisel
2.1 2012-09-24 ML kordustsükli blokk esile toodud
2.2 2012-11-04 ML erandi käsitlemine olematu konto korral
**********************************************************/
import java.io.*;
import java.sql.*;
public class BankTransfer {
   public static String moreRetries = "N";

   public static void main (String args[]) throws Exception
   {
      System.out.println("BankTransfer version 2.2");

      if (args.length != 6) {
         System.out.println("Usage:" +
         "BankTransfer %driver% %URL% %user% %password% %fromAcct% %toAcct%");
         System.exit(-1);
      }
      java.sql.Connection conn = null;
      // String moreRetries = "N";
      boolean sqlServer = false;
      int counter = 0;
      String URL = args[1];
      String user = args[2];
      String password = args[3];
      String errMsg = "";
      int amount = 100;
      int fromAcct = Integer.parseInt(args[4]);
      int toAcct = Integer.parseInt(args[5]);

      // stringi tuvastus SQL Serveri andmetehingute seadistamiseks
      if (URL.substring(5,14).equals("sqlserver")) {
        sqlServer = true;
      }
            // JDBC draiveri registreerimine ja ühenduse loomine erandi käitlusega
      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
  {

  // Kordus tsükkel TransaferTransaction andmetehingu ajaks
     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);
       }
     }
     TransferTransaction (conn,
                    fromAcct, toAcct, amount,
                    sqlServer, errMsg //,moreRetries
                    );
     System.out.println("moreRetries="+moreRetries);
     if (moreRetries.equals("Y")) {
        long pause = (long) (Math.random () * 1000); // max 1 sec.
        System.out.println("Waiting for "+pause+ " mseconds"); // just for testing
        Thread.sleep(pause);
     }
  } while (moreRetries.equals("Y") && counter < 10);  // max 10 retries
  // kordus tsükli lõpp
   conn.close();
   System.out.println("\n End of Program. ");
}

static void TransferTransaction (Connection conn,
      int fromAcct, int toAcct, int amount,
      boolean sqlServer,
      String errMsg //, String moreRetries
      )
      throws Exception {
      String SQLState = "*****";
      try {
          conn.setAutoCommit(false);   // Andmetehing algab
          conn.setTransactionIsolation(
              Connection.TRANSACTION_SERIALIZABLE);
          errMsg = "";
          moreRetries = "N";

