Westfälische Hochschule Bocholt

Fachbereich 5

Wintersemester 2016/2017

Dokumentation zur Praktikumsaufgabe 9:

Messen von Lasttransaktionen

im Praktikum „DBI“

bei Prof. Dr. Bernhard Convent

und Dipl.-Ing. Hans-Peter Huster

vorgelegt von:	Thomas Manderla
und	Miguel Oppermann
Praktikumsgruppe:	4.1
Studienfach:	Informatik.Softwaresysteme
Fachsemester:	3
E-Mailadressen:	thomas.manderla@studmail.w-hs.de
	miguel.oppermann@studmail.w-hs.de

Bocholt, den 22. Dezember 2016

Inhaltsverzeichnis

1 Einleitung

1.1 Aufgabenstellung

Aufgabe war es, die in Praktikumsaufgabe 7 erstellte 100-tps-Benchmark-Datenbank mit drei Methoden, die Transaktionen auf ihr durchführen, zu belasten und die erreichte Zahl an Transaktionen zu messen. Dabei galt es, mehrere Details zu beachten.
Die drei Lasttransaktionen sollten innerhalb einer bestehenden Verbindung durchgeführt werden. Sie sollten folgende Aufgaben erfüllen:

1. **Kontostand-Transaktion** Erwartet Kontonummer ACCID und gibt Kontostand BALANCE zurück.
2. **Einzahlungs-Transaktion** Erwartet Eingabewerte für:
   - Kontonummer ACCID
   - Automatennummer TELLERID
   - Zweigstellennummer BRANCHID
   - Betrag DELTA

   und soll folgende Aktionen durchführen:
   - In der Relation BRANCHES soll die zur BRANCHID gehörige BALANCE aktualisiert werden
   - In der Relation TELLERS soll die zu TELLERID gehörige Bilanzsumme BALANCE aktualisiert werden.
   - In der Relation ACCOUNTS soll der zu ACCID gehörige Kontostand BALANCE aktualisiert werden.
   - In der Relation HISTORY soll die Einzahlung (incl. des aktualisierten Kontostandes ACCOUNTS.BALANCE) protokolliert werden.

   und schließlich den neuen Kontostand, BALANCE, zurückgeben.
3. **Analyse-Transaktion** Erwartet Einzahlungsbetrag DELTA und gibt Anzahl der Transaktionen mit diesem Betrag zurück.

Diese Transaktionen sollten zufällig gewählt, mit der Gewichtung 35/50/15, in einer 10-minütigen Schleife durchlaufen werden, wobei mehrere Eigenheiten zu beachten waren:

- Nach jeder Transaktion herrscht eine Zwangspause von 50 ms.
- Die Messphase beginnt erst nach 4 Minuten „Einschwingphase“.
- Nach der 5-minütigen Messphase gibt es eine 1-minütige „Ausschwingphase“. In der Messphase werden die Anzahl der Transaktionen und die durchschnittliche Zahl der Transaktionen pro Sekunde ermittelt.
- Bei alldem sollten die ACID-Eigenschaften[1] der Transaktionen gewährleistet sein.

Diese Messungen sollten unter drei verschiedenen Last-Bedingungen durchgeführt werden, mit je 5 Load Drivers auf einem, zwei und drei Client-Rechnern.

1.2 Konfiguration des verwendeten Systems

Wir nutzten MySQL 5.7.15. Der für den erste remote-Zugriff genutzte Client-Rechner besaß einen Intel Core-i5 mit 1,7 GHz und 8 GB RAM und verwendete Windows 10. Für weitere Tests wurden die Pool-Rechner der Fachhochschule als Client-Rechner verwendet. Wir programmierten in Java. Die IDE war Eclipse Neon.1 und als Datenbankschnittstelle wurde JDBC verwendet. Unser JDBC Treiber war MySQL-Connector-Jave 5.1.40. Das DB-Benchmarking-Framework wurde von uns nicht verwendet. Dafür gab es im Wesentlichen drei Gründe:

- Das Framework bietet mehr Funktionalität als benötigt.
- Die Einarbeitung in das Framework wäre zeitaufwändig gewesen. Was genau wird gemessen?
- Wir sahen in der eigenständigen Programmierung einen besseren Lerneffekt.

Das Ergebnis ist unser Programm *Bextasy*, das im Folgenden vorgestellt wird.

[1] Die Verwendung in der Aufgabenstellung und das ausführliche Besprechen in der Vorlesung lassen uns davon Abstand nehmen, die ACID-Eigenschaften an dieser Stelle näher zu erläutern.

2 Aufbau des Programms *Bextasy*

2.1 Überblick

Brextasy wurde allein für diese Praktikumsaufgabe geschrieben und versucht, auf jeglichen für die Lösung der Aufgabe nicht benötigten Overhead zu verzichten. Es weist eine kleine Menüführung auf, die raschen Einstieg in die nötigen Programmabläufe bietet. Brextasy besteht aus drei Klassen, die die zentralen Aufgaben übernehmen. Diese werden im Folgenden vorgestellt. Der vollständige Code in seiner finalen Version befindet sich im Anhang. Die Javadoc findet sich unter der Website `https://s0t7x.github.io/WHS.DBI.Bextasy/`.

