Westfälische Hochschule - Gelsenkirchen Bocholt Recklinghausen - 45877 Gelsenkirchen

Studiengang: Wirtschaftsinformatik Bachelor

Wintersemester 2016/17

3. Fachsemester

Datenbank- und Informationssysteme

Praktikumsaufgabe 7 - Benchmarking

Abgabedatum:

1. Dezember 2016

Aktualisierungsdatum:

14. Dezember 2016

Autoren:

Jonas Stadtler – jonas.stadler@studmail.w-hs.de
Mario Kellner – mario.kellner@studmail.w-hs.de
Markus Hausmann – markus.hausmann@studmail.w-hs.de

Inhaltsverzeichnis

Teilaufgabe A)

Entwickeln Sie ein Programm, das einen Aufrufparameter n erwartet und eine initiale n- tps-Datenbank auf dem gewählten Datenbankmanagementsystem erzeugt.

Um eine Verbindung zu unserem DBMS[1] PostgreSQL aufbauen zu können, müssen wir uns erst einmal den JDBC-Treiber für unser DBMS herunterladen. Der JDBC-Treiber befindet sich auf der Webseite von PostgreSQL[2].

Den Treiber binden wir nun in unserem Buildpath ein und erstellen uns eine Klasse DatabaseConnection, die den Treiber in der „getConnection“-Methode laden lässt.

```
1  /**
2   * Liest die Informatioen aus dem Objekt ci (ConnectionInformation) und
3   versucht eine Verbindung * zur Datenbank auf zu bauen.
4   *
5   * @throws SQLException Wird geworfen, wenn der DriverManager keine Verbndung zur Datenbank
        aufbauen kann
6   */
7  public void connect() throws SQLException
8  {
9    databaseLink = DriverManager.getConnection(
10       "jdbc:postgresql://" + ci.getHost() +"/" + ci.getDatabase(),
11       ci.getUser(),
12       ci.getPassword()
13   );
14 }
```

Quellcodeauszug 1: connect Funktion

Die Verbindungsinformationen übergeben wir mittels der Klasse ConnectionInformation. Diese implementiert ein einfaches Modell, welches nur die Parameter für die Datenbankverbindung beinhaltet.

Anschließend implementieren wir eine Klasse TpsCreator, welche die gesamte Funktionalität zum Erstellen der **n-tps-Datenbank** beinhaltet. Dieser Klasse kann man wiederum ein DatabaseConnection-Objekt übergeben. Durch diese Handhabung ist es möglich, in einem Programm, Datenbanken auf verschiedene Servern anzulegen.

Die Queries der Klasse TpsCreator führen wir anschließend mit normalen Statements aus.

```
1  /**
2   * Erstellt die Tupel in der Tabelle Branch
3   *
4   * @param n Anzahl der Tupel, die erstellt werden
5   * @return Gibt an, ob ein Fehler vorhanden ist
6   */
7  public boolean createBranchTupel(int n) {
8    try {
9      Statement statement = connection.databaseLink.createStatement();
10
11     for (int i = 1; i <= n; i++) {
12       statement.executeUpdate("INSERT INTO branches (branchid, branchname, balance, address)
             VALUES(" + i + ", 'branch', 0, 'branch')");
13     }
14     return true;
15   } catch (SQLException e) {}
16   return false;
17 }
```

Quellcodeauszug 2: Eine der drei Tupel Funktionen

[1] Datenbank Magement System
[2] https://jdbc.postgresql.org/download.html

Teilaufgabe A)

Als nächstes befassen wir uns mit der Benutzerinteraktion. Wir waren uns einig, dass wir uns anstelle einer komplexen GUI Darstellung auf eine simple Konsolen-Anwendung beschränken wollen. Denn diese Form der Nutzerinteraktion hat große Vorteile gegenüber der Performance, da wir nicht das Swing-Framework laden und initialisieren müssen und uns so Wertvolle Sekunden im Benchmarking sparen können.

Um die Interaktion mit der Konsole so einfach wie möglich zu gestalten, schreiben wir uns eine Helfer Klasse ConsoleReader und benutzen diese anschließend in unserer Main-Klasse.

```
1  System.out.println("Verbindungsinformationen eingeben!");
2
3  System.out.println("Host:");
4  infos.setHost(ConsoleReader.readString());
5
6  System.out.println("Datenbank:");
7  infos.setDatabase(ConsoleReader.readString());
8
9  System.out.println("Benutzer:");
10 infos.setUser(ConsoleReader.readString());
11
12 System.out.println("Password:");
13 infos.setPassword(ConsoleReader.readString());
```

Quellcodeauszug 3: Verbindungsinformationen per Konsole

Am Ende unserer Main-Klasse rufen wir die Funkion *autoSetup*, in dem initialisieren TpsCreator-Objekt, auf. Diese Funktion erstellt uns anschließend die Datenbank. Ein Durchlauf mit einer **10-tps-Datenbank** dauert ca. 4 Minuten und ist nicht performant.

Teilaufgabe B)

Welche Mindestgrößen schätzen Sie für eine 1-tps-Datenbank bzw. allgemein für eine n- tps-Datenbank? Wie viel Plattenplatz wird auf dem Datenbank-Server tatsächlich für die erstellten Datenbanken benötigt?

Die reine Spaltenanzahl lässt sich folgendermaßen ausrechnen:

$$n + n \cdot 10 + n \cdot 100.000 = \text{Spaltenanzahl} \tag{1}$$

Also sind bei einer 1-tps-Datenbank 100.011 Datensätze enthalten. Inklusive der vier Tabellen sind wir dann bei einer Anzahl von 100.015 Statements.

Die Größe der Datenbank kann man in zwei Kategorien.

- Relationale Größe
- Physikalische Größe

Folgender SQL-Query liefert uns die relationale-Größe jeder einzelner Tabelle:

```
1  SELECT nspname || '.' || relname AS relation,
2    pg_total_relation_size(C.oid) AS total_size
3  FROM pg_class C
4  LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
5  WHERE nspname NOT IN ('pg_catalog', 'information_schema')
6    AND C.relkind <> 'i'
7    AND nspname !~ '^pg_toast'
8  ORDER BY pg_total_relation_size(C.oid) DESC
```

Quellcodeauszug 4: Größe der Tabelle und Datensätze ermitteln

Wenn wir diesen Query ausführen, erhalten wir folgende Ausgabe:

relation	total_size
public.accounts	16.392.192 bytes
public.tellers	24.576 bytes
public.branches	24.576 bytes
public.history	0 bytes

Tabelle 1: Ausgabe des Query

Die so errechnete Größe liegt bei 16.441.344 bytes. Umgerechnet ergibt das: 15,68 MB.

Die tatsächliche physikalische Größe der gesamten Datenbank lässt sich mit folgendem Query bestimmen:

```
1  SELECT d.datname AS Name, pg_catalog.pg_get_userbyid(d.datdba) AS Owner, pg_catalog.
       pg_database_size(d.datname) AS SIZE FROM pg_catalog.pg_database d
2      ORDER BY pg_catalog.pg_database_size(d.datname) DESC
```

Quellcodeauszug 5: Physikalische Größe der Datenbank ermitteln

Die tatsächliche Größe der Datenbank beträgt dabei: 24.469.676 Bytes. Umgerechnet ist die Datenbank dementsprechend 23,34 MB groß.

