# A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing

Stefan Richter
Information Systems Group
Saarland University
stefan.richter@infosys.uni-saarland.de

Victor Alvarez*
Dept. Computer Science
TU Braunschweig
alvarez@ibr.cs.tu-bs.de

Jens Dittrich
Information Systems Group
Saarland University
dittrich@cs.uni-saarland.de

## ABSTRACT

Hashing is a solved problem. It allows us to get constant time access for lookups. Hashing is also simple. It is safe to use an arbitrary method as a black box and expect good performance, and optimizations to hashing can only improve it by a negligible delta. Why are all of the previous statements plain wrong? That is what this paper is about. In this paper we thoroughly study hashing for integer keys and carefully analyze the most common hashing methods in a five-dimensional requirements space: (1) data-distribution, (2) load factor, (3) dataset size, (4) read/write-ratio, and (5) un/successful-ratio. Each point in that design space may potentially suggest a different hashing scheme, and additionally also a different hash function. We show that a right or wrong decision in picking the right hashing scheme and hash function combination may lead to significant difference in performance. To substantiate this claim, we carefully analyze two additional dimensions: (6) five representative hashing schemes (which includes an improved variant of Robin Hood hashing), (7) four important classes of hash functions widely used today. That is, we consider 20 different combinations in total. Finally, we also provide a glimpse about the effect of table memory layout and the use of SIMD instructions. Our study clearly indicates that picking the right combination may have considerable impact on insert and lookup performance, as well as memory footprint. A major conclusion of our work is that hashing should be considered a white box before blindly using it in applications, such as query processing. Finally, we also provide a strong guideline about when to use which hashing method.

## 1. INTRODUCTION

In recent years there has been a considerable amount of research on tree-structured main-memory indexes, e.g. [17, 13, 21]. However, it is hard to find recent database literature thoroughly examining the effects of different hash tables in query processing. This is unfortunate for at least two reasons: First, hashing has plenty of applications in modern database systems, including join processing, grouping, and accelerating point queries. In those applications, hash tables serve as a building block. Second, there is strong evidence that hash tables are much faster than even the most recent and best tree-structured indexes. For instance, in our recent experimental analysis [1] we carefully compared the performance of modern tree-structured indexes for main-memory databases like ARTful [17] with a selection of different hash tables[1]. A central lesson learned from our work [1] was that a carefully and well-chosen hash table is still *considerably* faster (up to factor 4-5x) for point queries than any of the aforementioned tree-structured indexes. However, our previous work also triggered some nagging research questions: (1) When exactly should we choose which hash table? (2) What are the most efficient hashing methods that should be considered for query processing? (3) What other dimensions affect the choice of "the right" hash table? and finally (4) What is the performance impact of those factors. While investigating answers to these questions we stumbled over interesting results that greatly enriched our knowledge, and that could greatly help practitioners, and potentially also the optimizer, to take well-informed decisions as of when to use what hash table.

### 1.1 Our Contributions

We carefully study *single-threaded* hashing for 64-bit integer keys and values in a five-dimensional requirements space:

1. **Data distribution.** Three different data distributions: dense, sparse, and a grid-like distribution (think of IP addresses).
2. **Load factor.** Six different load factors between 25- and 90%.
3. **Dataset size.** We consider a variety of sizes for the hash tables to observe performance when they are rather small (they fit in cache), and when they are of medium and large sizes (outside cache but still addressable by TLB using huge pages or not respectively).
4. **Read/write-ratio.** We consider whether the hash tables are to be used under a static workload (OLAP-like) or a dynamic workload (OLTP-like). For both we simulate an indexing workload — which in turn captures the essence of other important operations such as joins or aggregates.
5. **Un/successful lookup ratio.** We study the performance of the hash tables when the amount of lookups (probes) varies from *all successful* to *all unsuccessful*.

Each point in that design space may potentially suggest a different hash table. We show that a right/wrong decision in picking the

*Work done while at Saarland University.


*Proceedings of the VLDB Endowment,* Vol. 9, No. 3


[1]We use the term *hash table* throughout the paper to indicate that *both* the hashing scheme (say linear probing) and the hash function (say Murmur) are chosen.

right combination ⟨hashing scheme, hash function⟩ may lead to an order of magnitude difference in performance. To substantiate this claim, we carefully analyze two additional dimensions:

6. **Hashing scheme.** We consider linear probing, quadratic probing, Robin Hood hashing as described in [5] but carefully engineered, Cuckoo hashing [19], and two different variants of chained hashing.

7. **Hash function.** We integrate each hashing scheme with four different hash functions: Multiply-shift [8], Multiply-add-shift [7], Tabulation hashing [20], and Murmur hashing [2], which is widely used in practice. This gives 24 different combinations (hash tables).

Therefore, **we study in total a set of seven different dimensions** that are key parameters to the overall performance of a hash table. We shed light on these seven dimensions focusing on one of the most important use-cases in query processing: indexing. This in turn resembles very closely other important operations such as joins and aggregates — like `SUM`, `MIN`, etc. Additionally, we also offer a glimpse about the effect of different table layout and the use of SIMD instructions. Our main goal is to produce enough results that can guide practitioners, and potentially the optimizer, towards choosing the most appropriate hash table for their use case at hand. **To the best of our knowledge, no work in the literature has considered such a thorough set of experiments on hash tables**.

Our study clearly indicates that picking the right configuration may have considerable impact on standard query processing tasks such as main-memory indexing as well as join processing, which heavily rely on hashing. Hence, hashing should be considered as a white box method in query processing and query optimization.

We decided to focus on studying hash tables in a single-threaded context to isolate the impact of the aforementioned dimensions. We believe that a thorough evaluation of concurrency in hash tables is a research topic in its own and *beyond the scope of this paper*. However, our observations still play an important role for hash maps in multi-threaded algorithms. For partitioning-based parallelism — which has recently been considered in the context of (partition-based hash) joins [3, 4, 16] — single-threaded performance is still a key parameter: each partition can be considered an isolated unit of work that is only accessed by exactly one thread at a time, and therefore concurrency control inside the hash tables is not needed. Furthermore, all hash tables we present in the paper can be extended for thread safety through well-known techniques such as striped locking or compare-and-swap. Here, the dimensions we discuss still impact the performance of the underlying hash table.

This paper is organized as follows: In Sections 2 and 3 we briefly describe each of the five considered hashing schemes and the four considered hash functions respectively. In Section 4 we describe our methodology, setup, measurements, and the three data distributions used. We also discuss why we have narrowed down our result set — we present in this paper what we consider the most relevant results. In Sections 5, 6, and 7 we present *all* our experiments along with their corresponding discussion.

# 2. HASHING SCHEMES

In this paper, we study the performance of five different hashing schemes: (1) chained hashing, (2) linear probing, (3) quadratic probing, (4) Robin Hood hashing on linear probing, and (5) Cuckoo hashing — the last four belong to the so-called open-addressing schemes, in which *every* slot of the hash table stores exactly one element, or stores special values denoting whether the corresponding slot is free. For open-addressing schemes we assume that the tables have $l$ slots ($l$ is called *capacity* of the table). Let $0 \leq n \leq l$ be the number of occupied slots (we call $n$ the *size* of the table) and consider the ratio $\alpha = \frac{n}{l}$ as the *load factor* of the table. For chained hashing, the concept of load factor makes in general little sense since it can store more than one element in the same slot using a linked list, and thus we could obtain $\alpha > 1$. Hence, whenever we discuss chained hashing for a load factor $\alpha$, we mean that the presented chained hash tables are memory-wise comparable to open-addressing hash tables at load factor $\alpha$ — in particular, the hash tables contain the same number $n$ of elements, but their directory size can differ. We elaborate on this in Section 4.5.

Finally, one fundamental question in open-addressing is whether to organize the table as array-of-structs (AoS) or as a struct-of-arrays (SoA). In AoS, the table is stored in one (or more in case of Cuckoo hashing) arrays of key-value pairs, similar to a row layout. In contrast to that, SoA representation keeps keys and corresponding values separated in two corresponding, aligned arrays - similar to column layout. We found in a micro-benchmark that AoS is superior to SoA in most relevant cases for our setup and hence apply this organization in all open-addressing schemes in this paper. For more details on this micro-benchmark see Section 7. We now proceed to briefly describe each considered hashing scheme in turn.