2.2 Die Klasse *Main*

Die Klasse Main der Application Bextasy beinhaltet die Methode Main. Diese wird beim Start des Programms aufgerufen. Bextaxy stellt in dieser Methode ein konsolenbasiertes Menü dar, über welches der Nutzer interaktiv auf verschiedene Funktionen des Programms zugreifen und diese ausführen kann. Die Menüführung ist sehr simpel gehalten: Die verschieden Menüpunkte wie z.B. „[1] Manage Connection“ werden in der Konsole dargestellt. Der Nutzer tippt die vorhergehende Zahl für seine Auswahl ein und bestätigt die Eingabe mit drücken der Return-Taste. Die Menüführung ist so entworfen, dass der Nutzer in diversen Untermenüs weitere Auswahlmöglichkeiten hat. Auch wenn dies nicht einer besonders effizienten Entwicklung entspricht (z.B. Hätten auch Startparameter oder sogar gar keine Parameter benutzt werden können) stehen Nutzerfreundlichkeit und Vielseitigkeit hierbei im Vordergrund.

2.3 Die Klasse *DBmgmt*

Die gesamte Klasse DBmgmt, was für „Database Management“ stehen soll, dient zur Verbindungsherstellen und enthält die Methoden „getBalance“, „deposit“ und „analyse“, welche in der Praktikumsaufgabe definiert sind. Der Constructor der Klasse ist so ent-

wickelt, dass er mit den Parametern „_ServerAddress“, „_DatabaseName“, „_UserName“und „_Password“aufgerufen werden muss. Sobald ein Objekt der Klasse mit diesen Parametern erstellt wird, wird die Klasseneigene Methode „connect()“ aufgerufen.

2.3.1 Die Methode *connect()*

Mit Aufruf der Methode „connect()“ wird versucht via JDBC eine Verbindung zum Server auf zu bauen. Die Verbindungsengeschaften sind vom Constructor, welcher diese als Parameter übernimmt, in der Klasse hinterlegt. „AutoCommit“ wird direkt nach Verbindungsaufbau auf „false“ gesetzt, damit manuelle commits durchgeführt werden können. „TransactionIsolation“ wird mit der Konstanten „Connection.TRANSACTION_SERIALIZABLE“ gesetzt, um die Serialisierbarkeit zu gewährleisten.

2.3.2 Die Methode *getBalance*

Als Eingabeparameter wird der Wert einer Kontonummer (ACCID) erwartet. Die Methode führt eine einfache SQLQuery durch und liefert den zur Kontonummer gehörigen Kontostand (BALANCE) als Rückgabewert.

2.3.3 Die Methode *deposit*

Von dieser Methode wird als Eingabeparameter jeweils ein Wert für die Kontonummer (ACCID), die Geldautomatennummer (TELLERID), die Zweigstellennummer (BRANCHID) und den Einzahlungsbetrag (DELTA) erwartet. Mit diesen Parametern wird von der Methode sowohl in der Relation BRANCHES, als auch in der Relation TELLERS die zu BRANCHID bzw. zu TELLERID gehörige Bilanzsumme BALANCE aktualisiert werden. Ebenfalls wird BALANCE in der Relation ACCOUNTS zu der dazugehörigen ACCID aktualisiert. In der Relation HISTORY wird die Einzahlung als aktualisierter Konstostand (ACCOUNTS.BALANCE) protokoliert. Die Methode liefert als Ausgabewert den neuen Kontostand der ACCID aus.

2.3.4 Die Methode *analyse*

Diese Methode liefert als Rückgabewert die Anzahl der bisher protokollierten Einzahlung mit genau diesem Betrag. Dazu muss ihr als Eingabeparameter der Einzahlungsbetrag (DELTA) mitgegeben werden.

2.4 Die Klasse *LoadDriver*

Objekte der Klasse LoadDriver können als eigener Thread ausgeführt werden. Ein Thread der Klasse wählt in einer 10-minütigen Schleife zufällig eine der obigen Methoden aus DBmgmt und führt diese mit sinnvollen Parametern aus. Die relative Gewichtung für die zufällige Auswahl liegt für entweder „getBalance“, „deposit“ oder „analyse“ bei 35 zu 50 zu 15. Zwischen einzelnen Transaktionen wartet der Thread genau 50 msec. Dies ist eine festgelegte „Nachdenkzeit“ (engl. Think Time), welche eingehalten wird, bevor die nächste Lasttransaktion startet. Das Hauptprogramm wird nach aufruf des LoadDrivers pausiert, bis alle Threads, welche aufgerufen wurden, durchlaufen und beendet sind.

```
private void transaction() throws SQLException {
                // Get us a random number between 0 and 100
                int chance = random.nextInt(100);
                // So it is decided what to do based on the Chance 35/50/15
                if (chance < 35) {
                        // Get the balance of a random accid
                        int accid = random.nextInt(100 * 100000) + 1;
                        dbmgmt.getBalance(accid);
                } else if (chance >= 35 && chance < 85) {
                        // Deposit random parameters on random accid
                        // "+ 1" because we dont want it to be 0
                        int accid = random.nextInt(100 * 100000) + 1;
                        int tellerid = random.nextInt(100 * 10) + 1;
                        int branchid = random.nextInt(100 * 1) + 1;
                        int delta = random.nextInt(10000) + 1;
                        dbmgmt.deposit(accid, tellerid, branchid, delta);
                } else {
                        // Analyse with random delta
                        int delta = random.nextInt(10000) + 1;
                        dbmgmt.analyse(delta);
                }
```