Die Differenz beider Größen beträgt also 7,66 MB.

Teilaufgabe C)

Versuchen Sie, die Laufzeit Ihres Programms zu beschleunigen! Dokumentieren Sie einzelne Verbesserungsideen und die jeweiligen Laufzeitveränderungen für eine lokale Ausführung Ihres Programms bei der Erzeugung einer 10-tps-Datenbank!

Transaktion

Optimierung

Um den Durchsatz der Daten zu erhöhen, lassen sich die Queries in einer **Transaktion**[3] zusammenfassen.

Dazu muss die Option *autoCommit* abgeschaltet werden und die Transaktion mit einem manuellen *commit* zum Server geschickt werden. Wir mussten also unseren TpsCreator erweitern, in diesem legen wir dazu zwei neue Funktionen an:

```
1  public void beginTransaktion() {
2    try {
3      connection.databaseLink.setAutoCommit(false);
4    } catch (SQLException e) {
5      e.printStackTrace();
6    }
7  }
8
9  public void endAndCommitTransaction() {
10   try {
11     connection.databaseLink.commit();
12     connection.databaseLink.setAutoCommit(true);
13   } catch (SQLException e) {
14     e.printStackTrace();
15   }
16 }
```

Quellcodeauszug 6: Erweiterung der TpsCreator-Klasse

Jetzt brauchen wir noch die Funktionalität, um die Funktionen aufzurufen. Damit wir flexibel sind, haben wir hierfür ein Konsolen-Menü implementiert. Dieses erstellt uns dann eine Benchmark-Klasse, die der TpsCreator-Klasse alle nötigen Parameter übergibt und das Benchmarking ausführt.

```
1  public static int renderMenu() {
2    System.out.println("======================- Benchmark Menu -==========================");
3    System.out.println("= Wähle bitte eine der folgende Optionen:                 =");
4    System.out.println("= (1) Benchmark, Debug Log, incl. drop & create           =");
5    System.out.println("= (2) Benchmark, Debug Log, excl. drop & create           =");
6    System.out.println("= (3) Benchmark, Debug Log, transactions, incl. drop & create =");
7    System.out.println("= (4) Benchmark, Debug Log, transactions, excl. drop & create =");
8    System.out.println("= (5) Benchmark, incl. drop & create                      =");
9    System.out.println("= (6) Benchmark, excl. drop & create                      =");
10   System.out.println("= (7) Benchmark, transactions, incl. drop & create        =");
11   System.out.println("= (8) Benchmark, transactions, excl. drop & create        =");
12   System.out.println("====================================================================");
13   System.out.print("= Auswahl: ");
14
15   return ConsoleReader.readInt();
16 }
```

Quellcodeauszug 7: Renderer für das Konsolen Menü

[3] https://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html

Bewertung

Die Queries, die mittels einer Transaktion übertragen werden, werden bei einer **10-tps-Datenbank** um ca. das **11fache** schneller als die Queries die einzelnd zum Server geschickt werden.

Wenn man in der JDBC Bibliothek mit großen Datenmengen arbeitet, lohnt es sich performancemäßig dieses AutoCommit aus zu schalten. Der Nachteil, dass man eine Transaktion selbst abschließen muss, ist gegenüber dem Performancegewinn eher zu vernachlässigen.

Schlüsselbedingungen

Optimierung

In den create-Statements befinden sich Schlüsselbedingungen für die Primär- und Fremdschlüssel. Jeder Schlüssel erstellt auch einen sogenannten Index. Beim Einfügen der Daten in die jeweiligen Tabellen ist so ein Index hinderlich, da anhand dieser einige Bedingungen geprüft werden und somit wertvolle Sekunden im Benchmarking kostet.

Entkapselt man also diese und fügt sie erst am Ende des Benchmark-Vorgangs an, so braucht der Server diese Bedingungen nicht zu überprüfen. Wir müssen also eine Funktion schreiben, die am Ende jedes Benchmarks ausgeführt wird.

```
1  /**
2   * Alter Statements zum setzen der Keys
3   */
4
5  public String[] AlterStatements = {
6      "ALTER TABLE branches ADD PRIMARY KEY(branchid)",
7      "ALTER TABLE accounts ADD PRIMARY KEY(accid)",
8      "ALTER TABLE tellers ADD PRIMARY KEY(tellerid)",
9      "ALTER TABLE accounts ADD FOREIGN KEY(branchid) REFERENCES branches(branchid)",
10     "ALTER TABLE tellers ADD FOREIGN KEY(branchid) REFERENCES branches(branchid)",
11     "ALTER TABLE history ADD FOREIGN KEY(branchid) REFERENCES branches(branchid)",
12     "ALTER TABLE history ADD FOREIGN KEY(tellerid) REFERENCES tellers(tellerid)",
13     "ALTER TABLE history ADD FOREIGN KEY(accid) REFERENCES accounts(accid)"
14 };
15
16 /**
17  * Erstellt die Schlüasselbedingungen
18  *
19  * @return Gibt an, ob win Fehler vorhanden ist.
20  */
21 public boolean createKeys() {
22
23   try {
24     Statement statement = connection.databaseLink.createStatement();
25
26     for (String table : this.AlterStatements) {
27       statement.executeUpdate(table);
28       if(isDebug) {
29         System.out.println(table);
30       }
31     }
32
33     return true;
34   } catch (SQLException e) {
35     System.out.println(e.getMessage());
36     return false;
37   }
38 }
```

Quellcodeauszug 8: Alter Statements & Funktion createKeys

Bewertung

Durch die entkapselung der Schlüsselbedingungen sparen wir im Benchmarking einige Millisekunden. Insgesamt ist ein Performancezuwachs von 10 % messbar.

Diese Handhabung der Schlüsselbedingungen sollte jedoch nicht in einem Produktivsystem verwendet werden, denn so hindert man das DBMS etwaige Optimierungen an dem Datenbestand vorzunehmen. Dies äußert sich in längeren Anfragezeiten gegenüber Datenbank.

Implementierung des CopyManagers

Optimierung

Unser ausgewähltes DBMS PostgreSQL stellt ein **COPY**-Statement zur Verfügung. Dieses Statement erlaubt uns Daten aus einer Textdatei oder aber auch einer CSV Datei in unsere Datenbank zu importieren.

Um dieses Feature zu nutzen, stellt der JDBC-Treiber von PostgreSQL eine **CopyManager**[4]-Klasse zur Verfügung. Diesem Manager übergeben wir lediglich ein **COPY**-Statement:

```
CopyManager cpm = new CopyManager((BaseConnection) connection.databaseLink);

CopyIn ci = cpm.copyIn("COPY branches (branchid, branchname, balance, address) FROM STDIN WITH
    DELIMITER '|'");
```

Quellcodeauszug 9: Initialisierung des CopyManagers

Mit dem dadurch erzeugten **CopyIn**-Objekt ist es jetzt möglich dem CopyManager Daten zu übergeben. Das machen wir mittels der **writeToCopy**-Funktion:

```
ci.writeToCopy(sb.toString().getBytes(), 0, sb.length());
```

Quellcodeauszug 10: WriteToCopy-Funktion

Nach Beendigung des **COPY**-Vorgangs, muss nur noch die Funktion **endCopy** ausgeführt werden, um die Daten zum Server zu übertragen.