## 2.1 Chained Hashing

Standard chained hashing is a very simple approach for collision handling, where each slot of table $T$ (the directory) is a pointer to a linked list of entries. On inserts, entries are appended to the list that corresponds to their key $k$ under hash function $h$, i.e., $T[h(k)]$. In case of lookups, the linked list under $T[h(k)]$ is searched for the entry with key $k$. Chained hashing is a simple and robust method that is widely used in practice, e.g., in the current implementations of `std::unordered_map` in `C++` STL or `java.util.HashMap` in `Java`. However, compared to open-addressing methods, chained hashing has typically sub-optimal performance for integer keys w.r.t. runtime and memory footprint. Two main reasons for this are: (1) the pointers used by the linked lists lead to a high memory overhead and (2) using linked lists leads to additional cache misses (even for slots with one element and no collisions). This situation brings different opportunities for optimizing a traditional chained hash table. For example, we can reduce cache misses by making the directory wide enough (say 24-byte entries for key-value-pointer triplets) so that we can *always* store one element directly in the directory and avoid following the corresponding pointer. Collisions are then stored in the corresponding linked list. In this version we potentially achieved the latency of open-addressing schemes (if collisions are rare) at the cost of space. Throughout the paper we denote the two versions of chained hashing we mentioned by **ChainedH8**, and **ChainedH24** respectively.

In the very first set of experiments we studied the performance of ChainedH8, and ChainedH24 under a variety of factors, as to better understand the trade-offs they offer. One key observation that we would like to point out at this point is: We observed that entry allocation in the linked lists is a key factor for insert performance in all our variants of chained hashing. For example, a naive approach with dynamic allocation, i.e., using one `malloc` call per insertion, and one `free` call per delete, lead to a significant overhead. For most use cases, an alternative allocation strategy provides a considerable performance benefit. That is, for both chained hashing methods in our indexing experiments, Sections 5 and 6, we use a *slab allocator*. The idea is to bulk-allocate many (or up to all) entries in one large array and store all map entries consecutively in this arrays. This strategy is very efficient in all scenarios where the size of the hash table is either known in advance or only growing. We ob-

served an improvement over traditional allocation in both: memory footprint (due to less fragmentation and less `malloc` metadata) as well as raw performance (by up to one order of magnitude!).

## 2.2 Linear Probing

Linear probing (**LP**) is the simplest scheme for collision handling in open-addressing. The hash function is of the following form: $h(k,i) = (h'(k) + i) \mod l$, where $i$ represents the $i$-th probed location and $h'(k)$ is an auxiliary hash function. It works as follows: First, try to insert each key-value pair $p = \langle k, v \rangle$ with key $k$ at the optimal slot $T[h(k,0)]$ in an open-addressing hash table $T$. In case $h(k,0)$ is already occupied by another entry with different key, we (circularly) probe the consecutive slots $h(k,1)$ to $h(k,l-1)$. We store $p$ in the first free slot $T[h(k,i)]$, for some $0 < i < l$, we encounter[2]. We define the *displacement* $d$ of $p$ as $i$, and the sum of displacements over all entries as the *total displacement* of $T$. Observe that the total displacement is a measure of performance in linear probing since a high value implies long probe sequences entries during lookups.

The rather simple strategy of LP has two advantages: (1) Low code complexity which allows for fast execution and (2) Excellent cache efficiency due to the sequential linear scan. However, on high load factors $> 60\%$, LP noticeably suffers from *primary clustering*, i.e., a tendency to create long sequences of filled slots and hence high total displacement. We will address those areas of occupied slots that are adjacent w.r.t. probe sequences as clusters. Further, we can also observe that unsuccessful lookups worsen the performance of LP since they require a complete scan of all slots up to the first empty slot. Linear probing also requires dedicated handling of deletes, i.e., we can not simply remove entries from the hash table because this could disconnect a cluster and produce incorrect results under lookups. One option to handle deletes in LP are the so called *tombstones*, i.e., a special value (different from the empty slot) that marks deleted entries so that lookups continue scanning after seeing one tombstone — yielding correct results. Using tombstones makes deletes very fast. However, tombstones can have a negative impact on performance, as they potentially connect otherwise unconnected clusters, thus building larger clusters. Inserts can replace a tombstone that is found during a probe after confirming that the key to insert is not already contained. Another strategy to handle deletes is *partial cluster rehash*: we delete the entry from the slot and rehash all following entries in the same cluster. For our experiments we decided to implement an optimized version of tombstones which will only place tombstones when required to keep a cluster connected (i.e. only if the next slot from the deleted entry is occupied). Placing tombstones is very fast (faster in general than rehashing after every deletion), and the only negative point about tombstones are lookups after a considerable amount of deletions — in such a case we could shrink the hash table and perform a rehash anyway.

One of our main motivations to study linear hashing in this paper is not only that it belongs to the classical hashing schemes, which dates to the 50's [15], but also the recent developments regarding its analysis. Knuth was the first [14] to give a formal analysis of the operations of linear probing (insertion, deletions, lookups) and he showed that all these operation can be performed in $O(1)$ using *truly* random hash functions[3]. However, very recently [20] it was shown that linear probing with tabulation hashing (see Section 3.3) as a function matches asymptotically the bounds of Knuth in expected running time $O(\frac{1}{\varepsilon^2})$, where the hash table has capacity $l = (1+\varepsilon)n$. That is, from a theoretical point of view, there is no reason to use any other hashing table. We will see in our experiments, however, that the story is slightly different in practice.

[2]Observe that as long as the table is not full, an empty slot is found.

[3]Which map *every* key in a given universe of keys independently and uniformly onto the hash table.

## 2.3 Quadratic Probing

Quadratic probing (**QP**) is another popular approach for collision handling in open-addressing. The hash function in QP is of the following form: $h(k,i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \mod l$, where $i$ represents the $i$-th probed location, $h'$ is an auxiliary hash function, and $c_1 \geq 0$, $c_2 > 0$ are auxiliary constants.

In case that the capacity of the table $l$ is a power of two and $c_1 = c_2 = 1/2$, it can be proven that quadratic probing will consider *every* single slot of the table one time in the worst case [6]. That is, as long as there are empty slots in the hash table, this particular version of quadratic probing will *always* find them eventually. Compared to linear probing, quadratic probing has a reduced tendency for primary clustering and comparably low code complexity. However, QP still suffers from so-called *secondary clustering*: if two different keys collide in the very first probe, they will also collide in all sub-sequent probes. For deletions, we can apply the same strategies as in LP. Our definition of displacement for LP carries over to QP as the number of probes $0 < i < l$ until an empty slot is found.

## 2.4 Robin Hood Hashing on LP

Robin Hood hashing [5] is an interesting extension that can be applied to many collision handling schemes, e.g., linear probing [23]. For the remainder of this paper, we will only talk about Robin Hood hashing on top of LP and simply refer to this combination as Robin Hood hashing (**RH**). Furthermore, we introduce a new tuned approach to Robin Hood hashing that improves on the worst-case scenario of LP (unsuccessful lookups on high load factors) at a small cost on inserts, and very high rates of successful lookups (close to 100%, best-case scenario).

In general, RH is based on the observation that collisions can be resolved in favor of any of the keys involved. With this additional degree of freedom, we can modify the insertion algorithm of LP as follows: On a probe sequence to insert a new entry $e_{new}$, whenever we encounter an existing entry $e_{old}$ with displacement $d(e_{new}) > d(e_{old})$[4], we exchange $e_{old}$ by $e_{new}$ and continue the search for an empty slot with $e_{old}$. As a result, the variance in displacement between all entries (and hence their variance in lookup times) is minimized. While this approach does not change the total displacement compared to LP, we can exploit the established ordering in other ways. In this sense, the name Robin Hood was motivated by the observation that the algorithm takes from the"rich" elements (with smaller displacement) and gives to the "poor" (with higher displacement). Thus distributing the "wealth" (proximity to optimal slot) more fairly across all elements without changing the average "wealth" per element.