}

3 Optimierungen am Programm und Schwierigkeiten auf dem Weg

3.1 Initialmessung

Um von Optimierungen sprechen zu können, sollen an dieser Stelle kurz die Beschreibung der Besonderheiten unserer Eingangsmessung erfolgen. Wir begannen mit einem einzigen Load Driver, gestartet von einem Laptop mit oben gegebener Spezifizierung. Die Serialisierbarkeit der Transaktionen war noch nicht gewährleistet und es waren auch noch keine Rollbacks eingefügt.
Bei diesem Testlauf kam es zu einem vorzeitigen Abbruch der Verbindung zwischen Client und Server, was wir erst bemerkten, als die Messung abgeschlossen war. Der erzielte Durchsatz war demnach nicht sehr hoch, jedoch gestaltete sich die Fehlersuche sehr interessant.[2] Wir führten mehrere Probeläufe durch, die, da sie zur Fehlerfindung dienten, nicht in der Messergebnistabelle im Anhang aufgeführt sind. Dabei stellten wir zwei Dinge fest:

1. Der Abbruch der Verbindung zum Server erfolgte *jedesmal*.
2. Er fand zu unzusammenhängenden und unvorhersagbaren Zeitpunkten statt.

Die ersten Debugging-Versuche verliefen dementsprechend ergebnislos – das Programm lief wie geplant und die Verbindung blieb während des Durchschreitens der Einzelschritte bestehen. Erst als wir einen genaueren Blick auf die gerufenen Transaktionsmethoden warfen, entdeckten wir, dass beim Debuggen ausschließlich die Einzahlungs-TX-Methode aufgerufen worden war. Da sie die höchste Wahrscheinlichkeit zugewiesen bekommen hatte, war dies nicht verwunderlich. Bei ihr lag der Fehler möglicherweise nicht verborgen.

[2] Wenngleich der Fehler selbst im höchsten Maße trivial war.

Wir schraubten zunächst an den Wahrscheinlichkeiten, gingen dann aber rasch dazu über, die einzelnen Transaktionen bewusst aufzurufen, indem wir ihre Wahrscheinlichkeiten jeweils auf 100% setzten. So fanden wir schnell heraus, dass die Verbindung nur beim Aufrufen der dritten Transaktions-Methode, *Analyse-TX*, abbrach. Die Ursache war nun einfach zu entdecken: wir hatten die Verbindung mit

```
conn.close();
```

in der Analyse-TX-Methode selbst geschlossen. Nachdem wir dies behoben hatten, konnten wir uns endlich der Serialisierbarkeit zuwenden.

3.2 Messung 2: Gewährleistung der Serialisierbarkeit

Für das Einhalten der ACID-Bedingungen aktivierten wir die Option, dass die Transaktionen serialisierbar durchgeführt werden:

```
conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
```

Ein Vergleich mit der Initialmessung ist aufgrund des dortigen vorzeitigen Verbindungsabbruchs nicht sinnvoll möglich. Daher können wir die in dieser Messung erzielten Werte als Grundlage für die Einschätzung weiterer Veränderungen nutzen.

8.85	0	0	Initialtest mit einem Load Driver. Kein Prepared Statement. Keine Rollbacks. Vorzeitiger Abbruch der Verbindung.
75.1	0	0	Serialisierbarkeit gewährleistet. Test auf Laptop.

Abbildung 1: Die erste Messung, die unterbrechungslos durchlief und bei der auf Serialisierbarkeit geachtet wurde.

3.3 Messung 3: Einsetzen eines Rollbacks

Bevor wir uns an die Verbesserung des Transaktionsumsatzes machten, mussten wir sicherstellen, dass nicht vollständig abschließbare Transaktionen zurückgesetzt werden, sodass wir von wirklich atomaren Transaktionen ausgehen konnten.

In der Methode *deposit* fügten wir daher einen Rollback ein:

```
try {
 conn.commit();
} catch (SQLException e) {
 try {
 e.printStackTrace();
  conn.rollback();
 } catch (SQLException e2) {
  conn.rollback();
 }
}
stmt.close();
```

Verständlicherweise machten sich die zusätzlichen Operationen, die durch die Rollbacks erforderlich wurden, auch in den Messergebnissen bemerkbar. Wie in Abbildung 2 ersichtlich, sank die durchschnittliche Tps um etwa 5,5.

75.1	0	0	Serialisierbarkeit gewährleistet. Test auf Laptop.
69.53	0	0	wie oben; Rollback hinzugefügt

Abbildung 2: Das Einfügen des Rollbacks senkte die erzielte Zahl an Tps etwas.

3.4 Messung 4: Einführung von Prepared Statements

Mit unserer vierten Messung begannen wir mit den Optimierungen am Programm. Wir führten die Messungen nun von den Pool-PCs aus durch. Des Weiteren übergaben wir die SQL-Anfragen nun mittels Prepared Statement, um die Geschwindigkeit etwas anzuziehen:

```
PreparedStatement stmt = conn.prepareStatement("");
stmt.executeQuery("SET FOREIGN_KEY_CHECKS=0");

stmt = conn.prepareStatement("UPDATE accounts SET balance=balance+? WHERE accid = ?");
stmt.setInt(1, delta);
stmt.setInt(2, accid);
stmt.execute();

stmt = conn.prepareStatement("UPDATE tellers SET balance=balance+? WHERE tellerid = ?");
stmt.setInt(1, delta);
```

```
stmt.setInt(2, tellerid);
stmt.execute();

stmt = conn.prepareStatement("UPDATE branches SET balance=balance+? WHERE branchid = ?");
stmt.setInt(1, delta);
stmt.setInt(2, branchid);
stmt.execute();

stmt.executeUpdate("INSERT INTO history(accid, tellerid, delta, branchid, accbalance, cmmnt)
    " + "VALUES("
 + accid + "," + tellerid + "," + delta + "," + branchid + "," + newBalance + ",'" +
     string30 + "')");
```

