

Praktikumsaufgabe 7

Aufbau Benchmark-Datenbank mit Leistungsmessungen

Fach: Datenbanken und Informationssysteme

Semester: Wintersemester 15/16

Von: André Schlüß, Johannes Nowack, Timo Knufmann





Teilaufgabe b): Schätzung der Größe für n-tps Datenbank

Um eine annähernde Schätzung der Größe der Datenbank vornehmen zu können, wurden als Erstes die Größen der einzelnen Datentypen bestimmt, welche mittels der Dokumentation des Datenbanksystems ermittelt wurden.



Folgende Datentypen und –größen sind von Relevanz:

- Integer: 4 Byte
- Char-String: 2 Byte pro Character. In den verwendeten Strings wurde UTF-8 als Kodierung gewählt. UTF-8 kann 1 – 4 Bytes groß sein. Es wird davon ausgegangen, dass nur Buchstaben, Zahlen und einfache Sonderzeichen verwendet werden. Diese sind in der „Basic Multilingual Plane“ von UTF-8 vorhanden, welche 2 Byte groß sind.



Diese Größen wurden mit der entsprechenden Anzahl in den jeweiligen Tabellen verrechnet, sodass sich die Größe pro Datensatz in einer Tabelle ergab. Anschließend wurde die Anzahl der Datensätze für n in den jeweiligen Tabellen miteinbezogen. Daraus ergibt sich folgende Tabelle:

		Ein Datensatz:	Anzahl pro n		Größe für n:
Tabelle branches:		100	1		100
Tabelle accounts:		100	100000		10000000
Tabelle tellers:		100	10		1000
Tabelle history:		50	0		0
n=	1			Gesamt in Kilobytes:	9766,6992 2



Wie unten rechts zu lesen, liegt die geschätzte Größe bei ca. **9.800 Kilobytes** für n = 1.



Vergleich mit tatsächlicher Größe:

Die tatsächliche Größe der Datenbank beträgt ca. **14.440 Kilobytes**. Aus der Differenz ergibt sich somit, dass die tatsächliche Größe um ca. 4.600 Kilobytes **kleiner** ist.



Bei der Ermittlung der Größe wurde als Engine InnoDB verwendet. Nach Umstellung auf MyISAM betrug die Größe der Datenbank laut dem DBMS nur noch 11.880 Kilobyte.

Die Veränderung des Speicherplatzbedarfs lässt sich darauf zurückführen, dass die beiden Engines die Datenbank uneinheitlich verwalten und daher unterschiedlich viel zusätzlichen Speicher benötigen.

Implizit bedeutet das, dass im Allgemeinen zusätzlicher Speicher für die Verwaltung der Datenbanken benötigt wird. Aus diesem Grund weicht die tatsächliche Größe der Datenbank letztlich von der geschätzten ab.

Diese Verwaltung schließt beispielsweise die Benutzer, Berechtigungen, Verwaltung der Primär- und Sekundärschlüssel ein.

Teilaufgabe c): Laufzeitoptimierungen

Die erste Messung folgte auf der Entwicklermaschine, bei der ca. 1675 Sekunden für den Wert $n = 10$ benötigt wurde. Im Folgenden werden Optimierungen am Programmcode veranschaulicht, die die Laufzeiten beeinflussten.

Optimierung 1: Verwendung von „Prepared Statements“:

Die Idee bei der ersten Optimierung war, dass die Datenbank im Vorhinein bereits den SQL-Ausdruck kennt. Lediglich die Werte müssen dann noch übertragen werden.

Durch die Optimierung erreicht man eine geringere Netzlast und die Überprüfung der Syntax erfolgt ausschließlich einmal. Zuvor hingegen wurde eine Durchsicht für sämtliche SQL-Statements vom DBMS vorgenommen.

Die Durchlaufzeit für den Wert $n = 10$ beträgt auf dem gewerteten System zurzeit 1,18 Sekunden.

Optimierung 2: Deaktivierung von „Auto-Commit“ im Quelltext:


Der Grundgedanke der zweiten Optimierung beruhte auf der Sachlage, dass das DBMS Zwischenspeicher verwendet, auf denen Änderungen angewandt werden. Diese werden jedoch nicht in der eigentlichen Datenbank verzeichnet, sondern erst, wenn ein „Commit“-Befehl gegeben wird.

Diese Zwischenspeicher müssen für jede Transaktion neu angelegt und gelöscht werden, da beim Auto-Commit bei jeder Transaktion ein Commit ausgeführt wird. Vorteilhafter wäre ein manuelles Ausführen der Commits, sodass mehrere Statements den gleichen Zwischenspeicher nutzen.