          // parameetritega SQL käsk
          PreparedStatement pstmt1 = conn.prepareStatement(
            "UPDATE Accounts SET balance = balance + ? WHERE acctID = ?");
          // parameetrite väärtustamine
```

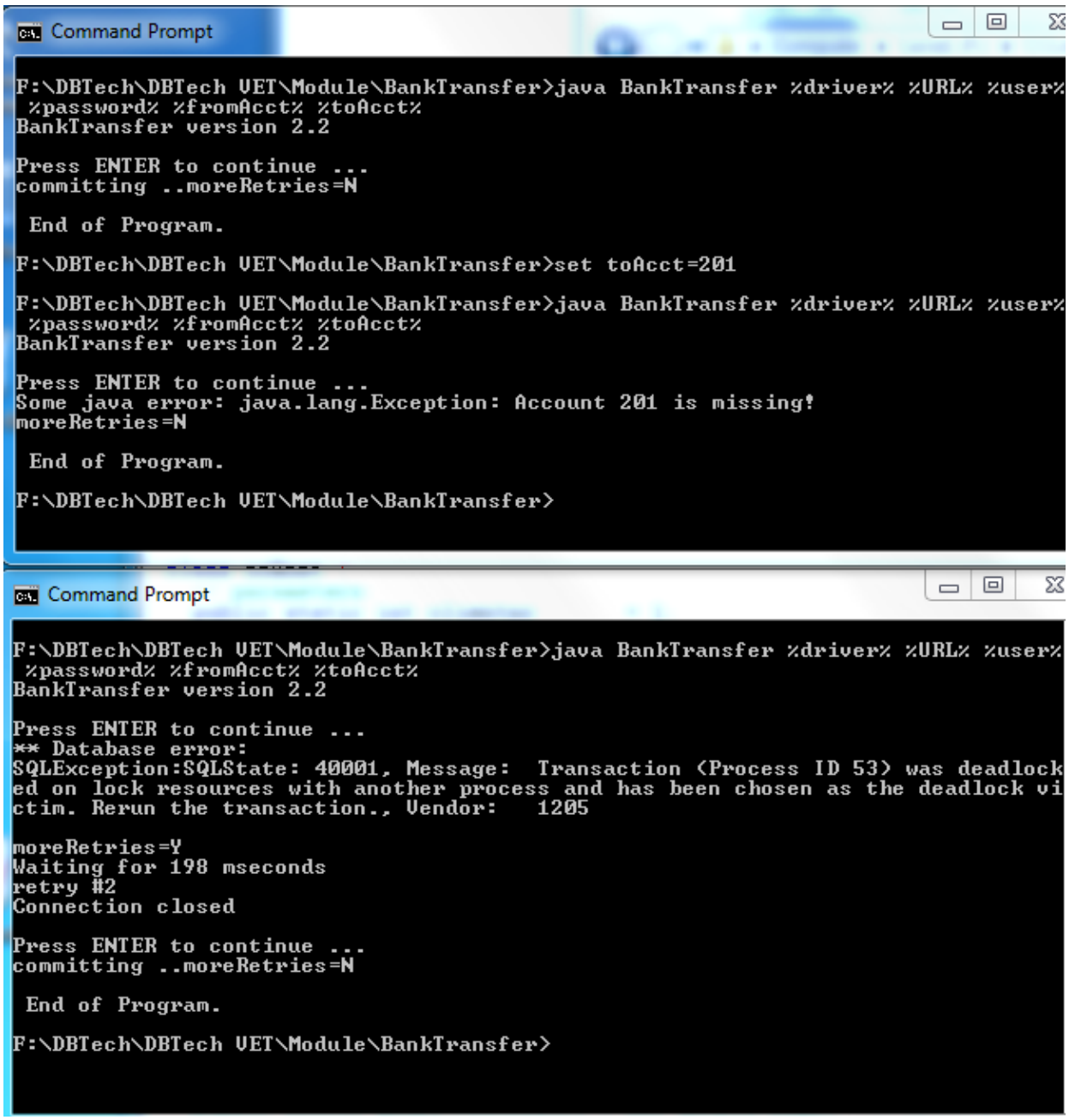
```
    pstmt1.setInt(1, -amount);  // välja võetava raha hulk
    pstmt1.setInt(2, fromAcct); // konto „number“
    int count1 = pstmt1.executeUpdate();
    if (count1 != 1) throw new Exception ("Account "+fromAcct + " is missing!");
    // --- paus samaaegse täitmise testimiseks -----
    // Peatame programmi kuni kasutaja vajutab ENTER klahvi,
    // et teine klient saaks jätkata konfliktset andmetehingut.
    // See on mõeldud samaaegsuse testimiseks, ärge kunagi
    // kasutage kasutaja dialoogi päris rakendustes!!!
      System.out.print("\nPress ENTER to continue ...");
      BufferedReader reader = new BufferedReader(
              new InputStreamReader(System.in));
      String s = reader.readLine();
    // --- pausi lõpp ------------------------------------
    pstmt1.setInt(1, amount); // ülekantava raha hulk
    pstmt1.setInt(2, toAcct);  // konto number
    int count2 = pstmt1.executeUpdate();
    if (count2 != 1) throw new Exception  ("Account "+toAcct + " is missing!");
    System.out.print("committing ..");
    conn.commit();  // Andmetehingu lõpp
    pstmt1.close();
  }
  catch (SQLException ex) {
    try {
       errMsg = "\nSQLException:";
       while (ex != null) {
            SQLState = ex.getSQLState();
            // kas tegemist on samaaegse täitmise konfliktiga?
            if ((SQLState.equals("40001")   // Solid, DB2, SQL Server,...
              || SQLState.equals("61000")   // Oracle ORA-00060: ummikseis
              || SQLState.equals("72000"))) // Oracle ORA-08177: ressurs lukustatud
                 moreRetries = "Y";
            errMsg = errMsg + "SQLState: "+ SQLState;
            errMsg = errMsg + ", Message:  "+ ex.getMessage();
            errMsg = errMsg + ", Vendor:   "+ ex.getErrorCode() + "\n";
            ex = ex.getNextException();
       }
       // SQL Server ei luba tühistamist peale ummikseisu !
       if (sqlServer == false) {
          conn.rollback();  // Oracle’ile on vaja eraldi tühistust
                            // ja see ei muuda DB2 käitumist
       }
       // println väljundtekst testimiseks
       System.out.println("** Database error: " + errMsg);
    }
    catch (Exception e) { // juhuks, kui erandi käsitlemisel tekib probleem
       System.out.println("SQLException handling error: " + e);
       conn.rollback(); // aktiivne andmetehing tühistatakse
       ;  // siia kirjutage vajadusel erandi käsitluse kood
    }
  } // SQL erand
```