Wir entdeckten außerdem, dass wir bislang die History-Relation – entgegen der Aufgabenstellung – noch nicht geleert hatten. Ein Blick in die entstandene Datei offenbarte uns eine Größe von über einem GB. Vermutlich ist das Leeren der History-Relation ebenfalls ein Beitrag zur Steigerung der Performance bei Last 1 gewesen.

Zudem stellten wir fest, dass unser DBMS noch in den Standardeinstellungen lief, weil wir noch vor der ersten Messung unsere Workbench neu installieren mussten. Daher setzten wir den der VM zugewiesenen Speicher auf 2 GB und erhöhten die „innodb_buffer_pool_size“ von 8MB auf 3GB.

Die vielen simultan durchgeführten Änderungen erschweren es natürlich, die genauen Ursachen für Veränderungen in den Messergebnissen zu ermitteln. Die Umstände ermöglichten es aber nicht, die Veränderungen einzeln zu messen.
Die durchgeführten Veränderungen bewirkten zunächst eine Verbesserung im Vergleich zu Messung 3. Ein Blick auf die durchschnittliche Transaktionszahl pro Sekunde pro Rechner offenbart jedoch, dass es mit jedem verwendeten Client-Rechner weniger Tps werden – bei Last 2 liegt der Durchschnitt nur knapp über dem Ergebnis von Messung 3 und bei Last 3 sogar deutlich darunter. Eine Analyse dieser Beobachtung findet im folgenden Abschnitt statt.

75.1	0	0	Serialisierbarkeit gewährleistet. Test auf Laptop.
69.53	0	0	wie oben; Rollback hinzugefügt
79.6	140.17	176.57	Arbeiten mit Pool PCs; Prepared Statements hinzugefügt. Leeren der History hinzugefügt. Änderungen am DBMS (wg. Reinstall der Workbench -> wurde bei vorherigen Messungen missachtet)

Abbildung 3: Die durchgeführten Veränderungen bewirkten zunächst eine Verbesserung im Vergleich zu Messung 2 und 3.

3.5 Messung 5: Umstellung der SQL-Statements

Die erste Messung über alle drei Lasten machte uns stutzig, hatten wir doch eine lineare Skalierung erwartet. Stattdessen sank die durchschnittliche Performance pro Rechner (und auch pro Thread) bei zunehmender Rechnerzahl. Es kam zu keinen vorzeitigen Verbindungsabbrüchen. Wir verdächtigten Konflikte, die mit der Zahl an Zugriffen auf die Relationen zu tun hatten. Gemäß dieser Hypothese würden mehr Rechner mehr Zugriffe mit sich bringen, die wiederum mehr Konflikte erzeugen, woraufhin mehr kurzfristige Sperrungen stattfinden, was zu mehr Rollbacks und weniger erfolgreichen Transaktionen pro Sekunde führe.

Ein Blick auf die Gestaltung unserer SQL-Statements zeigte uns Verbesserungspotential. Die Prepared Statements in der deposit-Methode griffen zuerst auf die Branches-Relation zu, anschließend auf die Tellers-Relation und erst zuletzt auf die Accounts-Relation. Das ist insofern problematisch, als wir nur n Branches, in dieser Aufgabe also 100, haben, während wir $n \cdot 100000$ Accounts haben. Ein Konflikt und damit eine Sperrung der Tupel oder im schlimmsten Fall der Seitenrahmen ist bei einer kleinen Zahl natürlich viel wahrscheinlicher. Die Reihenfolge ist relevant, weil in unserem Fall zunächst einer aus 100 Branches gesperrt wird, während die anderen Bereiche noch eingepflegt werden müssen, ein Tupel mit einer höheren Konfliktwahrscheinlichkeit also länger gesperrt bleibt.

Um diese Hypothese zu überprüfen, fügten wir Bextasy einen Rollback-Zähler hinzu und wiederholten Messung 4. Anschließend ordneten wir die Statements gemäß ihrer

Wahrscheinlichkeit, zu Konflikten zu führen, also von der größten Tupelzahl zur kleinsten:

vorher	**nachher**
Branches	Accounts
Tellers	Tellers
Accounts	Branches

Eine fortgeschrittenere Herangehensweise, für die uns leider die Zeit fehlte, wäre es gewesen, eine Analyse der vom Optimizer bearbeiteten Statements vorzunehmen.