It is known that RH can reduce the variance in displacement significantly over LP. Previous work [23] suggests to exploit this property to improve on unsuccessful lookups in several ways. For example, we could already start searching for elements at the slot with expected (average) displacement from their perfect slot and probe bidirectional from there. In practice, this is not very efficient due to high branch misprediction rates and/or unfriendly access pattern. Another approach introduces an early abort criterium for unsuccessful lookups. If we keep track of the maximum displacement $d_{max}$ among all entries in the hash table, a probe sequence can already stop after $d_{max}$ iterations. However, in practice we observed

[4]If $d(e_{new}) = d(e_{old})$ we can compare the actual keys as tie breaker to establish a full ordering.

that $d_{max}$ is often still too high[5] to obtain significant improvements over LP. We can improve on this method by introducing a different abort criterion, which compares the probe iteration $i$ with the displacement of currently probed entry $d(e_i)$ in every step and stops as soon as $d(e_i) < i$. However, comparing against $d(e_i)$ on each iteration requires us to either store displacement information or recalculate the hash value. We found all those approaches to be prohibitively expensive w.r.t. runtime and inferior to the plain LP in most scenarios. Instead, our approach applies early abortion by hash computation only on every $m$-th probe, where a good choice of $m$ is slightly bigger than the average displacement in the table. As computing the average displacement under updates can be expensive, a good sweet spot for most load factors is to check once at the end of each cache-line. We found this to give a good tradeoff between an overhead for successful probes and the ability to stop unsuccessful probes early. Hence, this is the configuration we use for RH in our experiments. Furthermore, our approach to RH applies partial rehash for deletions which turned out to be superior to tombstones for this table. Notice that tombstones in RH would, for correctness, require to store information that allow us to reconstruct the displacement of the deleted entry.

## 2.5 Cuckoo Hashing

Cuckoo hashing [19] (**CuckooH**) is a another open-addressing scheme that, in its original (and simplest) version, works as follows: There are two hash tables $T_0, T_1$, each one having its own hash function $h_0$ and $h_1$ respectively. Every inserted element $p$ is stored at either $T_0[h_0(p)]$ or $T_1[h_1(p)]$ but *never* in both. When inserting an element $p$, location $T_0[h_0(p)]$ is first probed, if the location is empty, $p$ is stored there, otherwise, $p$ kicks out the element $q$ already found at that location, $p$ is stored there, and $q$ is tried to be inserted at location $T_1[h_1(q)]$. If this location is free, $q$ is stored there, otherwise $q$ kicks out the element therein, and we repeat: in iteration $i \geq 0$, location $T_j[h_j(\cdot)]$ is probed, where $j = i \mod 2$. In the end we hope that *every* element finds its own "nest" in the hash table. However, it may happen that this process enters a loop, and thus a place for each element is never found. This is dealt with by performing only a fixed amount of iterations, once this limit is achieved, a rehash of the complete set is performed by choosing two new hash functions. The advantages of CuckooH are (1) For lookups, traditional CuckooH requires at most two tables accesses, which is in general optimal among hashing schemes using linear space. In particular, the load factor has only a small impact on the lookup performance of the hash table. (2) CuckooH has been reported [19] to be competitive with other good hashing schemes, like linear and quadratic probing, and (3) CuckooH is easy to implement. However, it has been empirically observed [19, 12] that the load factor of traditional CuckooH with 2 tables should stay slightly below $50\%$ in order to work. More precisely, below $50\%$ load factor creation succeeds with high probability, but it starts failing from $50\%$ on [11, 18]. This problem can be alleviated by generalizing CuckooH to use more tables $T_0, T_1, T_2 \ldots T_k$, each having its own hash function $h_k$, $k > 1$. For example, for $k = 4$ the load factor (empirically) increases to $96\%$ [12]. All this at the expense of performance, since now lookups require at most four table lookups. Furthermore, Cuckoo hashing is very sensitive to what hash functions are used [19, 10, 20] and requires robust hash functions. In our experiments we only consider Cuckoo hashing on four tables (called **CuckooH4**) since we want to study the performance of hash tables under many different load factors, that go up to $90\%$, and CuckooH4 is the only version of traditional Cuckoo hashing that offers this flexibility.

[5]For high load factor $\alpha$, $d_{max}$ can often be an order of magnitude higher than the average displacement.

# 3. HASH FUNCTIONS

In our study we want to investigate the impact of different hash functions in combination with various hashing schemes (Section 2) under different key distributions. Our set of hash functions covers a spectrum of different theoretical guarantees that also admit very efficient implementations (low code complexity) and thus are also used in practice. We also consider one hash function that is, in our opinion, the most representative member of a class of engineered hash functions[6] that *do not* necessarily have theoretical guarantees, but that show good empirical performance, and thus are widely used in practice. We believe that our chosen set of hash functions is very representative and offers practitioners a good set of **hash functions for integers** (64-bit in this paper) to choose from. The set of hash functions we considered is: (1) Multiply-shift [8], (2) Multiply-add-shift [7], (3) Tabulation hashing [20], and (4) Murmur hashing [2]. Formally, (1) is the weakest and (3) is the strongest w.r.t. randomization. The definition and properties of these hash functions are as follows:

## 3.1 Multiply-shift

Multiply-shift (**Mult**) is *very* well known [8], and it is given here:

$$h_z(x) = (x \cdot z \mod 2^w) \texttt{ div } 2^{w-d}$$

where $x$ is a $w$-bit integer in $\{0, \ldots, 2^w - 1\}$, $z$ is an odd $w$-bit integer in $\{1, \ldots, 2^w - 1\}$, the hash table is of size $2^d$, and the `div` operator is defined as: $a \texttt{ div } b = \lfloor a/b \rfloor$. What makes this hash function highly interesting is: (1) It can be implemented extremely efficiently by observing that the multiplication $x \cdot z$ is natively done modulo $2^w$ in current architectures for native types like 32- and 64-bit integers, and the operator div is equivalent to a right bit shift by $w - d$ positions. (2) It has been proven [8] that if $x, y \in \{0, \ldots, 2^w - 1\}$, with $x \neq y$, and if $z \in \{1, \ldots, 2^w - 1\}$ chosen uniformly at random, then the collision probability is $\frac{1}{2^{d-1}}$.

This also means that the family of hash functions $H_{w,d} = \{h_z \mid 0 < z < 2^w \text{ and } z \text{ odd}\}$ is the ideal candidate for simple and rather robust hash functions. Multiply-shift is a universal hash function.

## 3.2 Multiply-add-shift

Multiply-add-shift (**MultAdd**) is also a very well known hash function [7]. It's definition is very similar to the previous one:

$$h_{a,b}(x) = ((x \cdot a + b) \mod 2^{2w}) \texttt{ div } 2^{2w-d}$$

where again $x$ is a $w$-bit integer, $a, b$ are two $2w$-bit integers, and $2^d$ is the size of the hash table. For $w = 32$ this hash function can be implemented natively under 64-bit architectures, but $w = 64$ requires 128-bit arithmetic which is still not widely supported natively. It can nevertheless still be implemented (keeping its formal properties) using only 64-bit arithmetic [22]. When $a, b$ are randomly chosen from $\{0, \ldots, 2^{2^w}\}$, it can be proven that collision probability is $\frac{1}{2^d}$, and thus is stronger than Multiply-shift — although it also incurs into heavier computations. Multiply-add-shift is a 2-independent hash function.

## 3.3 Tabulation hashing

Tabulation hashing (**Tab**) is the strongest hash function among all the ones that we consider and also probably the least known. It became more popular in recent years since it can be proven [20]

[6]Like FNV, CRC, DJB, CityHash for example.

that tabulation and linear probing achieve $O(1)$ for insertions, deletions, and lookups. This produces a hash table that is, in asymptotic terms, unbeatable. Its definition is as follows (we assume 64-bit keys for simplicity): Split the 64-bit keys into $c$ characters, say eight chars $c_1, \ldots, c_8$. For every position $1 \leq i \leq 8$ initialize a table $T_i$ with 256 entries (for chars) with *truly* 64-bit random codes. The hash function for key $x = c_1 \cdots c_8$ is then:

$$h(x) = \bigoplus_{i=1}^{8} T_i[c_i]$$

where $\bigoplus$ denotes the bitwise XOR. So a hash code is composed by the XOR of the corresponding entries in tables $T_i$ of the characters of $x$. If all tables are filled with truly random data, then it is known that tabulation is 3-independent (but not stronger), which means that for *any* three distinct keys $x_1, x_2, x_3$ from our universe of keys, and three (not necessarily distinct) hash codes $y_1, y_2, y_3 \in \{0, \ldots, l\}$ then

$$Pr[h(x_1) = y_1 \wedge h(x_2) = y_2 \wedge h(x_3) = y_3] \leq \frac{1}{l^3}$$

which means that under tabulation hashing, the hash code $h(x_i)$ is uniformly distributed onto the hash table for *every* key in our universe, and that for *any* three distinct keys $x_1, x_2, x_3$, the corresponding hash codes are three independent random variables.