Nach Änderung create-Funktionen, in der Klasse TpsCreator, erhalten wir folgenden Code:

```
/**
 * Erstellt die Tupel in der Tabelle Accounts
 *
 * @param n Anzahl der Tupel, die erstellt werden mal 10000
 * @return Gibt an, ob ein Fehler vorhanden ist
 */
public boolean createAccountTupel(int n) {
  int localConst = n*10000;
  try {
    CopyManager cpm = new CopyManager((BaseConnection) connection.databaseLink);
    CopyIn ci = cpm.copyIn("COPY accounts(accid, NAME, balance, address, branchid) FROM STDIN
        WITH DELIMITER '|'");

    for (int i = 1; i <= localConst; i++) {
      StringBuilder sb = new StringBuilder("|aaaaaaaaaaaaaaaaaaaa|0|
          aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa|").append(
        ThreadLocalRandom.current().nextInt(1, n + 1)
      ).append("\n").insert(0, i);

      ci.writeToCopy(sb.toString().getBytes(), 0, sb.length());

      if(isDebug) {
        System.out.print(sb.toString());
      }
    }
      ci.endCopy();
    return true;
  } catch (SQLException e) {
    e.printStackTrace();
  }

  return false;
}
```

Quellcodeauszug 11: createAccountTupel mit CopyManager

[4] https://jdbc.postgresql.org/documentation/publicapi/org/postgresql/copy/CopyManager.html

Da bei dieser Variante viel mit Strings gearbeitet wird, erweist sich der Einsatz eines Stringbuilders als effektiv, anstatt der üblichen Verkettung, da der StringBuilder wesentlich performanter arbeitet als die Verkettung von Datentypen.

Bewertung

Das **COPY**-Statement ist dank seiner starken Spezialisierung auf den Anwendungszwecks um einiges schneller als die ansich schon schnellen *PreparedStatements*.

In Verbindung mit dem in JDBC-Treiber zur Verfügung stehenden CopyManagers, erlaubt uns das bei einer **10-tps-Datenbank** ein Benchmarkergebnis von 2500 Millisekunden. Damit sind wir um das **36fache** schneller als in der ursprünglichen Version.

Selbst Datenbank-Größen jenseits $n > 50$ laufen binnen weniger Sekunden problemlos durch.

Optimierungseinstellungen im DBMS

Optimierung

Um ein noch besseres Eregbnis zu erzielen kann man in PostgreSQL die *asyncronen Commits* aktivieren. Diese sind standardmäßig deaktiviert und müssen in der Konfigurationsdatei von PostgreSQL erst einmal aktiviert werden[5].

Unter Unix-Systemen liegt die Konfigurationsdatei, meistens unter
„*/etc/postgresql/{versionsnummer}/main/postgresql.conf*“

Aktivierte & geänderte Serverkonfiguration:

```
1 shared_buffers  = 1GB
2
3 #wal-writer Engine
4 synchronous_commit = off
5 commit_delay  = 50000
6 wal_writer_delay = 5000ms
7 wal_buffers   = 128MB
```

Veränderte Serverkonfiguration

shared_buffer beschreibt, wie viel Ram der Server belegen darf. Standardmäßig steht dieser Wert auf 128MB. Erhöht man diesen Wert, erhöht sich der Durchsatz der Daten maginal. Ein guter Wert ist $1/4$ des Server-Rams. In unseren Fall ist das dementsprechend 1GB.

Die „wal-writer“-Engine ist dafür verantwortlich, wie die Daten auf die Festplatte geschrieben werden. Der hohe **delay**-Wert beispielsweise, sorgt dafür, dass das Datenbanksystem nur alle 5 Sekunden 128MB auf die Festplatte schreiben darf. So zwingen wir unser DBMS vieles im Ram zu verwalten, was wesentlich performanter ist, da der Ram sehr geringe Zugriffszeiten besitzt. Wir laufen aber im Fehlerfall Gefahr Daten zu verlieren.

Bewertung

Der schlussendliche Performanceschub fällt je nach System unterschiedlich aus. So hat diese Konfiguration auf den Laptop eines Kommilitonen eine Einsparung von 100 Millisekunden bei einer **10-tps-Datenbank** eingebracht.

[5] https://www.postgresql.org/docs/current/static/wal-async-commit.html

Gesamter Performancezuwachs

Durch das schrittweise Optimieren des Programmcodes konnte ein betrachtlicher Performancezuwachs erzielt werden. Zwar ist es in der Praxis nicht immer möglich jede der hier aufgeführten Optimierungen zu nutzen, doch es zeigt was möglich ist, wenn man sich mit den jeweiligen Datenbanken und Datenbanktreibern beschäftigt.

Die folgende Tabelle zeigt den Performancezuwachs prozentual zur unoptimierten Programmversion an einer **10-tps-Datenbank**:

Optimierung	Zeit(ms)	Zuwachs(%)	gewonnene Zeit(ms)
Ausgangs-Programm	237.000	0	0
Transactions	20.277	1.168	216.723
Schlüsselbedingungen	18.543	1.278	218.457
CopyManager	4.520	5.243	232.480
Servereinstellungen	3.982	5.951	233.018

Tabelle 2: Leistungszuwachs durch Optimierungen

Teilaufgabe D)

Messen und protokollieren Sie die Laufzeit ihres optimierten Programms für $n = 10$, $n = 20$ und $n = 50$ sowohl lokal auf dem Datenbank-Server als auch von einem „remote“ Client! (Gemessen werden soll nur die Laufzeit zum Einfügen ohne evtl. notwendige vorherige DROP-TABLE- oder CREATE-TABLE-Anweisungen!)

Folgende Werte wurden bei dem Benchmarking ermittelt:

Typ	$n = 10$	$n = 20$	$n = 50$
Lokaler Server	4.834 ms	7.835 ms	15.807 ms
Remote Server	9.425 ms	18.724 ms	46.267 ms

