Architektur von Datenbanksystemen Praktikumsbericht 2

Benedikt Kappes
Tuan-Si Tran

3. November 2011

1 Einleitung

Im letzten Praktikum haben wir uns auf einer sehr tiefen Ebene mit der DB2-Datenbank beschäftigt, indem wir die Byte-Größe der Indexe und der Daten betrachtet haben. In diesem Praktikum werden die Daten aus dem vorherigen Praktikum als Testdaten verwendetet, um verschiedene Anwenderprofile zu simulieren und deren Auswirkung auf die Performance zu beurteilen.

Dafür sollen folgende vier Szenarios durchgespielt werden:

Szenario 1.1 Viermaliges sequenzielles lesen aller Bestellungsdatensätze in 300er-Portionen mit der Klasse `Statement`

Szenario 1.2 Viermaliges sequenzielles lesen aller Bestellungsdatensätze in 300er-Portionen mit der Klasse `PreparedStatement`

Szenario 2 Lesen von 3.000.000 Bestellungsdatensätze in 300er-Portionen mit zufälligem Startpunkt.

Szenario 3 Lesen von 3.000.000 Bestellungsdatensätze, wobei immer die selben 300 Datensätze gelesen werden.

Szenario 4 Wie Szenario 1, nur diesmal in 10er-Portionen

Szenario 1.1 und 1.2 dienen dabei als Referenzgröße. Es soll bei allen Szenarios sichergestellt werden, dass alle bzw. die richtigen Datensätze gelesen werden.

2 Vorbereitung

In den Szenarien lesen wir über JDBC mit `SELECT`-Statements 300 Bestellungsdatensätze (bzw. 10 in Szenario 4) aus. Um auszuschließen, dass die Treiber nur eine Untermenge der Ergebnismenge zurückgeben (falls nicht alle Datensätze benötigt werden), müssen wir zu jeden gewünschten Datensatz mindestens ein Attribut einmal lesend anfassen.

In unserem Fall wurde die Funktion `checkResult(resultSet, start, end)` angelegt, die immer die `BID` jedes Tupels ausliest. Als erstes Argument bekommt diese dabei das `resultSet` der `SELECT`-Anfrage übergeben. Als zweites und drittes Argument erhält die Funktion die erste und die letzte angefragte `BID` des 300er-Tupels. Die Funktion durchläuft dann in einer Schleife für

i=start bis i=end das `resultSet` und vergleicht, ob das Attribut `BID` der Variable i entspricht. Damit überprüfen wir gleichzeitig auf die richtige Anzahl als auch auf die richtigen Datensätze. Der Code dieser Funktion befindet sich im Anhang.

3 Durchführung

Als erstes wurde die automatische Leistungsoptimierung der DB2-Datenbank abgeschaltet, um reproduzierbare Ergebnisse zu erhalten. Danach wurde Szenario 1.1 implementiert. Dabei wurde nur mit der Klasse `Statement` gearbeitet. Erst in Szenario 1.2 wurden `PreparedStatements` verwendet. Das Lesen von 300er Portionen wurde anhand einer Zählvariable realisiert, die in jedem Durchlauf um 300 erhöht und in der `WHERE`-Klausel angegeben wird. Dabei machen wir uns zu Nutze, dass wir beim Eintragen der Bestellungsdaten für die ID eine fortlaufende Sequenz verwenden. Das resultierende SQL-Statement sieht dabei folgendermaßen aus:

```
SELECT * FROM Bestellung WHERE bid <= ? AND bid => ?
```

Für die nachfolgenden Szenarien wurden dann nur noch der korrekte Wertebereich für die beiden Platzhalter eingetragen.

Für Szenario 2 wurde dabei eine Zufallszahl zwischen 0 und 2500 erzeugt und mit 300 multipliziert um den Offset jedes 300er-Päckchen bestimmen zu können.

Für das dritte Szenario wurden immer nur die ersten 300 Bestellungsdatensätze gelesen, d.h. für die beiden Platzhalter wurden immer die 1 und 300 eingetragen.

Für das vierte Szenario wurden die Datensätze in 10er-Tupel gelesen. Dabei wurden die verschachtelten Schleifen angepasst.

4 Ergebnisse und Auswertung

Folgende Werte wurden bei der Durchführung der Szenarien gemessen (Tabelle 4.1):

	SSD in ms	mech. in ms
Szenario 1.1	54245	49389
Szenario 1.2	27466	20619
Szenario 2	38294	24468
Szenario 3	7156	10869

Szenario 4	126695	121315

Tabelle 4.1: Messwerte für Szenarien

Der Vergleich zwischen Szenario 1.1 und 1.2 zeigt keine Überraschung. Die Implementierung des sequenziellen Lesens mit `PreparedStatements` ist um etwa 50% schneller als mit `Statements`. Das ist darauf zurückzuführen, dass `PreparedStatements` bereits vorkompiliert auf der Datenbank liegen. Diese müssen daher nicht mehr geparsed werden und können sofort ausgeführt werden.