3.6 Nicht durchgeführte Maßnahmen

Stored Procedures

Mehrere Gruppen versuchten sich an Stored Procedures, um den Durchsatz zu erhöhen. Zunächst zogen wir es auch in Erwägung, unser Programm danach umzustellen. Unsere Recherche ergab allerdings, dass es zwischen einem normalen SQL-Aufruf und dem Verpacken in Stored Procedures Performance-mäßig keinen Unterschied geben würde. Mit „This is a myth“[3] wird in einer unserer Quellen die Annahme, Stored Procedures brächten eine Verbesserung in der Performance, abgeschmettert. Jede parametrisierte SQL-Anfrage werde ganz genauso wie eine Stored Procedure abgewickelt. Die Reihenfolge sei:[4]

- Syntaktische Überprüfung der Query.
- Wenn sie okay ist, wird der Plan Cache durchforstet, um zu sehen, ob es bereits einen Execution Plan für diese Query gibt.
- Wenn es bereits einen Execution Plan gibt, wird er verwendet und die Query ausgeführt. [Ende]
- Wenn es keinen Execution Plan gibt, wird einer ermittelt.

[3] Vgl. `http://stackoverflow.com/questions/8559443/`.
[4] `http://stackoverflow.com/questions/12948312/`.

- Dieser Plan wird für spätere Wiederverwendung gespeichert.
- Die Query wird ausgeführt. [Ende]

Es gebe keinen Unterschied in der Performance.[5]

Wie erklären wir uns die Verbesserungen anderer Gruppen? Möglicherweise führten sie die Messungen nicht oft genug durch, sodass natürliche Schwankungen der neuen Herangehensweise attribuiert wurden.

Leider erlaubte es uns der knappe Zeitplan nicht mehr, diese Änderung zu implementieren, um die Rechercheergebnisse mit harten Fakten zu untermauern. Idealerweise würden wir Stored Procedures einführen und die Messergebnisse in Kontrast zu den Werten aus Messungen 4 und 5 setzen und daraus unsere Schlussfolgerungen ziehen.

4 Fazit

Insgesamt konnte der durchschnittliche Durchsatz – verglichen mit der Aufgabe des Befüllens der Datenbank – nur im geringen Maße gesteigert werden. Durch das Einfügen von Prepared Statements und das Leeren der History konnte eine Erhöhung des Durchsatzes um 10 Tps – etwa 14% – erzielt werden.

Die Wahl, das Programm selbst zu schreiben, anstatt das Framework zu verwenden, erwies sich als gut. Wir konnten gezielt und schlank arbeiten. Eine derartige Entscheidung läuft oftmals auf eine Optimierungsentscheidung hinaus. Optimierungskriterium ist die verwendete Zeit. Wir entschieden uns nach dem Lesen der Dokumentation des Frameworks dagegen, Zeit in die Einarbeitung in das Framework zu investieren und dafür, die Zeit in das Entwickeln des eigenen Programms zu stecken.

[5] Ebd.

5 Anhang

5.1 Messergebnisse

	Messung	Last 1	Last 2	Last 3	Bemerkungen
Tps/Rechner	1	8,85	0	0	Initialtest mit einem Load Driver,
Tps gesamt		8,85	0	0	Kein Prepared Stmt. Keine Rollbacks,
TX gesamt		k. A	0	0	Vorzeitiger Abbruch der Verbindung.
Tps/Rechner	2	75,1[6]	0	0	Serialisierbarkeit gewährleistet,
Tps gesamt		75,1	0	0	Test auf Laptop.
TX gesamt		k. A	0	0	
Tps/Rechner	3	69,53	0	0	inklusive Rollback
Tps gesamt		69,53	0	0	
TX gesamt		20874	0	0	
ø Tps/Rechner	4	79,6[6]	70,35	58,86	Prepared Statements,
Tps gesamt		79,6	140,17	176,57	Verbesserungen am DBMS,
ø TX/Rechner		23923	21001	17621	Leeren der History.
TX gesamt		23923	42002	52863	
	Messung	Last 1	Last 2	Last 3	Bemerkungen

[6] Genauere Messwerte wurden leider nicht protokolliert.