Tabelle 3: Benchmark Ergebnisse

Quellcodeverzeichnis

Tabellenverzeichnis

Anhang

Programm (optimiert)

```java
/**
 *
 */
package de.whs.dbi.pa7;

import de.whs.dbi.pa7.database.ConnectionInformation;
import de.whs.dbi.pa7.database.DatabaseConnection;
import de.whs.dbi.pa7.database.TpsCreator;
import de.whs.dbi.pa7.database.TpsCreatorInterface;
import de.whs.dbi.pa7.database.TpsCreatorOldStatements;

/**
 * Unsere Hauptklasse. Hauptsächlich dient Sie als Einsteigspunkt unserer Applikation
 * und bietet gleichzeitig dem Benutzer eine Interaktionsmöglichkeit
 * über die Konsole an.
 *
 * @author Mario Kellner
 * @author Markus Hausmann
 * @author Jonas Stadtler
 *
 */
public class Main {
  /**
   * Datenbank Verbindungsobjekt
   */
  static DatabaseConnection con;

  /**
   * Ersteller der Datenbank
   */
  static TpsCreatorInterface tpsDBCreator;

  /**
   * Vom Benutzer angegebene Größe der tps-Datenbank
   */
  static int sizeN;

  /**
   * Gibt an, ob die heardcodierten Verbindungsinformationen benutzt werden.
   */
  static boolean useLocal = false;

  /**
   * Unserer Einstiegspunkt füt unsere Anwendung.
   * Hier werden die nötigen Informationen von dem Benutzer abgefragt
   * und der Benchmark Klasse übergeben.
   *
   *
   * @param args Wird irgoriert
   */

  public static void main(String[] args) {

    System.out.println("Lokale Datenbank verwenden? [j/n]:");
    useLocal = ConsoleReader.readJn();

    ConnectionInformation infos = new ConnectionInformation();

    if(useLocal) {
      // Lokale Informationen unserer Gruppe.
      // Diese Zugangsdaten werden nicht bei einer anderen Gruppe funktionieren.
      infos.setHost("127.0.0.1");
      //infos.setHost("192.168.122.70");
      infos.setDatabase("benchmark");
      infos.setUser("postgres");
      infos.setPassword("DBIPr");
    }
    else {

```

```
70      //Verbindungnsinformationen werden vom User angegeben.
71      System.out.println("Verbindungsinformationen eingeben!");
72
73      System.out.println("Host:");
74      infos.setHost(ConsoleReader.readString());
75
76      System.out.println("Datenbank:");
77      infos.setDatabase(ConsoleReader.readString());
78
79      System.out.println("Benutzer:");
80      infos.setUser(ConsoleReader.readString());
81
82      System.out.println("Password:");
83      infos.setPassword(ConsoleReader.readString());
84    }
85
86
87    try {
88      // erstelle die Datenbankverbindung und verbinde
89      System.out.print("Verbinde zur Datenbank ... ");
90
91      con = new DatabaseConnection(infos);
92      con.connect();
93
94      System.out.println("[OK]");
95
96      // Nutzer fragen, ob er die unoptimierte Programmversion nutzen möchte
97      System.out.println("Optimierte Programmversion benutzen?? [j/n]:");
98
99      // erstellen des jeweiligen TpsCreatorObjekts
100     if(!ConsoleReader.readJn()) {
101       tpsDBCreator = new TpsCreatorOldStatements(con);
102     }
103     else {
104
105       tpsDBCreator = new TpsCreator(con);
106     }
107   }
108   catch ( Exception err){
109
110     //Fehlerfall
111     err.printStackTrace();
112     System.err.println(err.getMessage());
113     System.err.println("Konnte keine Verbindung zur Datenbank aufbauen!");
114
115     return;
116   }
117
118
119   // Nutzer die größe der Datenbank angeben lassen
120   System.out.println("Gebe bitte die Größe (n) der Datenbank ein:");
121   sizeN = ConsoleReader.readInt();
122
123   // Zeige das Benchmarkmenü
124   benchmarkMenu();
125
126   System.out.println("Beende Anwendung...");
127 }
128
129 /**
130  * Zeigt das Benchmarkmenü und wertet die Nutzereingaben
131  * aus.
132  */
133 public static void benchmarkMenu() {
134   int menureturn = renderMenu();
135
136   // Übergabe der Benchmarkparameter
137   switch(menureturn) {
138   case 1:
139     Benchmark.BenchmarkTask(sizeN, tpsDBCreator, true, false, true);
140     break;
141   case 2:
142     Benchmark.BenchmarkTask(sizeN, tpsDBCreator, true, false, false);
143     break;
144   case 3:
145     Benchmark.BenchmarkTask(sizeN, tpsDBCreator, true, true, true);
```

```
146       break;
147     case 4:
148       Benchmark.BenchmarkTask(sizeN, tpsDBCreator, true, true, false);
149       break;
150     case 5:
151       Benchmark.BenchmarkTask(sizeN, tpsDBCreator, false, false, true);
152       break;
153     case 6:
154       Benchmark.BenchmarkTask(sizeN, tpsDBCreator, false, false, false);
155       break;
156     case 7:
157       Benchmark.BenchmarkTask(sizeN, tpsDBCreator, false, true, true);
158       break;
159     case 8:
160       Benchmark.BenchmarkTask(sizeN, tpsDBCreator, false, true, false);
161       break;
162     default:
163       System.out.println("Ungültige Menü Option");
164     }
165   }
166
167
168   /**
169    * Rendert das Benchmarkmenü.
170    *
171    * @return Nutzereingabe
172    */
173   public static int renderMenu() {
174     System.out.println("======================- Benchmark Menu -==========================");
175     System.out.println("= Wähle bitte eine der folgende Optionen:                 =");
176     System.out.println("= (1) Benchmark, Debug Log, incl. drop & create           =");
177     System.out.println("= (2) Benchmark, Debug Log, excl. drop & create           =");
178     System.out.println("= (3) Benchmark, Debug Log, transactions, incl. drop & create =");
179     System.out.println("= (4) Benchmark, Debug Log, transactions, excl. drop & create =");
180     System.out.println("= (5) Benchmark, incl. drop & create                      =");
181     System.out.println("= (6) Benchmark, excl. drop & create                      =");
182     System.out.println("= (7) Benchmark, transactions, incl. drop & create        =");
183     System.out.println("= (8) Benchmark, transactions, excl. drop & create        =");
184     System.out.println("====================================================================");
185     System.out.print("= Auswahl: ");
186
187     return ConsoleReader.readInt();
188   }
189 }
```

Quellcodeauszug 12: Main (optimiert)

```
1  package de.whs.dbi.pa7;
2
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.io.InputStreamReader;
6
7  /**
8   * Einige Konsolen Helfer. Die Klasse ist con der BasicIO.java aus dem GDI1 Kurs adaptiert
9   *
10  * @author Mario Kellner
11  * @author Markus Hausmann
12  * @author Jonas Stadtler
13  */
14 public class ConsoleReader {
15
16   /**
17    * Liest ein j/J oder ein n/N aus der Konsole
18    *
19    * @return j = true und n = false
20    */
21   public static boolean readJn() {
22     BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
23
24     try {
25       String strTmp = br.readLine();
26       if(strTmp.length() != 0 && (strTmp.charAt(0) == 'j' || strTmp.charAt(0) == 'J')) {
27         return true;
28       }
```

```
    }
    catch(Exception err) {}

    return false;
  }

  /**
   * Liest einen String aus der Konsole aus
   *
   * @return ausgelesener String
   */
  public static String readString () {
    BufferedReader din = new BufferedReader(new InputStreamReader(System.in));

    try {
      return din.readLine();
    }
    catch(IOException e) {
      return "Ein-/Ausgabe-Fehler";
    }
  }

  /**
   * Liest einen Integer aus der Konsole aus
   *
   * @return ausgelesener String
   */
  public static int readInt()
  {
    try
    {
      BufferedReader din = new BufferedReader(new InputStreamReader(System.in));
      return Integer.parseInt(din.readLine());
    }
    catch (NumberFormatException e)
    {
      System.out.println("Bitte gebe eine gültige Zahl ein.");
      return readInt();
    }
    catch (IOException e)
    {
      return -1;
    }
  }
}
```