Now, the interesting part of tabulation hashing is that it requires only bitwise operations, which are very fast, and lookups in tables $T_1, \ldots, T_8$. These tables are as heavy as $256 \cdot 8 \cdot 8$ B $= 16$ KB. Which mean that they *all* fit comfortably in the L1 cache of processors, which is 32 or 64 KB in modern computing servers. That is, lookups in those tables incur in potentially low latency operations, and thus the evaluation of single hash codes is potentially very fast.

## 3.4 Murmur hashing

Murmur hashing (**Murmur**) is one of the most common hash functions used in practice due to its good behavior. It is relatively fast to compute and it seems to produce quite good hash codes. We are not aware of any formal analysis on this, so we use Murmur hashing *essentially* as is. As we limit ourselves in this paper to 64-bit keys, we use Murmur3's 64-bit finalizer [2] as shown in the code below.

```
uint64_t murmur3_64_finalizer(uint64_t key) {
      key ^= key >> 33;
      key *= 0xff51afd7ed558ccd;
      key ^= key >> 33;
      key *= 0xc4ceb9fe1a85ec53;
      key ^= key >> 33;
      return key;}
```

# 4. METHODOLOGY

Throughout the paper we want to understand how well a hash table can work as a plain index for a set of $n$ ⟨key, value⟩ pairs of 64-bit integers. The keys obey three different data distributions, described later on in Section 4.3. This scenario, albeit generic, resembles very closely other interesting uses of hash tables such as in join processing or in aggregate operations like AVERAGE, SUM, MIN, MAX, and COUNT. In fact, we performed experiments simulating these operations, and the results were comparable those from the WORM workload.

We study the relation between (raw) performance and load factors by performing insertions and lookups (successful and unsuccessful) on hash tables at different load factors. For this we consider a write-once-read-many (**WORM**) workload, and a mixed read-write (**RW**) workload. These two kinds of workload simulate elementary operational requirements of OLAP and OLTP scenarios, respectively, for index structures.

## 4.1 Setup

All experiments are **single-threaded** and all implementations are our own. All hash tables have map semantics, i.e., they cover both key and value. All experiments are in **main memory**. For the experiments in Sections 5 and 6 we use a single core (one NUMA region) of a dual-socket machine having two hexacore Intel Xeon Processors X5690 running at 3.47 GHz. The machine has a total of 192 GB of RAM running at 1066 MHz. The OS is a 64-bit Linux (3.2.0) with **page size** of 2 MB (using transparent huge pages). All algorithms are implemented in C++ and compiled with `gcc-4.7.2` with optimization `-O3`. Prefetching, hyper-threading and turbo-boost are disabled via BIOS to isolate the real characteristics of the considered hash tables.

Since our server does not support AVX-2 instructions, we ran the layout and SIMD evaluation, Section 7, on a MacbookPro with Intel Core i7-4980HQ running at 2.80GHz (Haswell) with 16 GB DDR3 RAM at 1600 MHz running Mac OS X 10.10.2 in single-user mode. Here, the page size is 4 KB and pre-fetching is activated since we could not deactivate it as cleanly as for our linux server. All binaries are compiled with `clang-600.0.56` with optimization `-O3`.

## 4.2 Measurement and Analysis

For all indexing experiments of Sections 5 and 6 we report the average of three independent runs (three different random seeds for the generation and shuffling of data). We performed an analysis of variance on all results and we found that, in general, the results are overall very stable and uniform. Whenever variance was noticeable, we reran the corresponding experiment with the same setting to rule out machine problems. As variance was very insignificant, we decided that there is no added benefit in showing it in the plots.

## 4.3 Data distributions

Every indexed key is **64** bits. We consider three different kinds of data distributions: **Dense**, **Sparse**, and **Grid**. In the dense distribution we index *every* key in $[1 : n] := \{1, 2, \ldots, n\}$. In the sparse distribution, $n \ll 2^{64}$ keys are generated uniformly at random from $[1 : 2^{64} - 1]$. In the grid distribution *every* byte of *every* key is in the range $[1 : 14]$. That is, the universe under the grid distribution consists of $14^8 = 1,475,789,056$ different keys, and we use only the first $n$ keys (in the sorted order). Thus, the grid distribution is also a different kind of dense distribution. Elements are randomly shuffled before insertion, and the set of lookup keys is also randomly shuffled.

## 4.4 Narrowing down our result set

Our overall set of experiments contained the combinations of many different dimensions, and thus the amount of raw information obtained exceeds legibility easily and makes the presentation of the paper very difficult. For example, there are in total 24 different hash tables (hashing scheme + hash function). Thus, if we wanted to present all of them, *every* plot would contain 24 different curves, which is too much information for a single plot. Thus, we decided to present in this paper only the most representative set of results. We will make the complete set of results available in a technical report version of this paper. Therefore, although we originally considered four different hash functions Mult, MultAdd, Tab, and Murmur, see Section 3, the following observations were uniform across *all* experiments: (1) **Mult is the fastest hash function when integrated with all hashing schemes, i.e., producing the highest throughputs and also of good quality (robustness), and thus it definitely deserves to be presented**. (2) MultAdd, when integrated with hashing schemes, has a robustness that falls between Mult and

Murmur — more robust than Mult but less than Murmur. In terms of speed it was slower (in throughput) than Murmur. Thus we decided not to present MultAdd here and present Murmur instead. (3) Tabulation was indeed the strongest, most robust hash function of all when integrated with all hashing schemes. However, it is also the slowest, i.e., producing the lowest throughput. By studying the results provided by Mult and Murmur, we think that the trade-off for by tabulation (robustness instead of speed) is less attractive in practice. Hence we do not present results for tabulation here.

In the end, we observed the importance of reducing operations during hash code computations as much as possible. Mult, for example, requires only *one* multiplication and *one* right bit shift — it is by far the lightest to compute. MultAdd for 64-bit keys without 128-bit arithmetic [22] (natively unsupported on our server) requires two multiplications, six additions, plus a number of logical `AND`s and right bit shifts, which is more expensive than Murmur's 64-bit finalizer which requires *only* two multiplications and a number of `XOR`s and right bit shifts. As for tabulation, the eight table lookups per key ended up dominating its execution time. Assuming all tables remain in L1 cache, the latency of each table lookup is around 5-10 clock cycles. One addition requires one clock cycle and one multiplication at most five clock cycles (on Intel architectures). Thus, it is *very* interesting to observe and understand that, when hash code computation is part of hot loops during a workload (as in our experiments), we should really be concerned about how many clock cycles each computation costs — we could observe the effect of *even* one more instruction per hash code computation. We want to point out as well that **the situation of MultAdd changes if we use native 128-bit arithmetic, or if we use 32-bits keys with native 64-bit arithmetic (one multiplication, one addition, and one right bit shift). In that case we could use MultAdd instead of Murmur for the benefit of proven theoretical properties.**

## 4.5 On load factors for chained hashing

As we mentioned before, the load factor makes almost no sense for chained hashing since it can exceed one. Thus, throughout the paper we refrain ourselves from using the formal definition of load factor together with chained hashing. We will instead study chained hashing under memory budgets. That is, whenever we compare chained hashing against open-addressing schemes at a given load factor $\alpha = \frac{n}{l}$, what we do is that we modify the size of the directory of the chained hash table so that its overall memory consumption does not exceeds $110\%$ of what open-addressing schemes require. In such a comparison, *all* hash tables will contain the *exact* same number $n$ of elements. Thus, *all* hash tables compute the *exact* same number of hashes. In this regard, whether or not a chained hash table stays within memory constraints depends on the number of chained entries. Both variants of chained hashing considered by us can not place more than a fraction of $16/24 < 0.67$ of the total of elements that an open-addressing scheme could place under the same memory constraint. If we take the extra $10\%$ we grant to chained hash tables into account, this fraction grows to roughly $0.73$. However, in practice this threshold is smaller ($< 0.7$) due to how collisions distribute over the table. This already strongly limits the usability of chained hashing under memory constraints and also brings up the following interesting situation. If chained hashing has to work under memory constraints, we can also try an open-addressing scheme for the exact same task under the same amount of memory. This potentially means lower load factors ($< 0.5$) for the latter. Depending on the hash function used, collisions might thus be rare, and the performance might become similar to a direct-addressing scheme — which is ideal. This might render chained hashing irrelevant.