Szenario zwei ist um fast 30% langsamer als Szenario 1.2. Der Grund dafür ist, dass beim sequenziellen Lesen die angeforderten Seiten öfter schon im Puffer vorzufinden sind als bei dem zufalligen Auswählen des Starttupels. Bei genauerer Betrachtung lässt sich folgender Sachverhalt feststellen:

Größe eines Tupels aus `Bestellung`	18 Byte (vgl. Praktikumsbericht 1)
Größe von 300 Tupeln	18 Byte * 300 = 5400 Byte
Größe einer Pufferseite	4096 Byte [1]

Das Lesen von 300 Tupeln erfordert das Laden von (min.) zwei Seiten aus dem Speicher. Auf eine Seite passen also theoretisch 227 Bestellungstupel (4096 / 18=227; Overhead für TID wurde vernachlässigt). Daraus folgt, dass auf der zweiten Seite 73 Tupel aus der ersten Abfrage vorhanden sind. Die restlichen 154 Tupel sind somit schon vom nächsten 300er-Paket (siehe Abbild 4.1):

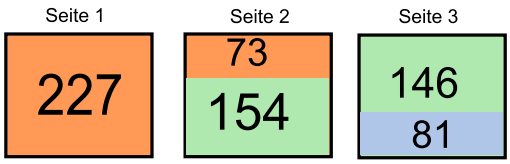

Abbildung 4.1: Verteilung der Tupel auf die Seiten

Nach dieser Überlegung muss die Datenbank beim sequenziellen Lesen für das zweite 300er-Pakete also nur eine weitere Seite in den Puffer laden. Dadurch lädt das sequenzielle Lesen immer meistens nur eine oder zwei Seiten nach. Bei Szenario zwei kann dieser Sachverhalt nicht gezielt ausgenutzt werden. In vielen Fällen wird die Zufallszahl auf Tupel zeigen, die nicht direkt auf einer der bereits vorhanden Seiten liegt. Dadurch muss die Datenbank häufiger eine Seite mehr von der Festplatte laden als Szenario 1.2.

Beim dritten Szenario betrug die Ausführungsdauer gerade einmal 7156ms. Da wir in diesem Szenario immer die identischen 300 Datensätze lesen, befinden diese sich nach dem ersten Lesen bereits vollständig im Puffer. Das DBMS muss also danach weder von der Festplatte lesen noch sich um das Auslagern

[1] `SELECT BPNAME FROM SYSCAT.BUFFERPOOLS`; Default Page Size ist ebenso 4KB

der Pufferseiten Gedanken machen. Wie man sehen kann, ist der Unterschied bereits beim Lesen von 3 Millionen Datensätzen signifikant im Vergleich zu den ersten beiden Szenarien.

Interessant sind die Messwerte des vierten Szenarios. Dieses Szenario hat mehr als doppelt so lange wie Szenario **1.1** gebraucht, trotz Verwendung von `PreparedStatements`. Auch der Puffer ist diesmal nicht der Grund für die große Dauer, da die Seite sich nach der ersten Anfrage eines 10er-Paket bereits im Puffer befinden sollte, auf die bei der nächsten Iteration zugegriffen werden kann. Alleine die Kommunikation zwischen der Anwendung über den Treiber zur Datenbank kostet enorm viel Zeit. Schließlich werden 30 Mal mehr Anfragen an die Datenbank geschickt als in Szenario 1.

Schlussfolgerungen

Anhand der Auswertung kommen wir zu folgenden wichtigen Schlussfolgerungen:

- Wenn eine bestimmte SQL-Abfrage sehr häufig benötigt wird und sich Teile des Querys parametrisieren lassen, dann sollte man `preparedStatements` verwenden.
- Die Zeitdifferenz beim Lesen vom Puffer und beim Lesen von der Festplatte ist bereits bei etwa 52 MB [2] sehr groß (vgl. Szenario 3 und Szenario 1.2).
- Die Kommunikation zwischen Applikation und Datenbank kostet viel Zeit. Braucht man in der Applikation eine Fragmentierung großer Datenmengen aus der Datenbank, so sollte diese Fragmentierung eher in der Applikation stattfinden, als über die Zerlegung in mehrere SQL-Queries.

5 Zusammenfassung

Das Ziel dieses Praktikums war die Auswirkungen auf die Performance durch verschiedene Anwenderprofile kennenzulernen. Dies sollte über vier simulierte Szenarien vermittelt werden. Dabei haben wir beobachten können, dass die größten Zeitdifferenzen in den Bereichen `Statement`/`PreparedStatement`, Zugriff auf Puffer/Festplatte und Häufigkeit der Anfrage von der Applikation zur Datenbank entstehen.

[2] 13,5 MB* 4, vgl. Datengröße der Tabelle `Bestellung` aus P1

6 Anhang

```
public boolean checkResult(ResultSet result, int start, int end){
    boolean ret = true;
    int bid;
    try {
        while(result.next()){
           bid = result.getInt(1);
           if((bid != start)||(bid > end)){
               ret = false;
               break;
           }
           start++;
        }
    } catch (SQLException ex) {
        [...]
    }
    return ret;
}
```

Literaturverzeichnis