5.2 Finaler Quellcode

5.2.1 Die Klasse *Main*

Die Javadoc findet sich unter der Website `https://s0t7x.github.io/WHS.DBI.Bextasy/`.

```
package bextasy.Main;

import java.sql.SQLException;
import java.sql.Statement;
import java.util.Scanner;

import bextasy.Connection.*;

/**
 * Main Class and Method
 *
 * @author s0T7x
 *
 */
public class Main {
	static DBmgmt DB_conn_main = null;
	static Scanner ReadConsole = new Scanner(System.in);

	public static void main(String[] args) throws SQLException, InterruptedException {
		// TODO Auto-generated method stub

		_Menu_Home();
	}

	/**
	 * Prints a simple console based Menu and lets the user choose between
	 * several options. This is the Home Menu. From here the user starts and can
	 * get into other menus.
	 *
	 * Menu: - 1. Goes to "Manage Connection" Menu - 2. Goes to "Database
	 * Functions" Menu - 3. Starts the Benchmarking 4. Exits application
	 *
	 * Points 2/3 are disabled till the user connected with a database in the
	 * "Manage connection" menu.
	 *
	 * @throws SQLException
	 * @throws InterruptedException
```

```
 */
static void _Menu_Home() throws SQLException, InterruptedException {
        int selection = 0;

        do {
                System.out.println("[1] Manage Connection");
                System.out.println("[2] Database Functions");
                System.out.println("[3] Benchmark Database");
                System.out.println("[4] Exit");
                System.out.print("\nBextasy~> ");
                selection = ReadConsole.nextInt();

                switch (selection) {
                case 1:
                        _Menu_ManageConnection();
                        System.out.println("");
                        break;
                case 2:
                        if (DB_conn_main != null)
                                _Menu_DatabaseFunctions();
                        System.out.println("");
                        break;
                case 3:
                        if (DB_conn_main != null)
                                _Menu_BenchmarkDatabase();
                        System.out.println("");
                        break;
                case 4:
                        break;

                default:
                        System.out.println("\nInvalid selection!\n");
                        break;
                }
        } while (selection != 4);
        ReadConsole.close();
}

/**
 * Menu to manage Connections and to interact with DBmgmt.
 *
 * Menu: - 1. Connects / Reconnect to given address, database with given
 * credentials - 2. Connects to localhost database "benchmark" with root and
 * no password which was used to test locally - 3. Goes back to "Home Menu"
```

```
 */
static void _Menu_ManageConnection() {
        // DB_conn_main = new DBmgmt("localhost","benchmark", "root", "");
        int selection = 0;

        do {
                if (DB_conn_main != null)
                        System.out.println("Connected: " + DB_conn_main.UserName + "
                            @ " + DB_conn_main.ServerAddress + "/"
                                        + DB_conn_main.DatabaseName + "\n");
                System.out.println("[1] Re-/Connect");
                System.out.println("[2] [DEBUG] Connect to localhost");
                System.out.println("[3] Back");
                System.out.print("\nBextasy~> ");
                selection = ReadConsole.nextInt();

                switch (selection) {
                case 1:
                        DB_conn_main = null;
                        System.out.print("Server Address: ");
                        String temp_address = ReadConsole.next();
                        System.out.print("Database Name: ");
                        String temp_name = ReadConsole.next();
                        System.out.print("Username: ");
                        String temp_user = ReadConsole.next();
                        System.out.print("Password: ");
                        String temp_pw = ReadConsole.next();
                        DB_conn_main = new DBmgmt(temp_address, temp_name, temp_user,
                             temp_pw);
                        System.out.print("");
                        selection = 3;
                        break;
                case 2:
                        // DB_conn_main = new DBmgmt("localhost", "benchmark", "root
                             ",
                        // "");
                        DB_conn_main = new DBmgmt("192.168.122.55", "benchmark", "dbi
                             ", "dbi_pass");
                        System.out.println("");
                        selection = 3;
                        break;
                case 3:
                        break;
```

```
                default:
                        System.out.println("\nInvalid selection!\n");
                        break;
                }
        } while (selection != 3);
}

/**
 * Menu to do different DBmgmt methods
 *
 * Menu: - 1. Get the Balance of a given ACCID - 2. Deposit a specified
 * amount to an ACCID - 3. Search for an deposit amount in the History - 4.
 * Goes back to Home Menu
 *
 * Suppress "static-access" warnings, because they are annoying in
 * eclipse...
 *
 * @throws SQLException
 * @throws InterruptedException
 */
@SuppressWarnings("static-access")
static void _Menu_DatabaseFunctions() throws SQLException, InterruptedException {
        int selection = 0;

        do {
                System.out.println("[1] GetBalance(ACCID)");
                System.out.println("[2] Deposit(ACCID, AMOUNT)");
                System.out.println("[3] Analyse(AMOUNT)");
                System.out.println("[4] Back");
                System.out.print("\nBextasy~> ");
                selection = ReadConsole.nextInt();

                int accid = 0;
                switch (selection) {
                case 1:
                        System.out.print("Enter ACCID: ");
                        accid = ReadConsole.nextInt();
                        System.out.println("Balance = " + DB_conn_main.getBalance(
                            accid));
                        System.out.println("");
                        break;
                case 2:
                        // System.out.print("Could do manual deposit here, but its
                            not
```

```
                        // coded yet\n");
                        // Above comment is a LIE!!!
                        System.out.print("Enter ACCID: ");
                        int temp_accid = ReadConsole.nextInt();
                        System.out.print("Enter TELLERID: ");
                        int temp_tellerid = ReadConsole.nextInt();
                        System.out.print("Enter BRANCHID: ");
                        int temp_branchid = ReadConsole.nextInt();
                        System.out.print("Enter DELTA: ");
                        int temp_delta = ReadConsole.nextInt();
                        DB_conn_main.deposit(temp_accid, temp_tellerid, temp_branchid
                            , temp_delta);
                        break;
                case 3:
                        System.out.print("Could do manual analyse here, but its not
                            coded yet\n");
                        break;
                case 4:
                        break;

                default:
                        System.out.println("\nInvalid selection!\n");
                        break;
                }
        } while (selection != 4);
}

/**
 * Asks for the amount of threads and starts LoadDriver threads on connected
 * Database
 *
 * @throws SQLException
 * @throws InterruptedException
 */
@SuppressWarnings("static-access")
static void _Menu_BenchmarkDatabase() throws SQLException, InterruptedException {
        System.out.print("Amount of LoadDrivers to execute: ");
        int drivers = ReadConsole.nextInt();

        Statement st1 = DB_conn_main.conn.createStatement();
        st1.execute("DELETE FROM history");

        Thread threads[] = new Thread[drivers];
        LoadDriver loadDrivers[] = new LoadDriver[drivers];
```

```
                for (int i = 0; i < drivers; i++) {
                        loadDrivers[i] = new LoadDriver(DB_conn_main);
                        threads[i] = new Thread(loadDrivers[i]);
                        threads[i].start();
                }

                // Waits till all threads finished
                for (int i = 0; i < drivers; i++) {
                        threads[i].join();
                }
                DB_conn_main.conn.close();
        }
}
```