# 5. WRITE-ONCE-READ-MANY (WORM)

In WORM we are interested in build and probe times (read-only structure) under six different load factors $25\%, 35\%, 45\%, 50\%, 70\%, 90\%$. These load factors are w.r.t. open addressing schemes on three different **pre-allocated capacities**[7]: $2^{16}$ (small — 1 MB), $2^{27}$ (medium — 2 GB) and $2^{30}$ (large — 16 GB). This gives a total of up to **54 different configurations** (three data distributions, six load factors, and three capacities) for *each* of the **24 hash tables**. Due to the lack of space, and by our discussion offered on the load factors of chained hashing, we present here only the subsets of the large capacity presented in Figure 1.

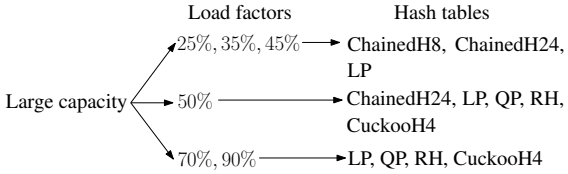


***Figure 1:*** *Subset of results for WORM presented in this paper.*

The main reason for presenting only the large capacity is that "big" datasets are nowadays of primary concern and most observations can be transferred to smaller datasets. Also, we divided the hash tables this way because, by our explanation before, at low load factors collisions will be rare and performance of open-addressing schemes will be chiefly dominated by the simplicity of the used hash table — i.e., low code complexity. Thus we decided to compare the two variants of chained hashing against the simplest open-addressing scheme (linear probing)[8]. At a load factor of $50\%$, collision resolution of different open-addressing schemes start becoming apparent and thus from that point on we include all open-addressing schemes considered by us. For chained hashing we consider only the best performing variant. For higher load factors ($\geq 70\%$), however, both variants of chained hashing could not place enough elements in the allocated memory. Thus we removed them altogether and study only open-addressing schemes.

## 5.1 Low load factors: 25%, 35%, 45%

In our very first set of experiments we are interested in understanding (1) the fundamental difference between chained hashing and open-addressing and (2) the trade-offs offer by the two different variants of chained hashing. The results can be seen in Figure 2.

**Discussion.** We start by discussing the memory footprints of all structures, see Figure 3. For linear probing, the footprint is constant (16 GB), independent of the load factor, and easily determined only be the size of the directory, i.e., $2^{30}$ slots of 16 B each. In ChainedH8, the footprint is calculated as size of directory, i.e. $2^{30}$ or $2^{29}$ slots, times the pointer size — 8 B. In addition to that come 24 B for each entry in the table. The footprint of ChainedH24 is computed as directory size, $2^{29}$, times 24 B, plus 24 B for each collision. From this data we can obtain the amount of collisions for ChainedH24. For example, at load factor $35\%$, ChainedH24 requires 12 GB for the directory, and all that goes beyond that is due to collisions. Thus, for the sparse distribution for example, ChainedH24 deals with $\approx 28\%$ rate of collisions. But under the dense distribution, it deals only with $\approx 3\%$ collision rate using Mult as hash function.

For performance results, let us focus on multiplicative hashing (Mult). Here, we can see a clear and stable ranking among the

[7]WORM is a static workload. This means that the hash tables *never* rehash during the workload.

[8]For insignificant amounts of collisions, the performance of LP, RH, and QP is essentially equivalent.

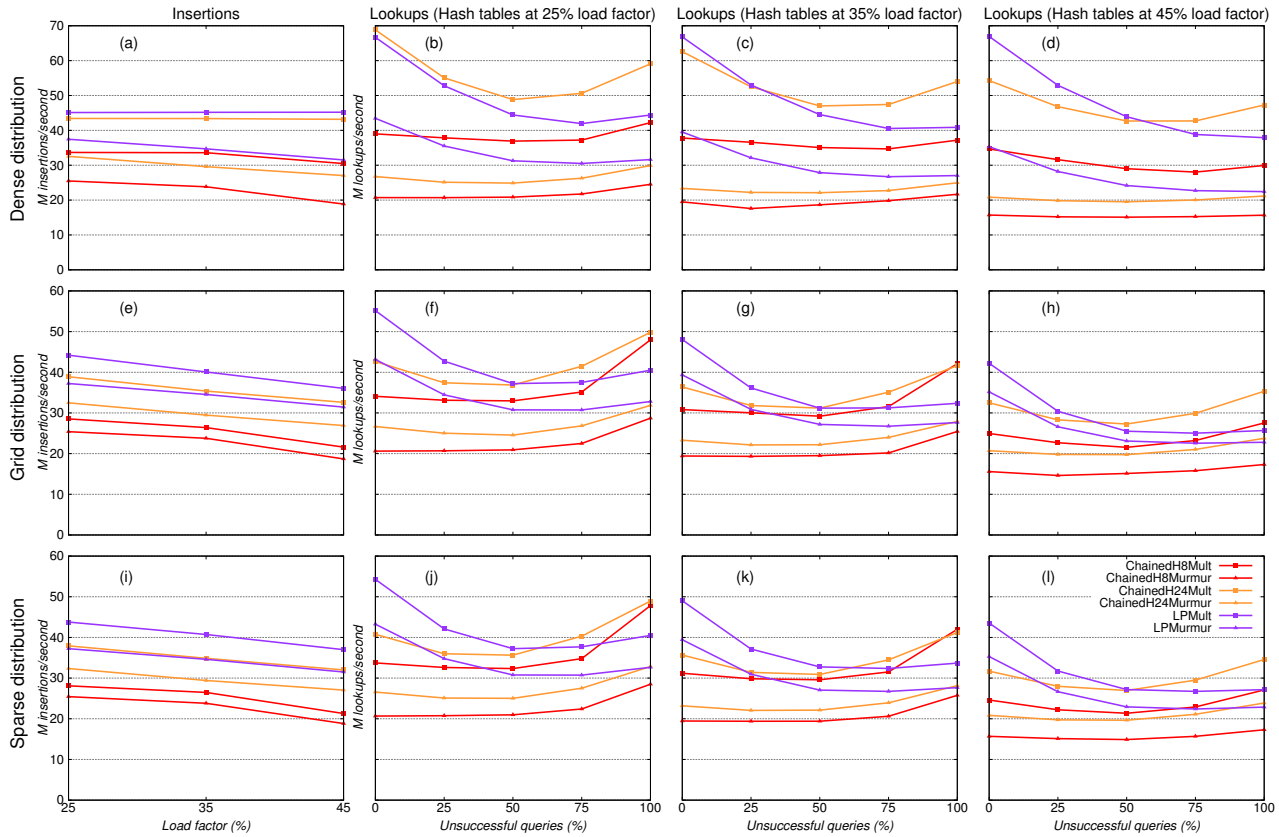


***Figure 2:*** *Insertion and lookup throughputs, comparing two different variants of chained hashing with linear probing, under three different distributions at load factors* $25\%, 35\%, 45\%$ *from* $2^{30}$ *for linear probing. Higher is better.*

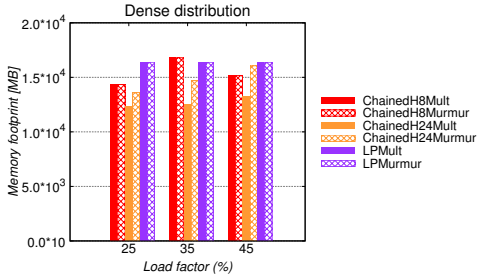


***Figure 3:*** *Memory usage under the dense distribution of the hash tables presented in Figure 2. This distribution produces the largest differences in memory usage among hash tables. For the sparse and grid distributions, memory of ChainedH24Mult matches that of ChainedH24Murmur, and the rest remain the same. Lower is better.*

methods. For inserts, ChainedH24 performs better than ChainedH8. This is expected as the inlining of ChainedH24 helps to avoid caches misses for all occupied slots. Linear probing is, however, the top performer. This is because low load factors allow for many in-place insertions to the perfect slot.

