# 4. Übungsblatt

Ausgabe: 19. Juni 2017 · Besprechung: 26. Juni / 3. Juli 2017

## 1 Hash Joins (Besprechung am 26.06.2017)

In der Vorlesung hatten wir vor einiger Zeit die cache-effiziente Implementierung von Hash Joins besprochen. In diesem Aufgabenblatt wollen wir diese Ideen nach-implementieren. 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).**

## 2 Parallele Hash Joins (Besprechung am 03.07.2017)

In modernen Systemen ist es natürlich naheliegend, Hash Joins auch parallel auszuwerten.

**Wie kann man Hash Joins (*radix joins*) effizient parallelisieren?**

Überlegen Sie sich zunächst eine geeignete Strategie zur Parallelisierung von Hash Joins. Integrieren Sie diese Strategie dann in Ihre Joinimplementation.

### 2.1 Paralleles Partitionieren

Besonders kritisch ist bei der Parallelisierung die Partitionierungs-Phase des Algorithmus. In der Vorlesung wurde am Rande skizziert, wie sich der einfach Hash Join parallelisieren lässt. Gelten die gleichen Voraussetzungen auch für *radix join*? Welches Verhalten Ihres Algorithmus erwarten Sie, wenn die Daten nicht gleichverteilt sind?

### 2.2 Parallele Joinverarbeitung

Nach der Partitionierung ergibt sich ganz natürlich die Möglichkeit zur parallelen Verarbeitung, weil die einzelnen Teil-Joins $r_i \bowtie s_i$ voneinander unabhängig berechnet werden können.

Wie könnten diese Teil-Joins möglichst geschickt auf vorhandene CPU-Ressourcen verteilt werden? Welches Verhalten erwarten Sie bei nicht-gleichverteilten Daten?

## 2.3 NUMA-Architekturen

In der Vorlesung wurden zwischenzeitlich NUMA-Architekturen diskutiert. Welches Verhalten erwarten Sie bei Ihrer Implementierung in Bezug auf NUMA-Effizienz? Überlegen Sie sich ob/wie sich Hash Joins besonders effizient in NUMA-Systemen verarbeiten lassen könnten.