```
        catch (Exception e) {
           System.out.println("Some java error: " + e);
           conn.rollback(); // ka siin tühistatakse aktiivne andmetehing
           ;  // siia vajadusel teise erandi käsitluse kood
        } // teised erandid
   }
}
```

Nimekirja L2.2 skripte võib kasutada programmi katsetamiseks Windows tööjaama peal kahes paralleelses käsurea aknas. Skriptid eeldavad SQL Server Express-i kasutamist, andmebaasiks on „TestDB“ ning JDBC-draiver on salvestatud programmi alamkataloogi „jdbc-drivers“.

Nimekiri L2.2 Skriptid BankTransfer-i katsetamiseks Windows-i keskkonnas

```
rem Esimese akna skript:
set CLASSPATH=.;jdbc-drivers\sqljdbc4.jar
set driver="com.microsoft.sqlserver.jdbc.SQLServerDriver"
set
URL="jdbc:sqlserver://localhost;instanceName=SQLEXPRESS;databaseName=TestDB"
set user="user1"
set password="sql"
set fromAcct=101
set toAcct=202
java BankTransfer %driver% %URL% %user% %password% %fromAcct% %toAcct%


rem Teise akna skript:
set CLASSPATH=.;jdbc-drivers\sqljdbc4.jar
set driver="com.microsoft.sqlserver.jdbc.SQLServerDriver"
set
URL="jdbc:sqlserver://localhost;instanceName=SQLEXPRESS;databaseName=TestDB"
set user="user1"
set password="sql"
set fromAcct=202
set toAcct=101
java BankTransfer %driver% %URL% %user% %password% %fromAcct% %toAcct%
```



Joonis L2.1 Lihtne BankTransfer test Windows-i keskkonnas.

Teiste ABHS-ide ja Linux-i platvormi jaoks saab lihtsa vaevaga modifitseerida skripte nimekirjas L2.2.

Nimekiri L2.3 Skriptid BankTransfer-i katsetamiseks Linux-i keskkonnas.

Skriptid Linuxi MySQL-ile:

```
# Loome kataloogi /opt/jdbc-drivers JDBC draiverite jaoks
cd /opt
mkdir jdbc-drivers
chmod +r+r+r jdbc-drivers
# kopeerime MySQL jdbc draiveri kataloogi /opt/jdb-drivers
cd /opt/jdbc-drivers
```

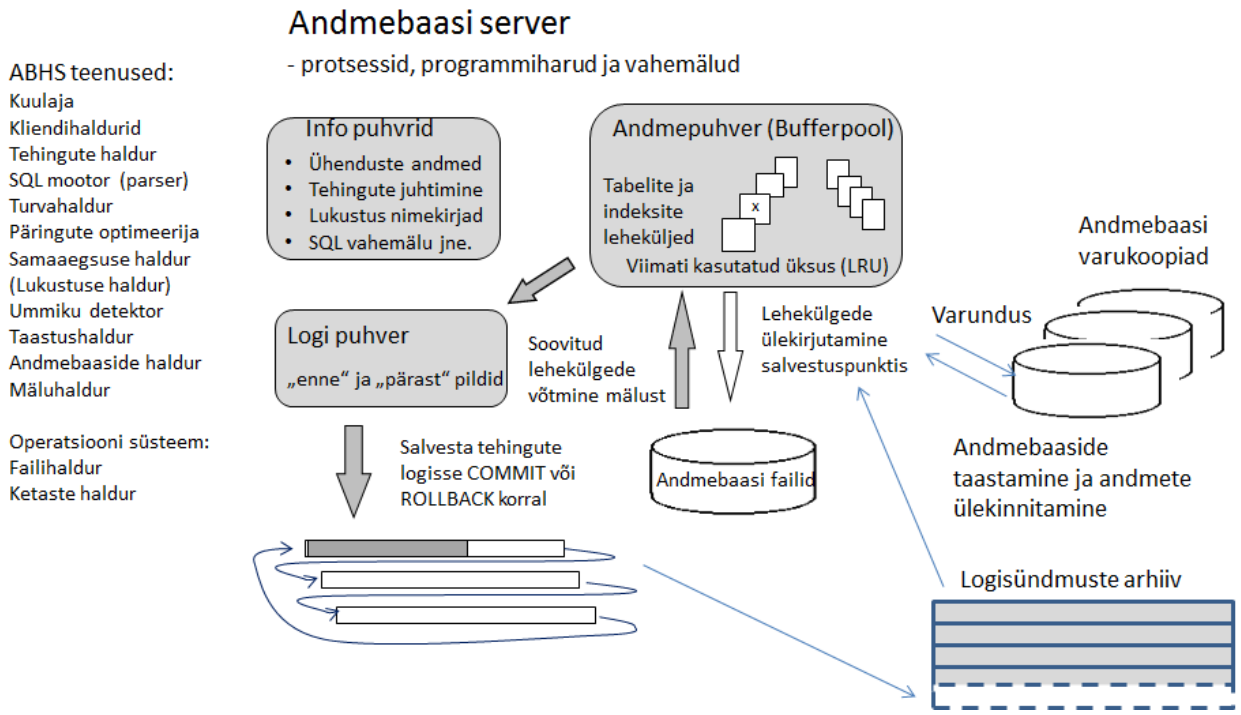
```
cp $HOME/mysql-connector-java-5.1.23-bin.jar
# anname draiveri lugemisõigused kõigile
chmod +r+r+r mysql-connector-java-5.1.23-bin.jar