5.2.2 Die Klasse *DBmgmt*

```
package bextasy.Connection;

import java.sql.*;

/**
 * Class to manage a connection with connection specific Methods.
 *
 * @author s0t7x
 */
public class DBmgmt extends Thread {

        // ----------- Definitions -------------
        public String ServerAddress;
        public String DatabaseName;
        public String UserName;
        public String Password;
        public static Connection conn = null;
        // -------------------------------------

        /**
         * Constructor directly connects to the database
         *
         * @param _ServerAddress
         * @param _DatabaseName
         * @param _UserName
         * @param _Password
         */
```

```
public DBmgmt(String _ServerAddress, String _DatabaseName, String _UserName, String
    _Password) {
        // Initialize a specified Connection and connect()
        ServerAddress = _ServerAddress;
        DatabaseName = _DatabaseName;
        UserName = _UserName;
        Password = _Password;
        System.out.println("Ready to connect to " + _ServerAddress);
        connect();
}

/**
 * Connects to the Database defined in this object It's not Public because
 * it is only used by the class itself in the constructor
 */
void connect() {
        // Try to establish a connection
        System.out.println("Connect to " + ServerAddress + " with given credentials
            for " + UserName + "...");
        try {
                conn = DriverManager.getConnection(
                                "jdbc:mysql://" + ServerAddress + "/" + DatabaseName
                                    + "?allowMultiQueries=true&useLocalSessionState=
                                    true", UserName,
                                Password);
                conn.setAutoCommit(false);
                conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
                System.out.println("Connection established!");
        } catch (SQLException e) {
                e.printStackTrace();
        }
}

/**
 * From "DBI: Aufgabenblatt 5": "Die Methode erwartet als Eingabeparameter
 * den Wert einer Kontonummer ACCID und gibt den zugehörigen Kontostand
 * BALANCE als Ausgabewert zurück."
 *
 * @param accid
 * @return Balance of ACCID
 * @throws SQLException
 */
public static int getBalance(int accid) throws SQLException {
        int balance = 0;
```

```
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT balance FROM accounts WHERE accid="
            + accid);
        if (rs.next())
            balance = rs.getInt("balance");
        conn.commit();
        stmt.close();
        return balance;
}

/**
 * From "DBI: Aufgabenblatt 5": "Die Methode erwartet als Eingabeparameter
 * jeweils Werte für - eine Kontonummer ACCID, - eine Geldautomatennummer
 * TELLERID, - eine Zweigstellennummer BRANCHID - und einen
 * Einzahlungsbetrag DELTA. Damit sollen innerhalb dieser Transaktion die
 * folgenden Einzelaktionen durchgeführt werden: - In der Relation BRANCHES
 * soll die zu BRANCHID gehörige Bilanzsumme BALANCE aktualisiert werden. -
 * In der Relation TELLERS soll die zu TELLERID gehörige Bilanzsumme BALANCE
 * aktualisiert werden. - In der Relation ACCOUNTS soll der zu ACCID
 * gehörige Kontostand BALANCE aktualisiert werden, und - in der Relation
 * HISTORY soll die Einzahlung (incl. des aktualisierten Kontostandes
 * ACCOUNTS.BALANCE) protokolliert werden. Der ermittelte neue Kontostand
 * soll als Ausgabewert der Methode zurückgegeben werden."
 *
 * @param accid
 * @param tellerid
 * @param branchid
 * @param delta
 * @return New Balance for ACCID
 * @throws SQLException
 */
public static int deposit(int accid, int tellerid, int branchid, int delta) throws
    SQLException {
        int newBalance = getBalance(accid) + delta;

        String string30 = "12345678901234567890123456789";

        PreparedStatement stmt = conn.prepareStatement("");
        stmt.executeQuery("SET FOREIGN_KEY_CHECKS=0");

        stmt = conn.prepareStatement("UPDATE accounts SET balance=balance+? WHERE
            accid = ?");
        stmt.setInt(1, delta);
        stmt.setInt(2, accid);
```

```
            stmt.execute();

            stmt = conn.prepareStatement("UPDATE tellers SET balance=balance+? WHERE
                tellerid = ?");
            stmt.setInt(1, delta);
            stmt.setInt(2, tellerid);
            stmt.execute();

            stmt = conn.prepareStatement("UPDATE branches SET balance=balance+? WHERE
                branchid = ?");
            stmt.setInt(1, delta);
            stmt.setInt(2, branchid);
            stmt.execute();

            stmt.executeUpdate("INSERT INTO history(accid, tellerid, delta, branchid,
                accbalance, cmmnt)" + "VALUES("
                            + accid + "," + tellerid + "," + delta + "," + branchid + ","
                                + newBalance + ",'" + string30 + "')");

            try {
                    conn.commit();
            } catch (SQLException e) {
                    try {
                            e.printStackTrace();
                            conn.rollback();
                    } catch (SQLException e2) {
                            conn.rollback();
                    }
            }
            stmt.close();
            return getBalance(accid);
    }

    /**
     * From "DBI: Aufgabenblatt 5": "Die Methode erwartet als Eingabeparameter
     * den Wert eines Einzahlungsbetrages DELTA und gibt die Anzahl bisher
     * protokollierter Einzahlungen mit genau diesem Betrag als Ausgabewert
     * zurück."
     *
     * @param delta
     * @return Amount of Deposits with given Value delta
     * @throws SQLException
     */
    public static int analyse(int delta) throws SQLException {
```

```
                int count = 0;
                Statement stmt = conn.createStatement();
                ResultSet rs = stmt.executeQuery("SELECT COUNT(*) AS count FROM history WHERE
                     delta=" + delta);
                if (rs.next())
                        count = rs.getInt("count");
                conn.commit();
                return count;
        }

}
```

