3. Übungsblatt

Ausgabe: 13. Mai 2014 · Besprechung: 22. Mai 2014

Einleitung

Bei diesem Aufgabenblatt soll es darum gehen, cache-optimierte Implementationen von Hashjoins im Hauptspeicher nachzuvollziehen. 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 $[1..AnzahlTupel]$. 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.

Aufgabe 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?

Aufgabe 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?

Aufgabe 3

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