#******* MySQL/InnoDB ************************************************

# Esimene aken:
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


# Teine aken:
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
#*********************************************************************
```

Lisa 3 Andmetehingud ja andmebaasi taastamine

Joonis L3.1 kirjeldab „suurt pilti“ andmetehingute käitlemisest tüüpilises andmebaasi serveris. Juhendi "Basics of SQL Transactions" slaidid on saadaval

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

Seal kirjeldatakse detailsemalt laiatarbe ABHS-e, seal hulgas andmebaasi failide ja andmetehingute logide („mälupilt“ MS SQL Server-i Andmetehingute logist) ja andmete vahemälu haldusest, hoidmaks mälu ketaste tööd miinimumis tõusnud koormuse juures; seda, kuidas SQL andmetehinguid käitletakse; sellest, kuidas tagatakse COMMIT käsu usaldusväärsus ning ROLLBACK käsu toimimine.



Joonis L3.1 Andmebaasi serveri üldine struktuur.

Edaspidi kirjeldame, kuidas ABHS on võimeline taastama andmebaasi seisu kuni viimase kinnitatud andmetehinguni toite katkemise või süsteemi kokkujooksmise järel. Riistvara tõrke puhul saab andmebaasi taasta varukoopialt ja Andmetehingute logist.

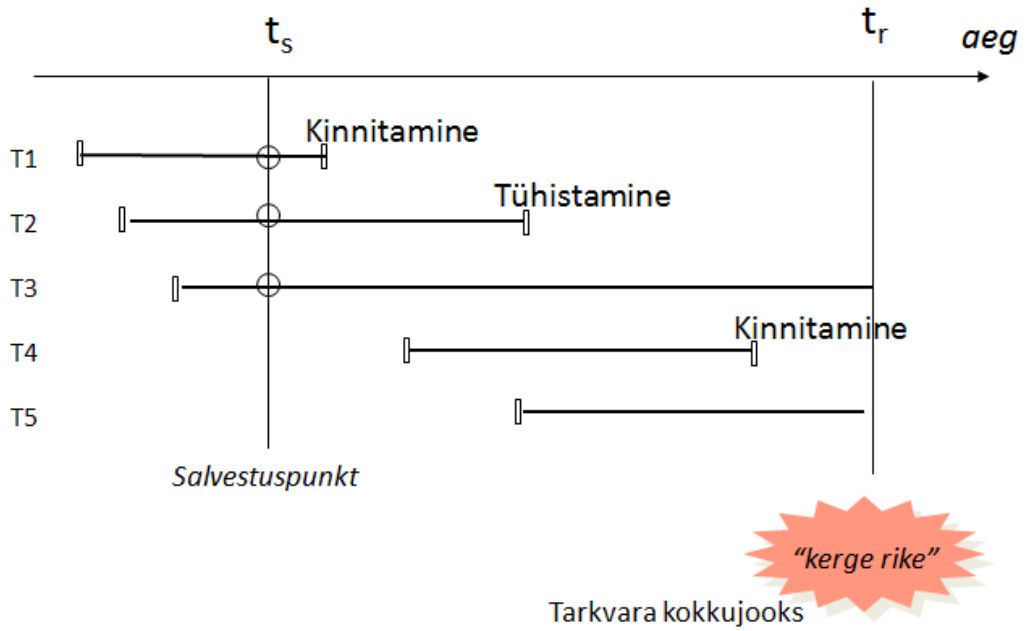
(vaata ka: **http://www.dbtechnet.org/papers/RdbmsRecovery.ppt**)

Kui SQL andmetehing algab, annab andmebaasiserver sellel unikaalse identifitseerimisnumbri ja iga tegevus andmetehingus salvestatakse andmetehingute logifaili. Iga töödeldud rea kohta on logifailis andmetehingu identifikaator, rea sisu enne ja pärast andmetehingut. INSERT käsu koha peal puudub „enne“ ja DELETE käsu puhul „pärast“ osa. Ka COMMIT ja ROLLBACK käskudest jääb jälg logisse ning kogu info salvestatakse serveri kettale. Näiteks peale COMMIT käsu sisestamist antakse kliendile tagasiside alles peale logi sissekande salvestamist.
Aeg-ajalt teeb server CHECKPOINT operatsioone, kus peatatakse hetkeks klientide teenindamine ning andmetehingud kirjutatakse vahemälust logifaili ning kõik uuendatud andmed (vastava märkmega read) kirjutatakse andmepuhvrist andmebaasi. Ka salvastuspunkti