Quellcodeauszug 13: ConsoleReader (optimiert)

```
package de.whs.dbi.pa7;

import java.io.PrintStream;

import de.whs.dbi.pa7.database.TpsCreatorInterface;

public class Benchmark {

  /**
   * Globale Startzeit des Benchmarks
   */
  private static long startTime;

  /**
   * Führt den Benchmark mit den Übergebenen Parametern durch.
   *
   * @param size Die geplante Größe der n-tps-Datenbank
   * @param tpsDBCreator Das Objekt, welches die Datenbank erzeugen kann
   * @param debugLog Aktiviert das Debug Logging (SQL-Queries)
   * @param useTransaction Gibt an, ob SQL-Transaktionen für die Datenbank ausgeführt werden sollen
   * @param usePreamble Gibt an, ob die drop- und createstatements mit ins Benchmark mit einfließen sollen
   */
  public static void BenchmarkTask(int size, TpsCreatorInterface tpsDBCreator, boolean debugLog,
      boolean useTransaction, boolean usePreamble) {
    tpsDBCreator.setDebug(debugLog);
```

```
25      tpsDBCreator.createFiles(size);
26
27      System.out.println("Brenchmarking gestartet");
28      System.out.println("Parameter: debugLog = " + debugLog + ", useTransaction = " +
            useTransaction + ", usePreamble = " + usePreamble);
29
30
31      if(!usePreamble) {
32        startTimer();
33        if(useTransaction) {
34          tpsDBCreator.beginTransaktion();
35        }
36      }
37
38      tpsDBCreator.dropSchema();
39      tpsDBCreator.createSchema();
40
41      if(usePreamble) {
42        startTimer();
43        if(useTransaction) {
44          tpsDBCreator.beginTransaktion();
45        }
46      }
47
48      long localTime = System.currentTimeMillis();
49      tpsDBCreator.createBranchTupel(size);
50      System.out.println("Branches angelegt! Dauer " + (System.currentTimeMillis() - localTime) + "
            Millisekunden.");
51
52      localTime = System.currentTimeMillis();
53      tpsDBCreator.createAccountTupel(size);
54      System.out.println("Accounts angelegt! Dauer " + (System.currentTimeMillis() - localTime) + "
            Millisekunden.");
55
56      localTime = System.currentTimeMillis();
57      tpsDBCreator.createTellerTupel(size);
58      System.out.println("Teller angelegt! Dauer " + (System.currentTimeMillis() - localTime) + "
            Millisekunden.");
59
60      tpsDBCreator.finishUp();
61
62      if(useTransaction) {
63        tpsDBCreator.endAndCommitTransaction();
64      }
65
66      System.out.println("Benchmark abgeschlossen. Gesamtdauer " + getCurrentMilliseconds() + "
            Millisekunden.");
67
68
69    }
70
71    /**
72     * Setzt die Startzeit auf die jetzige Zeit
73     */
74    public static void startTimer() {
75      startTime = System.currentTimeMillis();
76    }
77
78    /**
79     * Gibt die Differenz aus der Startzeit und der jetzigen Zeit aus
80     *
81     * @return Differenz der Subtraktion
82     */
83    public static long getCurrentMilliseconds() {
84      return System.currentTimeMillis() - startTime;
85    }
86  }
```

Quellcodeauszug 14: Benchmark (optimiert)

```
1   package de.whs.dbi.pa7.database;
2
3   import java.sql.Connection;
4   import java.sql.DriverManager;
5   import java.sql.SQLException;
```

```

/**
 * Unsere Datebankklasse ermöglicht die Kommunikation
 * mit der Datenbank. Es ist ein Wrapper für das
 * Connection-Objekt der JDBC Connection
 *
 * @author Mario Kellner
 * @author Markus Hausmann
 * @author Jonas Stadtler
 */
public class DatabaseConnection
{
  /**
   * Das eigentliche Connection Objekt
   */
  public Connection databaseLink;

  /**
   *  Unser Verbindungs-Model
   */
  private ConnectionInformation ci;

  /**
   * Der Konstruktur der Klasse. Überprüft ob das Objekt im C
   *
   * @param cInfo Das Connection Objekt
   * @throws Exception Wird geworfen, wenn das Objekt was übergeben wird null ist.
   */
  public DatabaseConnection(ConnectionInformation cInfo) throws Exception {
    if(cInfo == null) {
      throw new Exception("ConnectionInfo Object cannot be null");
    }

    ci = cInfo;
  }

  /**
   *
   * List die Informatioen aus dem Objekt ci (ConnectionInformation) und versucht eine Verbindung
   * zur Datenbank auf zu bauen.
   *
   * @throws SQLException Wird geworfen, wenn der DriverManager keine Verbndung zur Datenbank
       aufbauen kann
   */
  public void connect() throws SQLException
  {
    /*
     * Der Compiler übersetzt " + var + " automatisch in ein StringBuilder Objekt
     */
    databaseLink = DriverManager.getConnection(
        "jdbc:postgresql://" + ci.getHost() +"/" + ci.getDatabase() + "?useCompression=true",
        ci.getUser(),
        ci.getPassword()
    );
  }
}
```

Quellcodeauszug 15: DatabaseConnection (optimiert)

```
package de.whs.dbi.pa7.database;

/**
 * Dieses Model beinhaltet alle nötigen Informationen
 * um sich zu einer Postgre Datenbank zu verbinden
 *
 * @author Mario Kellner
 * @author Markus Hausmann
 * @author Jonas Stadtler
 *
 */
public class ConnectionInformation {

  // Eigenschaften
  public String host = null;
  public String database = null;
```

```
17    public String user = null;
18    public String password = null;
19
20
21    // Getter & Setter
22    public String getHost() {
23      return host;
24    }
25    public void setHost(String host) {
26      this.host = host;
27    }
28    public String getDatabase() {
29      return database;
30    }
31    public void setDatabase(String database) {
32      this.database = database;
33    }
34    public String getUser() {
35      return user;
36    }
37    public void setUser(String user) {
38      this.user = user;
39    }
40    public String getPassword() {
41      return password;
42    }
43    public void setPassword(String password) {
44      this.password = password;
45    }
46  }
```