In terms of lookup performance, we can also find a clear ranking among the chained hashing variants. Here, ChainedH24 performs best again. The superior performance of ChainedH24 is again easily explainable by the lower amount of pointer-chasing in the structure. We can also observe that between LP and ChainedH24, in all cases, one of the two structures performs best — but each in a different case and the order is typically determined by the ratio of unsuccessful queries. For all successful lookups, LP outperforms, in all but one case, all variants of chained hashing. The only exception is under the dense distribution at $25\%$ load factor, Figure 2(b). There, both methods are essentially equivalent because the amount of collisions is essentially zero. The difference we observe is due to variance in code complexity and different directory sizes — smaller directories lead to better cache behavior. Otherwise, in general, LP improves significantly over ChainedH24 if most queries are successful. In turn, ChainedH24 improves over LP, also by a significant amount in general, if most lookups are unsuccessful. We typically find the crossover point at around $50\%$ unsuccessful lookups. Interestingly, in some cases we can even observe ChainedH8 performing slightly better than LP for $100\%$ unsuccessful lookups. This is explainable because even when collisions are rare, primary clusters can build up in linear probing (think of a continuous sequence of perfectly placed elements). For *every* unsuccessful query, LP has to scan until it finds an empty slot, and as the amount of unsuccessful queries increases, LP becomes considerably slower. If the unsuccessful query falls into a primary cluster, chained hashing answers right away if it detects and empty slot, or it will follow the linked list until the end. However, linked lists are very short on average. The highest observed collision rate is $\approx 34\%$ (sparse distribution at $45\%$ load factor). This means that, at most, roughly one-third of the elements are outside the directory. Under the probabilistic properties of Mult, it can be argued that the linked list in chained hashing are in expectation of length at most 2, and thus chained hashing follows on average at most two pointers. **We can conclude that, at low load factors ($< 50\%$), LPMult is the way to go if most queries are successful ($\geq 50\%$), and ChainedH24 must be considered otherwise**.

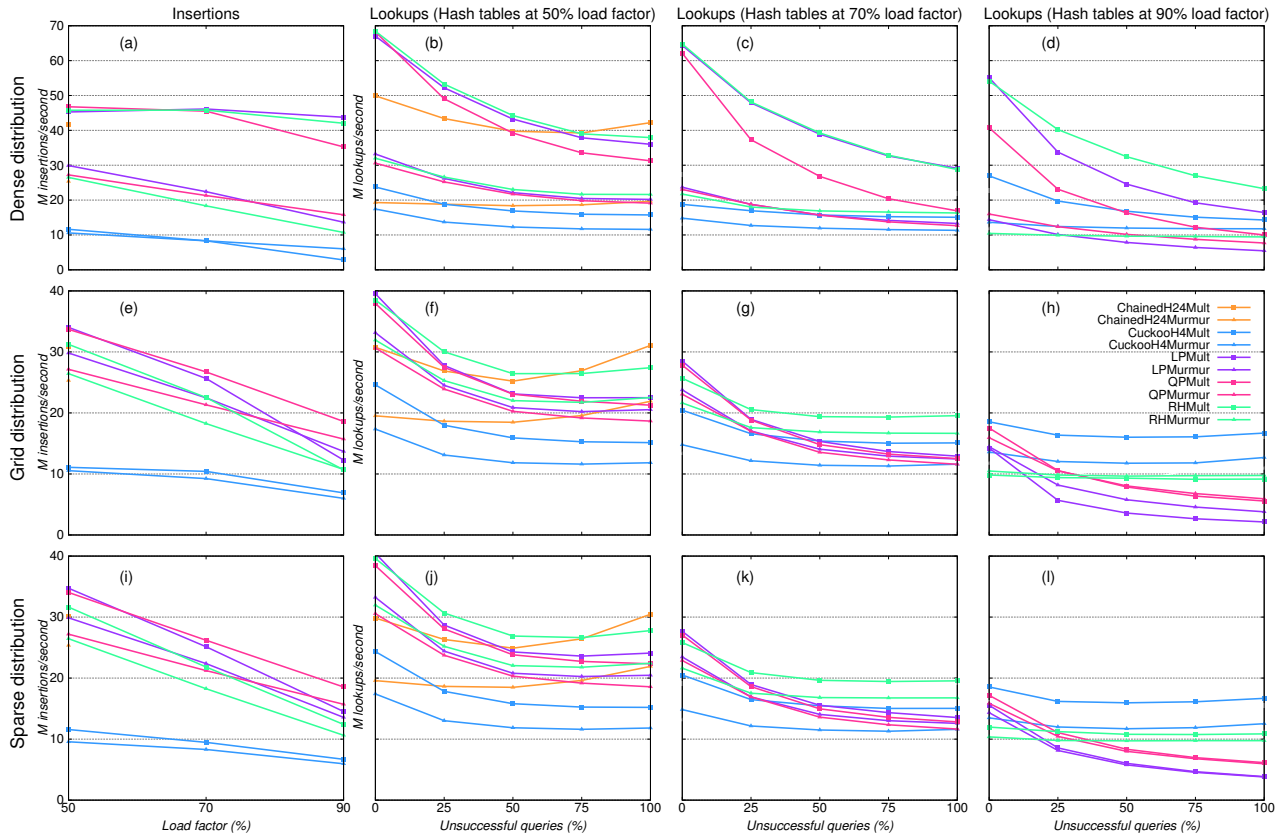


***Figure 4:*** *Insertion and lookup throughputs, open-addressing variants and chained hashing, under three different distributions at load factors* $50\%, 70\%, 90\%$ *from* $2^{30}$*. Higher is better. Memory consumption for all open-addressing schemes is* $16$ *GB, and* $16.4$ *GB for ChainedH24.*

## 5.2 High load factors: 50%, 70%, 90%

In our second set of experiments we study the performance of hash tables when space efficiency is required, and thus we are not able to use hash tables at low load factors. That is, we stress the hash tables to occupy up to $90\%$ of the space assigned to them (chained hashing is allowed up to $10\%$ more). We decided to use Cuckoo hashing on four tables, rather than on two or three tables, because this version of Cuckoo hashing is known to achieve load factors as high as $96.7\%$ [9, 12] with high probability. In contrast, Cuckoo hashing on two and three tables have stable load factors of $< 50$ and up to $\approx 88\%$ respectively [18]. This means that if in practice we want to consider *very* high load factors ($\geq 90\%$), then Cuckoo hashing on four tables is the best candidate. An overview of the absolute best performers w.r.t. the other two capacities (small and medium) is given as a table in Figure 6.

**Discussion.** Let us first start with a general discussion about the impact of distributions and hash functions on both, insert and lookup performance across all tables. Our first important observation is that Multiply-shift (Mult) performs essentially *always* better than Murmur hashing in this experiment. We can conclude from this that, overall, the improved quality of Murmur over Mult does not justify the higher computational effort. Mult seems already good enough to drive our five considered hash tables: ChainedH24, LP, QP, RP, and CuckooH4 up to the significantly high load factor of $90\%$ — observe that *no* hash table is the absolute best using Murmur, see all plots of Figure 4. Another interesting observation is that, while we can see a significant variance in throughput under Mult across different data distributions — compare for example the throughputs of dense and sparse distributions under Mult — this variance is minimal under Murmur. This indicates that Murmur provides a very good randomization of the input data, basically transforming all input distribution into a distribution that is very close to uniform, and hence the distribution seems not to have much effect under Murmur [9]. However, sensitivity of a hash function to certain data distributions is not necessarily bad. For example, under the dense distribution[10] Mult is known [15] to produce an approximate arithmetic progression as hash codes, which reduces collisions. For a comparison, just observe that the dense distribution achieves higher throughputs than the sparse distribution that is usually considered as an unbiased reference of speed. We have observed that the picture does not easily change, even in the presence of a certain degree of gaps in the sequence of dense keys. Overall this makes Mult a strong candidate for dense keys, which appear *very* often in practice, e.g., for generated primary keys. In contrast to that, Mult is slightly slower on the grid distribution compared to the sparse distribution. We could observe that Mult produces indeed more collisions than the expected amount on uniformly distributed keys. However, this larger amount of collisions does not get highly reflected in the observed performance. Thus, **we consider Mult as the best candidate to be used in practice when quality results on high throughputs is desired, but at the cost of a high variance across data distributions**.