(checkpoint) andmed kirjutatakse logisse koos kõigi aktiivsete andmetehingute indentifikaatoritega. Lõpuks jätkub klientide teenindamine.

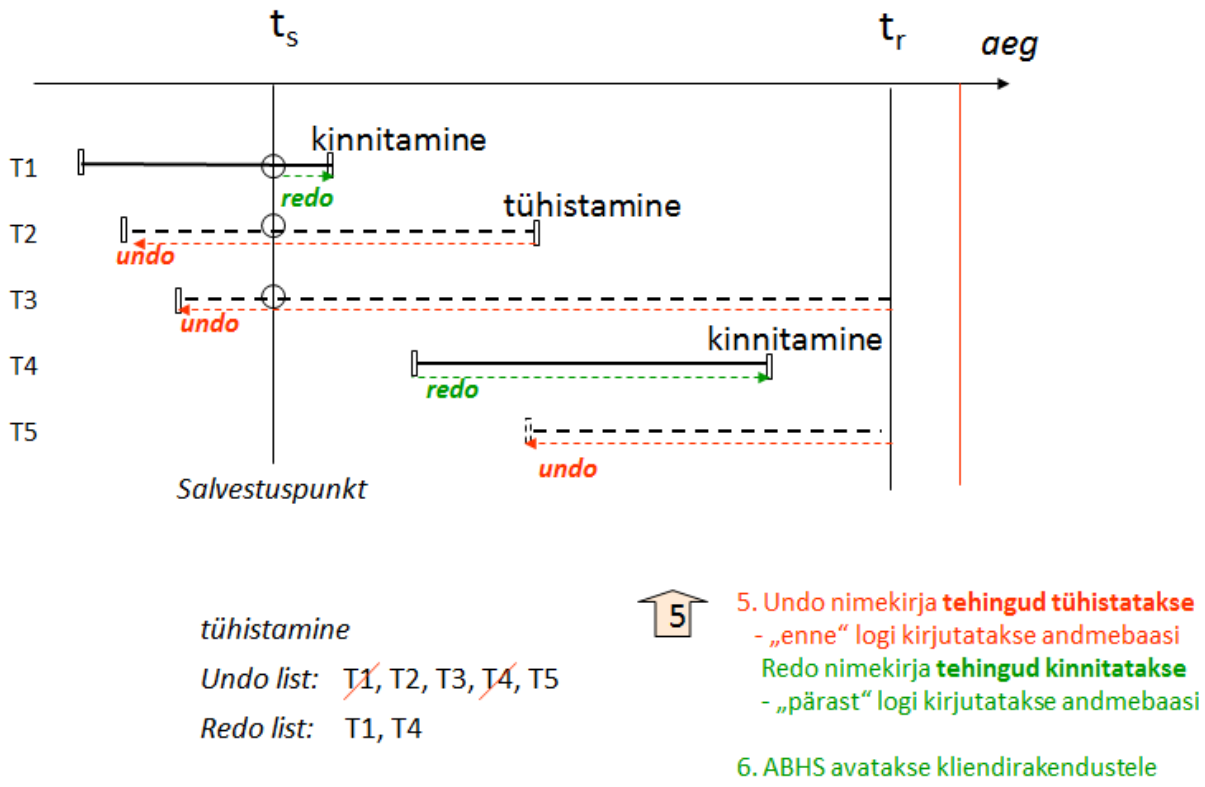
> Plaanipärase serveri sulgemise (shutdown) ajal ei tohiks olla käimas ühtegi andmetehingut, sellel juhul kirjutab viimane logi operatsioon logisse tühja salvestuspunkti, mis on märgiks puhtast lõpetamisest.

Joonis L3.2 kirjeldab andmetehingute logi ajalugu enne andmebaasi riket või arvuti sulgumist näiteks toitekatkestuse tõttu. Kõik kettale kirjutatud andmed on loetavad, kuid andmepuhvri sisu läheb kaduma. Seda olukorda nimetatakse „**kergeks rikkeks**“.



Joonis L3.2 Andmetehingute logi ajalugu enne „kerget riket“.

Kui server taaskäivitatakse, leiab see andmetehingute logist viimase salvestuspunkti. Kui see on tühi, oli tegemist normaalse sulgemisega ning server on valmis kliente teenindama. Vastasel juhul hakkab server tühistama tegevusi järgnevalt. Kõigi salvastuspunktis olevate andmetehingute identifikaatori kopeeritakse tühistamiseks UNDO nimekirja ning REDO nimekirja need, mis on enne riket kinnitatud. Seega algul satuvad kõigi andmetehingute numbrid tühistatavate tegevuste nimekirja ning seejärel liigutatakse logi lõpupoole kinnituse saanud andmetehingud ülekinnitamise nimekirja, et nad andmebaasis kindluse mõttes üle kirjutada. Seejärel tühistatakse kõik UNDO nimekirja andmetehingud ja nende poolt muudetud read taastatakse olukorda, mis oli enne vastavate andmetehingute alustamist andmetehingu logi „enne“ pildi järgi ning kinnitatakse andmetehingud, mis oleks pidanud enne riket andmebaasi kirjutatama logis oleva „pärast“ pildi järgi. Peale seda on server taastatud viimase kinnitatud andmetehinguni ning klientide teenindamine võib jätkuda (vt. joonis L3.3).



Joonis L3.3 Andmebaasi taastamine.

Erinevalt lihtsustetud taastamise protseduurist, mida just kirjeldasime, kasutab enamus kaasaegseid ABHS-e ARIES protokolli, kus salvestuspunktide vahel andmebaasi ei kirjutada, seda teeb taustal haruprogramm, mis märgib muudetud ridu LSN märgenditega. Nende märgendite kasutamine teeb taastamise protsessi kiiremaks.