Quellcodeauszug 16: ConnectionInformation (optimiert)

```
1   package de.whs.dbi.pa7.database;
2
3   import java.io.ByteArrayInputStream;
4   import java.io.File;
5   import java.io.FileReader;
6   import java.io.FileWriter;
7   import java.io.IOException;
8   import java.io.StringReader;
9   import java.sql.SQLException;
10  import java.sql.Statement;
11  import java.util.concurrent.ThreadLocalRandom;
12
13  import org.postgresql.copy.CopyIn;
14  import org.postgresql.copy.CopyManager;
15  import org.postgresql.core.BaseConnection;
16
17  /**
18   * Die tpsCreator Klasse enthält alle nötigen Befehle um eine initial n-tps-Datenbank
19   * erstellen zu können. Hier kann auserdem ein instanziiertes Verbindungsobjekt übergeben werden,
20   * um auf unterschiedliche Datenbanken gleichzeitig arbeiten zu können
21   *
22   * @author Mario Kellner
23   * @author Markus Hausmann
24   * @author Jonas Stadtler
25   */
26  public class TpsCreator implements TpsCreatorInterface {
27
28    /**
29     * Debug Variable
30     */
31    private boolean isDebug = false;
32
33    /**
34     * Das Verbindungsobjekt
35     */
36    private DatabaseConnection connection;
37
38    /**
39     * Tabellen die in dem Shema enthalten sind, momentan nur nötig zum "droppen" der Datenbank
40     */
41    public String[] tables = {"branches", "accounts", "tellers", "history"};
42
```

```
43  /**
44   * Die Create-Statements für das Schema
45   */
46  public String[] schema = {
47      "CREATE TABLE branches (branchid INT NOT NULL, branchname CHAR(20) NOT NULL, balance INT NOT
            NULL, address CHAR(72) NOT NULL)",
48      "CREATE TABLE accounts (accid INT NOT NULL, NAME CHAR(20) NOT NULL, balance INT NOT NULL,
            branchid INT NOT NULL, address CHAR(68) NOT NULL)",
49      "CREATE TABLE tellers (tellerid INT NOT NULL, tellername CHAR(20) NOT NULL, balance INT NOT
            NULL, branchid INT NOT NULL, address CHAR(68) NOT NULL)",
50      "CREATE TABLE history (accid INT NOT NULL, tellerid INT NOT NULL, delta INT NOT NULL,
            branchid INT NOT NULL, accbalance INT NOT NULL, cmmnt CHAR(30) NOT NULL)",
51  };
52
53  /**
54   * Alter Statements zum setzen der Keys
55   */
56  public String[] AlterStatements = {
57      "ALTER TABLE branches ADD PRIMARY KEY(branchid)",
58      "ALTER TABLE accounts ADD PRIMARY KEY(accid)",
59      "ALTER TABLE tellers ADD PRIMARY KEY(tellerid)",
60      "ALTER TABLE accounts ADD FOREIGN KEY(branchid) REFERENCES branches(branchid)",
61      "ALTER TABLE tellers ADD FOREIGN KEY(branchid) REFERENCES branches(branchid)",
62      "ALTER TABLE history ADD FOREIGN KEY(branchid) REFERENCES branches(branchid)",
63      "ALTER TABLE history ADD FOREIGN KEY(tellerid) REFERENCES tellers(tellerid)",
64      "ALTER TABLE history ADD FOREIGN KEY(accid) REFERENCES accounts(accid)"
65  };
66
67
68  public String fileBranches;
69  public String fileAccounts;
70  public String fileTellers;
71
72  /**
73   * Unser Konstruktor. Er Überprüft, ob ein nicht leeres DatabaseConnection Objekt übergeben
          worden ist.
74   *
75   *
76   * @param connection Das Verbindungsobjekt
77   * @throws NullPointerException Das Verbindungsobjekt ist leer
78   */
79  public TpsCreator(DatabaseConnection connection) throws Exception {
80    System.out.println("Info: Optimierte Version wird benutzt!");
81
82    if(connection == null ) {
83      throw new NullPointerException("connection object cannot be null");
84    }
85
86    try {
87      connection.databaseLink.isValid(0);
88    }
89    catch(Exception err) {
90      throw err;
91    }
92
93    this.connection = connection;
94
95  }
96
97  /**
98   * Aktiviert das Transaktion-System.
99   */
100 public void beginTransaktion() {
101   try {
102     connection.databaseLink.setAutoCommit(false);
103   } catch (SQLException e) {
104     e.printStackTrace();
105   }
106 }
107
108 /**
109  * Comitted die Queries zur DAtenbank und beendet das Transaktion-System
110  *
111  * @return Gibt an, ob win Fehler vorhanden ist.
112  */
113 public void endAndCommitTransaction() {
```

```
114     try {
115       connection.databaseLink.commit();
116       connection.databaseLink.setAutoCommit(true);
117     } catch (SQLException e) {
118       e.printStackTrace();
119     }
120   }
121
122   /**
123    * Löscht die Tabellen aus der Datenbank
124    *
125    * @return Gibt an, ob win Fehler vorhanden ist.
126    */
127   public boolean dropSchema() {
128     try {
129       Statement statement = connection.databaseLink.createStatement();
130
131       for (String table : this.tables) {
132         StringBuilder strBuilder = new StringBuilder("DROP TABLE IF EXISTS ").append(table).append
              (" CASCADE");
133         statement.executeUpdate(strBuilder.toString());
134
135         if(isDebug) {
136           System.out.println(strBuilder.toString());
137         }
138       }
139
140       return true;
141     } catch (SQLException e) {
142       e.printStackTrace();
143       return false;
144     }
145
146   }
147
148   /**
149    * Erstellt die Tabellen
150    *
151    * @return Gibt an, ob win Fehler vorhanden ist.
152    */
153   public boolean createSchema() {
154
155     try {
156       Statement statement = connection.databaseLink.createStatement();
157
158       for (String table : this.schema) {
159         statement.executeUpdate(table);
160         if(isDebug) {
161           System.out.println(table);
162         }
163       }
164
165       return true;
166     } catch (SQLException e) {