Let us now focus on the difference between the hash tables. In general, it can immediately be seen that all open-addressing schemes, except CuckooH4, are better than ChainedH24 in almost all cases for up to $50\%$ unsuccessful lookups, see Figure 4 (a, b,

[9] We observed the same for Tab.

[10] Actually under a generalized dense distribution of keys following an arithmetic progression $k, k+d, k+2d, \ldots$

e, f, i, j). And only for the degenerated case of 100% unsuccessful lookups, ChainedH24 is the overall winner — for the same reasons as for low load factors. ChainedH24 is removed from the comparison for load factors $> 50\%$ because it exceeds the memory limit.

Between open-addressing schemes, things are more interesting. On insertions (leftmost column of Figure 4), we can observe a rather clear ranking among methods that holds across all distributions and load factors. CuckooH4 is showing a very stable insert performance that is only slightly affected by increasing load factors. However, this performance is rather low. We can explain this result by the expensive reorganization that happens during Cuckoo cycles, and can often incur into several cache misses (whenever an element is moved between the tables) for a single insert. Unsurprisingly, LP, QP, and RH show rather similar insert performance characteristics because their insertion algorithm is very similar. Starting with high performance at $50\%$ load factors, this performance drops significantly as the load factor increases. However, even under a high load factor, linearly and quadratically probing a hash table seems to be very effective. Among the three methods, we observe that RH is in general slightly slower than LP and QP. This is because RH performs small reorganizations on already inserted elements. However, these reorganizations often stay within one cache line, and thus the decrease in performance stays typically within less than 10%. With respect to QP and LP, the following are the most relevant observations. QP and LP have very similar insertion throughput for low load factors (up to 50). For higher load factors, when the difference in collision handling plays a role: (1) LPMult is considerable faster than QPMult under the dense distribution of keys (45M insertions/second versus 35M insertions/second — Figure 4(a)), and (2) QP (Mult/Murmur) is faster than LP (Mult/Murmur) otherwise. This is explainable: for (1) it suffices to observe that a dense distribution is the best case for LPMult – since Mult produces an approximate arithmetic progression (very few collisions). The best way to lay out an (approximate) arithmetic progression, in order to have better data locality, is to do so linearly, just as LP does. We could also observe that when primary clusters start appearing, they appear well distributed across the whole table, and they have similar sizes. Thus no cluster is arbitrarily long, which is good for LP. On the other hand, QP touches a new cache line in *every* probe subsequent to the third, and touching a new cache line results usually in a cache miss. Data locality is thus not optimal. For (2) the argument complements (1). Data is distributed more randomly, by the hash function, across the table. This causes an increment in collisions w.r.t. the combination $\langle$dense distribution + Mult$\rangle$. For high load factors this increment in collisions means considerable long primary clusters that LP has to deal with. In this case, QP is a better strategy to handle collisions since it scatters collisions more sparsely across the table, and chances to find empty slots fast, over the whole sequence of insertions, are better than in LP with considerable long primary clusters.

For lookups we can find a similar situation as for inserts. LP, QP, and RH perform better than CuckooH4 in many situations, i.e., up to relatively high load factors. However, the performance of the former three significantly decreases with (1) higher load factors and (2) more unsuccessful lookups. We could observe that from a load factor of $80\%$ on, CuckooH4 clearly surpasses the other methods. In general, LP, QP, and RH are better in dealing with higher collision rates than Cuckoo hashing, which is known to be negatively affected by "weak" hash functions [19] such as Mult. However, these "weak" hash functions affect *only* during the construction of the hash table, since once the hash table is constructed, then lookups in Cuckoo hashing are performed in constant time (four cache misses at most for CuckooH4). As such, Cuckoo hashing is also less affected by unsuccessful lookups than LP, QP, and RH. However, it seems that we can benefit from CuckooH4 only on very high load factors $\geq 80\%$.

As expected, the more complex re-organization that RH performs on the keys during insertions, see Section 2.4, can be seen to pay off under unsuccessful lookups — RH is much less affected by them than LP and QP. In RH, unsuccessful lookups can stop as soon as the displacement of the search key is exceeded by another key we encounter during the probe. Hence, RH does not necessarily require a complete scan of all adjacent keys in the same cluster, and can stop probing after less iterations than LP or QP. Clearly, this advantage of RH over LP and QP increases with higher load factors and higher rates of unsuccessful lookups — significantly improving on the worst-case of the methods. However, in the best of cases, i.e., when all lookups are successful, RH is slightly slower than the competitors. This is also expected as RH does not improve on the average displacement or amount of loaded cache lines w.r.t. LP (clusters contain *only* different permutations of the elements therein contained under RH and LP). When all lookups are successful, the (small) performance penalty of RH is due to its slightly more complex code. **We can conclude that RH provides a very interesting trade-off: for a small penalty (often within 1-5%) in peak performance on the best of cases (all lookups successful), RH significantly improves on the worst-case over LP in general, up to more than a factor 4**. Under the dense distribution — Figure 4 (a – c) — RH and LP have similar performance up to 70% load factor, but for 90% load factor, RH is significantly faster than LP (up to 40%) from 25% unsuccessful lookups on.

**Across the whole set of experiments, RH is always among the top performers, and even the best method for most cases. This observation holds for all data set sizes we tested**. In this regard, Figure 6 gives an overview and summarizes the absolute best methods we tested in this experiment under all capacities (small, medium, and large). Methods are color-coded as in the curves in the plots. Observe that patterns are nicely recognizable. For lookups in general, RH seems to be an excellent all-rounder unless the hash table is expected to be very full, or the amount of unsuccessful queries is rather large. In such cases, CuckooH4 and ChainedH24 would be better options, respectively, if their slow insertion times are acceptable. With respect to insertions, it is natural not to see RH appearing more often, and certainly CuckooH4 and ChainedH24 not at all, due to their complicated insertion procedures. For insertions, QP seems to be the best option in general. Even when LP or RH are sometimes better, the difference is rather small, less than 10%.

# 6. READ-WRITE WORKLOAD (RW)

In RW we are interested in analyzing how growing (rehashing) over a long sequence of operations affects overall throughput and memory consumption. The set of operations we consider is the following: insertions, deletions (all successful), and lookups (successful and unsuccessful). In RW we let the hash tables grow over a set of 1000 million operations that appear in random order. Each hash table initially contains 16 millions keys[11]. We set the insertion-to-deletion ratio (updates) to 4:1 ($20\%$ deletions), and the successful-to-unsuccessful-lookup ratio to 3:1 ($25\%$ unsuccessful queries). For this kind of workload we present here only the results concerning the sparse distribution of keys. We consider three different thresholds for rehashing: at $50\%, 70\%$, and $90\%$. Rehashing at $50\%$ allows us to always have enough empty slots, and thus also less collisions. However, this also means a potential loss

[11] In the beginning (no updates), the hash tables have a load factor of roughly $47\%$.