5.2.3 Die Klasse *LoadDriver*

```
package bextasy.Connection;

import bextasy.Connection.DBmgmt;
import java.sql.*;
import java.util.Random;

/**
 * From "DBI: Aufgabenblatt 5": "[...] in einem Load-Driver-Programm, das 10
 * Minuten lang in einer Schleife jeweils zufällig gewählt eine der obigen TXs
 * mit zufällig gewählten, sinnvollen Parametern1 durchfhrt und dabei die
 * bekannten ACID-Eigenschaften garantiert. Zwischen zwei einzelnen TXs soll
 * jeweils eine feste "Nachdenkzeit" (engl. Think Time) von genau 50 msec
 * liegen, in der das Benchmark-Programm nach der erfolgreichen Abarbeitung
 * einer TX einfach wartet, bevor es die nächste Lasttransaktion startet. [...]"
 *
 * @author s0T7x
 *
 */
public class LoadDriver extends Thread {
        // 50ms Think Time
        private int thinkTime = 50;

        // 4 Minuten Einschwingphase
        private int initTime = 240000;//240000;

        // 1 Minute Ausschwingphase
        private int endTime = 60000;//60000;

        // 5 Minuten Messphase
        private int benchmarkTime = 300000;//300000;
```

```
// Counts the amount of transactions and is used to calculate TPSs
private int transactionCount;

private DBmgmt dbmgmt;
Connection conn;
private Random random = new Random();

@SuppressWarnings("static-access")
public LoadDriver(DBmgmt dbmgmt) {
        this.dbmgmt = dbmgmt;
        this.conn = dbmgmt.conn;
}

public void run() {
        init();

        benchmark();

        // Starts "Ausschwingphase"
        end();

        // Calculate and print TPS, cause thats what we want to know
        tps();

        // Even without print we should see when it's done but to print it makes it
            much cooler
        System.out.println("LoadDriver done!");
}

/**
 * "Einschwingphase" Does transactions for 4 minutes WITHOUT counting!
 */
private void init() {
        // Remember when we started
        long startTime = System.currentTimeMillis();
        // And loop till we reach the defined time
        while ((System.currentTimeMillis() - startTime) < initTime) {
                try {
                        // Do some cool transaction stuff
                        transaction();
                        // And sleep for the defined thinkTime
                        Thread.sleep(thinkTime);
                } catch (InterruptedException | SQLException e) {
                        e.printStackTrace();
```

```
                        break;
                }
        }
}

/**
 * "Messphase" Does transactions for 5 minutes WITH counting
 */
private void benchmark() {
        long startTime = System.currentTimeMillis();
        while ((System.currentTimeMillis() - startTime) < benchmarkTime) {
                try {
                        transaction();
                        Thread.sleep(thinkTime);
                } catch (InterruptedException | SQLException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                        continue;
                }
                ++transactionCount;
        }
}

/**
 * "Ausschwingphase" Does transactions for 1 minute WITHOUT counting!
 */
private void end() {
        long startTime = System.currentTimeMillis();
        while ((System.currentTimeMillis() - startTime) < endTime) {
                try {
                        transaction();
                        Thread.sleep(thinkTime);
                } catch (InterruptedException | SQLException e) {
                        e.printStackTrace();
                }
        }
}

/**
 * Either does "getBalance(accid)", "Deposit(...)" or "Analyse(delta)"
 * depending on a chance of 35/50/15 From "DBI: Aufgabenblatt 5": "Die
 * relative Gewichtung für die zufällige Auswahl der TXs sei dabei (35 zu 50
 * zu 15) für Kontostands-, Einzahlungs- und Analyse-TXs."
 *
```

```
         * @throws SQLException
         */
        @SuppressWarnings("static-access")
        private void transaction() throws SQLException {
                // Get us a random number between 0 and 100
                int chance = random.nextInt(100);
                // So it is decided what to do based on the Chance 35/50/15
                if (chance < 35) {
                        // Get the balance of a random accid
                        int accid = random.nextInt(100 * 100000) + 1;
                        dbmgmt.getBalance(accid);
                } else if (chance >= 35 && chance < 85) {
                        // Deposit random parameters on random accid
                        // "+ 1" because we dont want it to be 0
                        int accid = random.nextInt(100 * 100000) + 1;
                        int tellerid = random.nextInt(100 * 10) + 1;
                        int branchid = random.nextInt(100 * 1) + 1;
                        int delta = random.nextInt(10000) + 1;
                        dbmgmt.deposit(accid, tellerid, branchid, delta);
                } else {
                        // Analyse with random delta
                        int delta = random.nextInt(10000) + 1;
                        dbmgmt.analyse(delta);
                }

        }

        /**
         * Calculates and prints TPS based on transactionCount
         */
        private void tps() {
                // Calculates
                float tps = (float) transactionCount / (float) (benchmarkTime / 1000);
                // And prints
                System.out.println("TPS: " + tps);
                System.out.println("TXs: " + transactionCount);
        }
}
```