3. Übungsblatt

Ausgabe: 14. Mai 2015 · Besprechung: 21./28. Mai 2015

1 Operator-At-A-Time, Tuple-At-A-Time, Vector-At-A-Time (Besprechung: 21.05.2015)

In den Quelldateien für das vorige Aufgabenblatt finden Sie auch eine Codevorlage für einen einfachen *Volcano*-basierten Anfrageprozessor. Konkret ist in `engine.h` eine Menge von Operatoren deklariert. Aus diesen Operatoren wird in `cs_iterator.c` ein Anfrageplan zusammengesetzt, der der Anfrage

```
SELECT  MAX(orderkey)
  FROM  lineitem
```

entspricht.

Die Vorlage sieht dabei bereits ein "Vector-At-A-Time"-Verarbeitungsmodell vor (mit dem sich wiederum durch Wahl der Vektorgröße die anderen Modelle simulieren lassen, siehe Vorlesung).

Aufgaben:

1. Ergänzen Sie die fehlenden Operatorimplementationen (insbesondere die Funktionen `next_*`. Am besten legen Sie dazu eine neue Datei `engine.c` an, die die Implementationen enthält. Wahrscheinlich müssen Sie außerdem einige der `structs` in `engine.h` ergänzen.

2. Auf Vorlesungsfolie 60 wurde der Einfluss der Vektorgröße auf die Ausführungszeit am Beispiel des MonetDB/X100-Systems gezeigt. Erstellen Sie ein entsprechendes Diagramm auch für Ihre Implementation eines "Vector-At-A-Time"-Prozessors.

2 Hashjoins (Besprechung: 28.05.2015)

In dieser Aufgabe wollen wir die cache-effiziente Implementation eines Hashjoin (gemäß der Vorlesung) nachvollziehen. Wir machen dabei Annahmen, wie sie typisch sind für ein hauptspeicherbasiertes System:

- Wir verwenden nur ein **zweispaltiges Tabellenschema**, wobei `key` das Joinattribut ist, `value` zusätzlicher Dateninhalt (der aber vom Joinalgorithmus selbst gar nicht interpretiert wird).
- Wir stellen die Tabelle in C als einfaches **Array** dar, z. B.

```
struct tuple_t {
    unsigned long int   key;
    unsigned long int   value[N];
}
```

  (wobei mittels `N` die Größe des Payloads variiert werden kann; Sie können gerne auch mit 32-Bit-Schlüsseln experimentieren.)

Die `key`-Felder füllen wir mit gleichverteilten Zahlen. Am besten eignet sich dabei (für eine der Tabellen) eine Permutation der Zahlen $[1..\mathit{AnzahlTupel}]$. In der anderen Relation können dann auch Zufallszahlen aus diesem Bereich verwendet werden. Bei gleichverteilten Zahlen können Sie dann als Hashfunktion auch einfach die Identität verwenden.

Teilaufgabe 1

Implementieren Sie einen "gewöhnlichen" Hashjoin (Folie 73).

Untersuchen Sie experimentell, wie sich die Kosten pro Tupel in Abhängigkeit vom Datenvolumen verhält. Lässt sich hier ein Zusammenhang zu den Cache-Parametern Ihres Systems erkennen?

Teilaufgabe 2

Die Cache-Effizienz eines Hashjoins kann gesteigert werden durch Partitionierung (wie in der Vorlesung besprochen).

Implementieren Sie die Partitionierungs-Operation.

Dabei stehen Ihnen im Prinzip drei Optionen zur Auswahl:

1. Einfache ("naïve") Partitionierung
2. Partitionierung mittels "software-managed buffers"
3. Mehrstufige Partitionierung.

Welche der Optionen erreicht (in Abhängigkeit vom Datenvolumen) den besten Durchsatz?

Teilaufgabe 3

Bauen Sie schließlich Partitionierung und Hashjoin zusammen ($\rightarrow$ Radix Join).