Als Resultat würde eine gewisse Menge an Statements gleichzeitig in die Datenbank übernommen werden. Durch dieses Mittel soll eine beschleunigte Verarbeitung erfolgen. Die Zeit für $n = 10$ ergab auf dem gewerteten System **412,567 Sekunden**.


Optimierung 3: Änderungen an Datenbankeinstellungen

Das Leitmotiv der dritten Optimierung basierte auf die Beschleunigung der Laufzeit durch Veränderungen von Parametern der Datenbank. In Erwägung wurden verschiedene Puffergrößen, Dateigrößen oder auch die Deaktivierung des Transaktionslogs gezogen.

Als Ergebnis lässt sich darstellen, dass angesichts größere Anzahl an zur Verfügung stehenden Ressourcen, Vorgänge schneller ausgeführt und bearbeitet werden können. In Folge dessen betrug die Zeit für den Wert $n = 10$ auf dem gewerteten System **209,089 Sekunden**. 

Optimierung 4: Umstellung auf MyISAM

Nach Recherche und Tests stellte sich heraus, dass MyISAM eine geeignetere Engine für den aktuellen Anwendungsfall darstellt. Dies liegt daran, dass das Einfügen von Datensätzen unter MyISAM schneller ist als unter InnoDB. InnoDB hingegen ist effizienter bei Query-Statements, die im aktuellen Anwendungsfall jedoch nicht verwendet werden.

Des Weiteren wurde „Batching“  eingeführt, sodass die vielen vorhandenen INSERT-Statements mittels Batch-Verarbeitung ausgeführt werden. Der dadurch erhoffte Geschwindigkeitsprofit wurde jedoch später revidiert (siehe Optimierung 5), da die Anzahl der Statements verringert wurde. Der Nutzen des Batchings ging dabei verloren.

Die Benchmarktests für die aktuelle Optimierung ergaben, dass die Verarbeitung sich für $n = 10$ auf **140 Sekunden** erstreckte, die für $n = 20$ auf **278 Sekunden**. Die Messungen wurden auf einem Entwicklersystem und nicht auf dem gewerteten System durchgeführt.

Optimierung 5: Verwendung von „Multiple Row Insertion“

Der Aufbau eines SQL-Insert-Statement kennzeichnet sich durch zwei wesentliche Angaben aus. Zu Beginn wird die Tabelle genannt und der Aufbau der einzufügenden Daten dargestellt. Auf das SQL Schlagwort „VALUES“ folgen die eigentlichen Daten eines Datensatzes.

Bisher wurde ein Insert-Statement pro Datensatz abgesetzt.

Ziel dieser Optimierung ist die Übermittlung mehrerer Datensätze mithilfe eines einzigen Insert-Statements.

Nachdem eine Verbesserung der Geschwindigkeit festgestellt werden konnte, wurden die vorherigen Optimierungen nochmals überprüft. Dabei stellte sich heraus, dass das Batching negativen Einfluss auf die Laufzeit der Verarbeitung hatte. Ein mutmaßlicher Grund ist die

starke Dezimierung der Statements durch Verwendung von „Multiple Row Insertions“. Der Vorteil des Batchings liegt bei einem Geschwindigkeitsprofit, wenn viele Statements abgesetzt werden müssen, der durch diese Dezimierung verloren ging.

Auf einem Entwicklerlaptop konnten folgende Geschwindigkeiten ermittelt werden:

- Für $n = 10$: **8,12 Sekunden**
- Für $n = 20$: **18,202 Sekunden**
- Für $n = 50$: **42,632 Sekunden**

Teilaufgabe d): Endgültige Messung

Die nachfolgenden Tests wurden auf dem gewerteten System ausgeführt.

Die Client-Tests wurden von einem Entwicklersystem ausgeführt, das mit der Datenbank auf dem gewerteten System verbunden war.

	Dauer auf Serversystem in Sekunden	Dauer auf Clientsystem in Sekunden
n = 10	20,942	20,46
n = 20	41,366	41,585
n = 50	105,999	106,77



Kommentierter Programmcode

Klasse DbConnectionInfo.java

```
package com.whs.dbi21.benchmark;

/**
 * Beinhaltet Logindaten für das DBMS.
 *
 * @author André Schlüß, Johannes Nowack, Timo Knufmann
 */
public class DbConnectionInfo {
    public final static String JDBCSTRING = "jdbc:mysql://Datenbank-PC/benchmarktest";
    public final static String DBUSER = "dbuser";
    public final static String DBPASSWORD = "daten2";
}
```