167       return false;
168     }
169   }
170
171   /**
172    * Erstellt die Tabellen
173    *
174    * @return Gibt an, ob win Fehler vorhanden ist.
175    */
176   public boolean createKeys() {
177
178     try {
179       Statement statement = connection.databaseLink.createStatement();
180
181       for (String table : this.AlterStatements) {
182         statement.executeUpdate(table);
183         if(isDebug) {
184           System.out.println(table);
185         }
186       }
187
188       return true;
```

```
189     } catch (SQLException e) {
190       System.out.println(e.getMessage());
191       return false;
192     }
193   }
194
195   /**
196    * Erstellt die Tupel in der Tabelle Branch.
197    *
198    * @param n Anzahl der Tupel, die erstellt werden
199    * @return Gibt an, ob ein Fehler vorhanden ist
200    */
201   public boolean createBranchTupel(int n) {
202     try {
203
204       CopyManager cpm = new CopyManager((BaseConnection) connection.databaseLink);
205       long ci = cpm.copyIn("COPY branches (branchid, branchname, balance, address) FROM STDIN WITH
            DELIMITER '|'", new FileReader("branches.txt"));
206
207       return true;
208     } catch (SQLException e) {
209       e.printStackTrace();
210     } catch (IOException e) {
211       // TODO Auto-generated catch block
212       e.printStackTrace();
213     }
214
215     return false;
216   }
217
218   /**
219    * Erstellt die Tupel in der Tabelle Accounts
220    *
221    * @param n Anzahl der Tupel, die erstellt werden mal 10000
222    * @return Gibt an, ob ein Fehler vorhanden ist
223    */
224   public boolean createAccountTupel(int n) {
225     try {
226       CopyManager cpm = new CopyManager((BaseConnection) connection.databaseLink);
227       long ci = cpm.copyIn("COPY accounts(accid, NAME, balance, address, branchid) FROM STDIN WITH
            DELIMITER '|'", new FileReader("accounts.txt"));
228
229       return true;
230     } catch (SQLException e) {
231       e.printStackTrace();
232     } catch (IOException e) {
233       // TODO Auto-generated catch block
234       e.printStackTrace();
235     }
236
237     return false;
238   }
239
240   /**
241    * Erstellt die Tupel in der Tabelle Teller
242    *
243    * @param n Anzahl der Tupel, die erstellt werden mal 10
244    * @return Gibt an, ob ein Fehler vorhanden ist
245    */
246   public boolean createTellerTupel(int n) {
247     try {
248       CopyManager cpm = new CopyManager((BaseConnection) connection.databaseLink);
249       long ci = cpm.copyIn("COPY tellers (tellerid, tellername, balance, address, branchid) FROM
            STDIN WITH DELIMITER '|'", new FileReader("tellers.txt"));
250       return true;
251     } catch (SQLException e) {
252       e.printStackTrace();
253     } catch (IOException e) {
254       // TODO Auto-generated catch block
255       e.printStackTrace();
256     }
257
258     return false;
259   }
260
261   /**
```

```java
 * Bündelt die Funktionen des tpsCreators und führt diese nach ein ander aus.
 *
 * @param n Anzahl der Tupel, die erstellt werden
 */
public void autoSetup(int n) {
  System.out.println("Starte automatisches Setup...");
  System.out.println("Erzeuge eine " + n + "-tps-Datenbank");

  if(!dropSchema()) {
    System.out.println("Warnung: Datenbank konnte nicht geleert werden!");
    return;
  }
  if(!createSchema()) {
    System.out.println("Tabellen konnten nicht erzeugt werden. Stoppe Setup ...");
    return;
  }

  if(!createBranchTupel(n)) {
    System.out.println("Branches konnten nicht angelegt werden. Stoppe Setup ...");
    return;
  }
  if(!createAccountTupel(n)) {
    System.out.println("Accounts konnten nicht angelegt werden. Stoppe Setup ...");
    return;
  }
  if(!createTellerTupel(n)) {
    System.out.println("Teller konnten nicht angelegt werden. Stoppe Setup ...");
    return;
  }

  System.out.println("Erstellen der " + n + "-tps-Datenbak erfolgreich");
}

/**
 * Gibt an, ob das DebugLogging aktiviert ist.
 */
public boolean isDebug() {
  return isDebug;
}

/**
 * Aktivert bzw deaktiviert den Debug Modus
 */
public void setDebug(boolean isDebug) {
  this.isDebug = isDebug;
}

@Override
public void finishUp() {
  this.createKeys();
}

public void createFiles(int size) {
  System.out.println("Erstelle Benchmark Datein!");

  FileWriter fw1;
  try {
    System.out.print("Erstelle Branches ...");

    fw1 = new FileWriter(new File("branches.txt"));
    for (int i = 1; i <= size; i++) {

      fw1.write(i + "|aaaaaaaaaaaaaaaaaaaa|0|
          aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\n");

    }
    fw1.close();

    System.out.println("[OK]");
    System.out.print("Erstelle Accounts ...");
    fw1 = new FileWriter(new File("accounts.txt"));
    for (int i = 1; i <= size*100000; i++) {

       fw1.write(i + "|aaaaaaaaaaaaaaaaaaaa|0|
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa|" +
           ThreadLocalRandom.current().nextInt(1, size + 1) + "\n");
```

```
      }

      fw1.close();

      System.out.println("[OK]");
      System.out.print("Erstelle Tellers ...");

      fw1 = new FileWriter(new File("tellers.txt"));
      for (int i = 1; i <= size*10; i++) {

        fw1.write(i +"|aaaaaaaaaaaaaaaaaaaa|0|
            aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa|" +
           ThreadLocalRandom.current().nextInt(1, size + 1) +
        "\n");
      }

      fw1.close();
      System.out.println("[OK]");

    } catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }

  }
}
```

