Data Processing on Modern Hardware
Assignment 3 – Hardware optimized hash joins

Handout: **20th May 2020**
Due: **3rd June 2020 by 9am**

Introduction

The goal of this exercise is to apply some of the optimization techniques we covered in the last two weeks in the context of hash join, and analyze the cache characteristics of the original and optimized version of the code that joins two relations R and S. The assumptions we make are the following:

- Each relation contains a set of tuples, identified by an ID (join attribute key) and a payload.
- For the purpose of this exercise, the join only counts the matching results.
- We can implement a relation in C++ using a simple array, e.g.,

```
struct tuple_t {
  uint64_t key;
  uint64_t value[N];
}
```

We populate both relations with tuples with uniform distribution. For one of the relations as key you can use a permutation of the numbers [1 :: number of tuples], while for the second relation for the join key, you can generate random numbers from the same range.

Part 1 – Hash Join baseline

Implement a simple hash join (e.g., slide 4 from the lecture on In-memory Joins) and analyze the performance characteristics (per tuple) as you vary the dataset size (e.g., the size of the input relations).

Part 2 – Partitioning

The cache efficiency of the hash join can be improved by partitioning the input relations to cache-size chunks. Implement the partitioning operation with three options:

1. Simple naïve partitioning,
2. Partitioning with "software-managed buffers" and "non-temporal writes",
3. Multi-pass partitioning.

Analyze which option gives you the best performance, depending on the input relation size.

Part 3 – Radix Join

Implement the Radix Join algorithm by joining the partitioning phase with the hash join baseline. Analyze the performance characteristics (per tuple) of your implementation of the radix join and compare it to the baseline results from Part 1.

Submission guidelines

This homework has a duration of two weeks. Fork the repository, commit your changes in the git, and invite us (@*dpmh*) to hand in your homework. We advise you to start by implementing *Part 1*, *Part 2.1*, *Part 2.3* and *Part 3* at first and then implement *Part 2.2* afterward.

The programming language of this homework is C++. We provide you a simple code skeleton, feel free to add functions or change the function signatures if needed.

The lecture covers all parts of this homework and provides pseudo code for the hash join and the partitionings. Use the slides and the pseudo-code as a hint for the tasks.
Furthermore, you are allowed to use libraries, e.g., for the hash table. However, please make sure you understand how the data structures work to interpret your performance results correctly. If you want to implement the hash table yourself, do not spend too much time on it. A simple hash table is sufficient, and the lecture slides cover some ideas for hash table design.