Klasse Main.java

```
package com.whs.dbi21.benchmark;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Date;
import java.util.Scanner;

import com.whs.dbi21.benchmark.DbConnectionInfo;

/**
 * Benchmark-Test für MySQL
 *
 * In diesem Benchmark-Test werden abhängig von n viele INSERT-Statements zur Datenbank geschickt. Beim Aufruf der main-Methode hat der User die Möglichkeit einen Wert für n einzutragen. üblicherweise werden Werte von 1-50 für n verwendet, um die Datenbank zu testen.
 *
 * Sobald der Test durchgelaufen ist, wird die Dauer in Sekunden zurückgegeben.
 *
 * Des Weiteren werden die Tabellen zur Vorbereitung des Tests beim Aufruf der main-Methode geleert.
 *
 * @author Johannes Nowack, André Schlüß, Timo Knufmann
 */
public class Main {

    private static Connection dbCon;

    /**
     * Stellt die Verbindung zu einem DBMS her. Wenn eine Verbindung erfolgreich hergestellt werden konnte, wird true zurückgegeben.
     *
     * @return Gibt false zurück, falls es einen Fehler beim Herstellen der
     */
}
```

Verbindung zu einem DBMS gab

```
*/
private static boolean initializeConnection() {
    try {
        dbCon = DriverManager.getConnection(DbConnectionInfo.JDBCSTRING,
        DbConnectionInfo.DBUSER, DbConnectionInfo.DBPASSWORD);
        return true;
    } catch (SQLException e) {
        e.printStackTrace();
        return false;
    }
}

/**
 * Start des Programms. Zuerst wird versucht eine Verbindung zu einem
 * DBMS herzustellen. Anschließend
 * werden die Tabellen geleert. Daraufhin wird der eigentliche
 * Benchmark-Test ausgeführt und am Ende
 * wird die benötigte Dauer in Sekunden zurückgegeben.
 *
 * @param args
 */
public static void main(String[] args) {
    // Herstellen der Verbindung zur Datenbank
    if (!initializeConnection()) {
        System.out.println("Could not establish connection to database!");
        System.exit(-1);
    }
    System.out.println("Connection to database established!");

    // Aufräumen der Tabellen
    cleanDatabase();
    System.out.println("Database cleaned!");

    // Einlesen von n. Falls Wert nicht gültig, wird der default 10
    verwendet
    System.out.print("Please enter a number for n: ");
    int n = readNumberInInt();
    if (n < 1) {
        System.out.println("No valid number for n was entered, using
default: 10");
        n = 10;
    }

    // Starten des Benchmark-Tests und festhalten der Zeitpunkte
    System.out.println("Start benchmark test!");
    Date testStart = new Date();
    Statements.createdb(n, dbCon);
    Date testFinish = new Date();

    // Ausgabe der benötigten Zeit in Sekunden
    long timeUsed = testFinish.getTime() - testStart.getTime();
    System.out.println("Time used in seconds = " + (double)timeUsed /
1000);
}

/**
 * Bereinigt die Benchmark-Test Datenbank.
 *
 * @return Gibt true zurück, wenn das Bereinigen erfolgreich ausgeführt
 werden konnte.
 */
```



```

private static boolean cleanDatabase() {
    Statement st;
    try {
        boolean autoCommit = dbCon.getAutoCommit();

        dbCon.setAutoCommit(false);
        st = dbCon.createStatement();
        st.addBatch("DELETE FROM history;");
        st.addBatch("DELETE FROM accounts;");
        st.addBatch("DELETE FROM tellers;");
        st.addBatch("DELETE FROM branches;");
        st.executeBatch();
        dbCon.commit();
        st.close();

        dbCon.setAutoCommit(autoCommit);
        return true;
    } catch (SQLException e) {
        e.printStackTrace();
        return false;
    }
}

/**
 * Liest eine Integervariablen aus dem STDIN ein und gibt diese zurück.
 * Falls es sich um keine gültige Zahl handelt,
 * wird -1 zurückgegeben
 *
 * @return Eingelesene Integerzahl
 */
private static int readNumberInInt() {
    Scanner scanner = new Scanner(System.in);
    String s = scanner.next();
    try {
        int i = Integer.parseInt(s);
        scanner.close();
        return i;
    } catch (NumberFormatException e) {
        scanner.close();
        return -1;
    }
}
}