Quellcodeauszug 17: tpsCreator (optimiert)

```
package de.whs.dbi.pa7.database;

public interface TpsCreatorInterface {
  void beginTransaktion();
  void endAndCommitTransaction();
  boolean dropSchema();
  boolean createSchema();
  boolean createBranchTupel(int n);
  boolean createAccountTupel(int n);
  boolean createTellerTupel(int n);
  void autoSetup(int n);
  boolean isDebug();
  void setDebug(boolean isDebug);
  public void finishUp();
  public void createFiles(int size);
}
```

Quellcodeauszug 18: TpsCreatorInterface (optimiert)

```
package de.whs.dbi.pa7.database;

import java.sql.SQLException;
import java.sql.Statement;
import java.util.concurrent.ThreadLocalRandom;

/**
 * Die tpsCreator Klasse enthält alle nötigen Befehle um eine initial n-tps-Datenbank
 * erstellen zu können. Hier kann auserdem ein instanziiertes Verbindungsobjekt übergeben werden,
 * um auf unterschiedliche Datenbanken gleichzeitig arbeiten zu können
 *
 * @author Mario Kellner
 * @author Markus Hausmann
 * @author Jonas Stadtler
 */
public class TpsCreatorOldStatements implements TpsCreatorInterface {
  private boolean isDebug = false;

  /**
   * Das Verbindungsobjekt
   */
  private DatabaseConnection connection;

  /**
   * Tabellen die in dem Shema enthalten sind, momentan nur nötig zum "droppen" der Datenbank
   */
  public String[] tables = { "branches", "accounts", "tellers", "history"};
```

```
28
29  /**
30   * Die Create-Statements für das Schema
31   */
32  public String[] schema ={
33      "CREATE TABLE branches (branchid INT NOT NULL, branchname CHAR(20) NOT NULL, balance INT NOT
            NULL, address CHAR(72) NOT NULL, PRIMARY KEY (branchid))",
34      "CREATE TABLE accounts ( accid INT NOT NULL, NAME CHAR(20) NOT NULL, balance INT NOT NULL,
            branchid INT NOT NULL, address CHAR(68) NOT NULL, PRIMARY KEY (accid), FOREIGN KEY (
            branchid) REFERENCES branches )",
35      "CREATE TABLE tellers ( tellerid INT NOT NULL, tellername CHAR(20) NOT NULL, balance INT NOT
            NULL, branchid INT NOT NULL, address CHAR(68) NOT NULL, PRIMARY KEY (tellerid), FOREIGN
            KEY (branchid) REFERENCES branches)",
36      "CREATE TABLE history ( accid INT NOT NULL, tellerid INT NOT NULL, delta INT NOT NULL,
            branchid INT NOT NULL, accbalance INT NOT NULL, cmmnt CHAR(30) NOT NULL, FOREIGN KEY (
            accid) REFERENCES accounts, FOREIGN KEY (tellerid) REFERENCES tellers, FOREIGN KEY (
            branchid) REFERENCES branches )",
37  };
38
39
40  /**
41   * Unser Konstruktor. Er Überprüft, ob ein nicht leeres DatabaseConnection Objekt übergeben
         worden ist.
42   *
43   *
44   * @param connection Das Verbindungsobjekt
45   * @throws NullPointerException Das Verbindungsobjekt ist leer
46   */
47  public TpsCreatorOldStatements(DatabaseConnection connection) throws Exception {
48    System.out.println("Info: Unoptimierte Version wird benutzt!");
49    if(connection == null ) {
50      throw new NullPointerException("connection object cannot be null");
51    }
52
53    try {
54      connection.databaseLink.isValid(0);
55      //connection.databaseLink.setAutoCommit(false);
56    }
57    catch(Exception err) {
58      throw err;
59    }
60
61    this.connection = connection;
62
63  }
64
65  /**
66   * Aktiviert das Transaktion-System.
67   */
68  public void beginTransaktion() {
69    try {
70      connection.databaseLink.setAutoCommit(false);
71    } catch (SQLException e) {
72      e.printStackTrace();
73    }
74  }
75
76  /**
77   * Comitted die Queries zur DAtenbank und beendet das Transaktion-System
78   *
79   * @return Gibt an, ob win Fehler vorhanden ist.
80   */
81  public void endAndCommitTransaction() {
82    try {
83      connection.databaseLink.commit();
84      connection.databaseLink.setAutoCommit(true);
85    } catch (SQLException e) {
86      e.printStackTrace();
87    }
88  }
89
90  /**
91   * Löscht die Tabellen aus der Datenbank
92   *
93   * @return Gibt an, ob ein Fehler vorhanden ist.
94   */
```

```
95  public boolean dropSchema() {
96    try {
97      Statement statement = connection.databaseLink.createStatement();
98
99      for (String table : this.tables) {
100       statement.executeUpdate("DROP TABLE IF EXISTS " + table + " CASCADE");
101       if(isDebug) {
102         System.out.println("DROP TABLE IF EXISTS " + table + " CASCADE");
103       }
104     }
105
106     return true;
107   } catch (SQLException e) {
108     e.printStackTrace();
109     return false;
110   }
111
112 }
113
114 /**
115  * Erstellt die Tabellen
116  *
117  * @return Gibt an, ob win Fehler vorhanden ist.
118  */
119 public boolean createSchema() {
120
121   try {
122     Statement statement = connection.databaseLink.createStatement();
123
124     for (String table : this.schema) {
125       statement.executeUpdate(table);
126       if(isDebug) {
127         System.out.println(table);
128       }
129     }
130
131     return true;
132   } catch (SQLException e) {
133     return false;
134   }
135 }
136
137
138 /**
139  * Erstellt die Tupel in der Tabelle Branch
140  *
141  * @param n Anzahl der Tupel, die erstellt werden
142  * @return Gibt an, ob ein Fehler vorhanden ist
143  */
144 public boolean createBranchTupel(int n) {
145   try {
146     Statement statement = connection.databaseLink.createStatement();
147
148     for (int i = 1; i <= n; i++) {
149       statement.executeUpdate("INSERT INTO branches (branchid, branchname, balance, address)
              VALUES(" + i + ", 'branch', 0, 'branch')");
150
151       if(isDebug) {
152         System.out.println("INSERT INTO branches (branchid, branchname, balance, address) VALUES
              (" + i + ", 'branch', 0, 'branch')");
153       }
154     }
155
156     return true;
157   } catch (SQLException e) {
158     e.printStackTrace();
159   }
160
161   return false;
162 }
163
164
165 /**
166  * Erstellt die Tupel in der Tabelle Accounts
167  *
168  * @param n Anzahl der Tupel, die erstellt werden mal 10000
```

```
169     * @return Gibt an, ob ein Fehler vorhanden ist
170     */
171    public boolean createAccountTupel(int n) {
172      int localConst = n*100000;
173      int localRandom;
174
175      try {
176
177        Statement statement = connection.databaseLink.createStatement();
178
179        for (int i = 1; i <= localConst; i++) {
180
181          // Generiert eine zufällige ID zwischen 1 und n
182          localRandom = ThreadLocalRandom.current().nextInt(1, n + 1);
183
184          statement.executeUpdate("INSERT INTO accounts (accid, NAME, balance, address, branchid)
                 VALUES("
185            + i + ", 'account', 0,'test', " + localRandom + ")");
186
187          if(isDebug) {
188            System.out.println("INSERT INTO accounts (accid, NAME, balance, address, branchid) "
189                + "VALUES(" + i + ", 'account', 0,'test', " + localRandom + ")");
190          }
191        }
192
193        return true;
194      } catch (SQLException e) {
195        e.printStackTrace();
196      }
197
198      return false;
199    }
200
201    /**
202     * Erstellt die Tupel in der Tabelle Teller
203     *
204     * @param n Anzahl der Tupel, die erstellt werden mal 10
205     * @return Gibt an, ob ein Fehler vorhanden ist
206     */
207    public boolean createTellerTupel(int n) {
208      int localConst = n*10;
209      int localRandom;
210
211      try {
212
213        Statement statement = connection.databaseLink.createStatement();
214
215        for (int i = 1; i <= localConst; i++) {
216
217          // Generiert eine zufällige ID zwischen 1 und n
218          localRandom = ThreadLocalRandom.current().nextInt(1, n + 1);
219
220          statement.executeUpdate("INSERT INTO tellers (tellerid, tellername, balance, address,
                 branchid) "
221              + "VALUES(" + i + ", 'teller', 0,'adress', " + localRandom + ")");
222          if(isDebug) {
223            System.out.println("INSERT INTO tellers (tellerid, tellername, balance, address,
                   branchid) "
224                + "VALUES(" + i + ", 'teller', 0,'adress', " + localRandom + ")");
225          }
226        }
227
228        return true;
229      } catch (SQLException e) {
230
231        e.printStackTrace();
232      }
233
234      return false;
235    }
236
237
238    /**
239     * Bündelt die Funktionen des tpsCreators und führt diese nach ein ander aus.
240     *
241     * @param n Anzahl der Tupel, die erstellt werden
```

```
242     */
243   public void autoSetup(int n) {
244     System.out.println("Starte automatisches Setup...");
245     System.out.println("Erzeuge eine " + n + "-tps-Datenbank");
246 
247     if(!dropSchema()) {
248       System.out.println("Warnung: Datenbank konnte nicht geleert werden!");
249       return;
250     }
251     if(!createSchema()) {
252       System.out.println("Tabellen konnten nicht erzeugt werden. Stoppe Setup ...");
253       return;
254     }
255 
256     /**
257      * Jetzt kommen die Datensätze in die Datenbank yay \o/
258      */
259 
260     if(!createBranchTupel(n)) {
261       System.out.println("Branches konnten nicht angelegt werden. Stoppe Setup ...");
262       return;
263     }
264     if(!createAccountTupel(n)) {
265       System.out.println("Accounts konnten nicht angelegt werden. Stoppe Setup ...");
266       return;
267     }
268     if(!createTellerTupel(n)) {
269       System.out.println("Teller konnten nicht angelegt werden. Stoppe Setup ...");
270       return;
271     }
272 
273     System.out.println("Erstellen der " + n + "-tps-Datenbak erfolgreich");
274   }
275 
276   public boolean isDebug() {
277     return isDebug;
278   }
279 
280   public void setDebug(boolean isDebug) {
281     this.isDebug = isDebug;
282   }
283 
284   public void finishUp() {
285 
286   }
287   public void createFiles(int size) {
288 
289   }
290 }
```

Quellcodeauszug 19: TpsCreatorOldStatements (optimiert)