```


Klasse Statements.java

```
package com.whs.dbi21.benchmark;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Date;
import java.util.Random;

/**
 * In dieser Klasse wird der eigentliche Benchmark-Test durchgeführt. Dafür
 * wird eine bereits bestehende Verbindung
 * zu einer Datenbank benötigt.
 *
 * @author André Schlöß, Johannes Nowack, Timo Knufmann
 */
public class Statements {

    /**
     * Führt den eigentlichen Benchmark-Test durch. Es werden n Datensätze
     * in der "branches" Tabelle angelegt.
     * Des Weiteren werden n * 100000 Datensätze in der "accounts" Tabelle
     * angelegt und n * 10 Datensätze in der
     * "tellers" Datenbank.
     *
     * @param pn Faktor für die Anzahl der anzulegenden Daten
     * @param pconnection Datenbankverbindung zur Datenbank, auf der der
     * Test durchgeführt werden soll
     */
    static void createdb(int pn, Connection pconnection){

        PreparedStatement stmt=null;
        Statement stmt2;

        // Erzeugung zufälliger Strings, die im Benchmark als "Dummies"
        // verwendet werden
        String name = createString(20);
        String addresslong = createString(72);
        String addressshort = createString(68);

        // Statements als Strings werden vorgeneriert, dabei wird das
        // Einfügen von bis zu 50000 zeilen in einem Statement verwendet
        String insertB = prepareInsertString("INSERT INTO branches (branchid,
        balance, branchname, address) VALUES",4,pn);
        String insertA = prepareInsertString("INSERT INTO accounts (accid,
        balance, branchid, name, address) VALUES",5,50000);
        String insertT = prepareInsertString("INSERT INTO tellers (tellerid,
        balance, branchid, tellername, address) VALUES",5,pn*10);

        try {
            // Deaktivierung AutoCommit (Optimierung)
            pconnection.setAutoCommit(false);

            stmt2 = pconnection.createStatement();
            stmt2.close();
            pconnection.commit();

            // Verwendung von Prepared Statements (ebenfalls Optimierung)
```

```

        stmt = pconnection.prepareStatement(insertB);

        for (int i=0;i<pn;i++){
            // Einfügen von Datensätzen in die Tabelle "branches" mithilfe
            // der Prepared Statements
            stmt.setInt(i*4+1, i+1);
            stmt.setInt(i*4+2, 0);
            stmt.setString(i*4+3, name);
            stmt.setString(i*4+4, addresslong);
        }

        stmt.executeUpdate();
        stmt.close();

        // INSERT-Statements für die Tabelle "accounts" vorbereiten und
        // ausführen
        stmt = pconnection.prepareStatement(insertA);

        for (int j=0;j<pn*2;j++){
            for (int i=0;i<50000;i++){
                stmt.setInt(i*5+1, i+1+j*50000);
                stmt.setInt(i*5+2, 0);
                stmt.setInt(i*5+3, ((int) (Math.random()*pn)+1));
                stmt.setString(i*5+4, name);
                stmt.setString(i*5+5, addressshort);
            }
            stmt.executeUpdate();
        }
        stmt.close();

        // INSERT-Statements für die "tellers" Tabelle vorbereiten und
        // ausführen
        stmt = pconnection.prepareStatement(insertT);

        for (int i=0;i<pn*10;i++){
            stmt.setInt(i*5+1, i+1);
            stmt.setInt(i*5+2, 0);
            stmt.setInt(i*5+3, ((int) (Math.random()*pn)+1));
            stmt.setString(i*5+4, name);
            stmt.setString(i*5+5, addressshort);
        }
        stmt.executeUpdate();
        stmt.close();

        // Commit in der Datenbank aufrufen
        pconnection.commit();

        stmt2 = pconnection.createStatement();
        stmt2.close();
        pconnection.commit();

    } catch (SQLException e1) {
        e1.printStackTrace();
    }
}

/**
 * Erzeugt einen zufälligen String der Länge azchar
 *
 * @param azchar Länge des zufälligen Strings
 * @return Zufälliger String

```

```

*/
static String createString(int azchar){
    String rString="";
    Random rnd=new Random();

    String alphabet="abcdef";

    for (int i=0;i<azchar;i++){
        rString=rString+alphabet.charAt(rnd.nextInt(alphabet.length()));
    }

    return rString;
}

/**
 * Generiert einen String, der ein Multi-Row-Insert-Statement enthält
 *
 * @param head Kopf des Insert-Statements bis "Values"
 * @param azparam Anzahl der Parameter (Fragezeichen im String) einer
Zeile
 * @param azwdh Anzahl der Zeilen, die das Statement einfügen soll
 * @return Fertiges (Prepared-)Statement als String
 */
static String prepareInsertString(String head, int azparam, int azwdh){
    StringBuilder prepInsert=new StringBuilder();

    prepInsert.append(head);

    for (int i1=0;i1<azwdh;i1++){
        prepInsert.append("(");
        for (int i2=0;i2<azparam;i2++){
            prepInsert.append("?");
            if (i2<azparam-1){
                prepInsert.append(",");
            }
        }
        prepInsert.append(")");
        if (i1<azwdh-1){
            prepInsert.append(",");
        }
    }

    prepInsert.append(";");
    return prepInsert.toString();
}
}

```