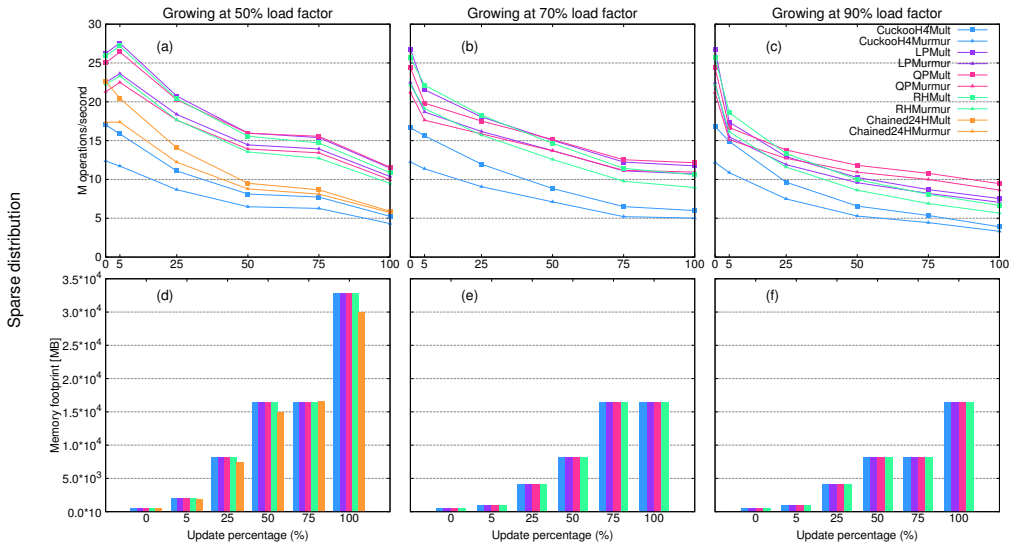


***Figure 5:*** *1000M operations of RW workload under different load factors and update-to-lookup ratios. For updates, the insertion-to-deletion ratio is 4:1. For lookups, the successful-to-unsuccessful-lookup is ratio 3:1. The key distribution is sparse. Higher is better in performance, lower is better for memory.*

| Dist. | Load factor | Capacity | Insertions | Unsuccessful queries (%) 0% | 25% | 50% | 75% | 100% |
|---|---|---|---|---|---|---|---|---|
| Dense | 50% | S | 290 | 345 | 126 | 91 | 114 | 188 |
| | | M | 56 | 76 | 56 | 44 | 46 | 51 |
| | | L | 46 | 68 | 53 | 44 | 39 | 42 |
| | 70% | S | 289 | 346 | 113 | 71 | 62 | 63 |
| | | M | 56 | 75 | 48 | 35 | 30 | 29 |
| | | L | 46 | 64 | 48 | 39 | 32 | 28 |
| | 90% | S | 169 | 183 | 80 | 63 | 61 | 64 |
| | | M | 51 | 60 | 37 | 27 | 22 | 21 |
| | | L | 43 | 55 | 40 | 32 | 26 | 23 |
| Grid | 50% | S | 194 | 212 | 95 | 72 | 85 | 124 |
| | | M | 38 | 43 | 33 | 29 | 28 | 32 |
| | | L | 34 | 39 | 30 | 26 | 26 | 31 |
| | 70% | S | 113 | 119 | 67 | 57 | 56 | 58 |
| | | M | 28 | 30 | 24 | 22 | 21 | 21 |
| | | L | 25 | 28 | 20 | 19 | 19 | 19 |
| | 90% | S | 72 | 90 | 52 | 47 | 49 | 52 |
| | | M | 20 | 22 | 19 | 19 | 20 | 20 |
| | | L | 18 | 18 | 16 | 16 | 16 | 16 |
| Sparse | 50% | S | 104 | 109 | 68 | 63 | 73 | 103 |
| | | M | 39 | 44 | 33 | 29 | 29 | 33 |
| | | L | 34 | 40 | 30 | 26 | 26 | 30 |
| | 70% | S | 77 | 74 | 52 | 50 | 51 | 53 |
| | | M | 28 | 30 | 22 | 21 | 20 | 20 |
| | | L | 26 | 27 | 20 | 19 | 19 | 19 |
| | 90% | S | 57 | 59 | 51 | 53 | 58 | 66 |
| | | M | 19 | 22 | 19 | 19 | 20 | 21 |
| | | L | 18 | 18 | 16 | 15 | 16 | 16 |

CuckooH4Mult LPMult QPMult RHMult ChainedH24

***Figure 6:*** *Absolute best performers for the WORM workload (Section 5.2) across distributions, different load factors, and different capacities: Small (S), Medium (M) and Large (L). Throughput of the corresponding hash table is shown inside its cell in millions of operations per second.*

in space since the workload might stop short after growing, and thus up to 75% of the hash table could be empty. On the other hand, rehashing at 90% deals with a large amount of collisions as the table gets full, but then we potentially waste less space. In addition to that, high load factors will incur into slow lookup times before a rehash. Observe again that by the natural load factors of Cuckoo hashing on two and three tables, Cuckoo hashing on four tables is the best candidate again for controlling at what load factor the hash table must rehash. For chained hashing, similar to the situation in WORM, we present here only the case where rehashing is performed at 50% load factor. This is the only case in which we can keep memory consumption of chained hashing (ChainedH24) comparable to what the open-addressing schemes require. The results of these experiments are shown in Figure 5.

**Discussion.** With respect to the performance in the WORM scenario on high load factors — Section 5.2 — the outcome of the RW comparison offers few surprises. One of these surprises is to see that ChainedH24 offers better performance than CuckooH4 (50% load factor only), and sometimes even by a large margin. They both, however, lag clearly behind the other (open-addressing) schemes. As RW workload is write-heavy, what we see in the plots is mostly the cost of table reorganization (rehashing) — except for data points at 0% updates. In that case, what we see are only lookups with 25% of unsuccessful queries, see Figure 4(j) for a comparison. For CuckooH4 the gap narrows as the load factor increases, see Figure 5(c), but is not enough to become really competitive with the best performers — which are at least twice as fast as the updates become more frequent. **As a conclusion, although memory requirements of ChainedH24 and CuckooH4 are competitive with that of the other schemes in a dynamic setting, both — chained and Cuckoo hashing — should be avoided for write-heavy workloads**.

We can also see that Mult governs again over Murmur on all hash tables — Figure 5 (a – c). Which is to be expected since the hash tables rehash many times and thus hash function computations are fundamental. Also, we always find LP, QP, and RH as the fastest methods, and often with very similar performance. Growing at 50% load factor — Figure 5(a) — the difference in throughput of all three methods is mostly within the boundary of variance. In case of high update percentage ($> 50\%$), we can observe a small performance penalty for RH in comparison to LP and QP, which is due to the slightly slower insert performance that we already observed in the WORM benchmark, see Figure 4(i). This is expected because at 50% load factor, there are few collisions, and more sophisticated strategies for handling collisions can not benefit as much. At 70% and 90% load factors — Figures 5(b) and 5(c) — all three methods are getting slower, and we can also observe a clearer difference between them because different strategies have an impact now. Interestingly, with increasing load factor and update ratios, QP is showing the best performance, with LP being second and RH in third place. This is consistent with our

observation in the WORM experiment that QP is best for inserts on high load factors and RH is typically the slowest. **As a conclusion, in a write-heavy workload, quadratic probing looks as the best option in general**.

# 7. ON TABLE LAYOUT: AOS OR SOA

One fundamental question in open-addressing is whether to organize the table as an *array-of-structs* (AoS) or as a *struct-of-arrays* (SoA). In AoS, the table is internally represented as array of key-value pairs whereas SoA keeps keys and values separated in two corresponding, aligned arrays. Both variants offer different performance characteristics and tradeoffs. These tradeoffs are somewhat similar to the difference between row and column layout for storing database tables. In general, we can expect to touch less cache-lines for AoS when the total displacement of the table is rather low, ideally just one cache line. In contrast to that, SoA already needs to touch at least two cache lines for each successful probe (one for the key and one for the value) in the best case. However, for high displacement (and hence longer probe sequences) SoA layout offers the benefit that we can just search through keys only, thus scanning up to only half the amount of data compared to AoS, where keys and values are interleaved. Another advantage of SoA over AoS is that a separation of keys from values makes vectorization with SIMD easy, essentially allowing us to load and compare four densely packed keys at a time on 256-bit SIMD registers as offered on current AVX-2 platforms. In contrast to that, comparing four keys in AoS with SIMD requires to first extract only the keys from the key-value pairs into the SIMD register, e.g., by using gather-scatter vector addressing which we found to be not very efficient on current processors. Independent of this, AoS also needs to touch up to two times more cache lines for long probe sequences compared to SoA when many keys are scanned.

In the following, we present a micro-benchmark to illustrate the effect of different layout and SIMD for inserts and lookups in linear probing. Since our computing server does not support AVX-2 instructions, we ran this micro-benchmark on a new MacBook Pro as described in Section 4. We implemented key comparisons with SIMD instructions for lookup and inserts on top of our existing linear probing hash tables by manually introducing intrinsics to our code. For example, in AoS, we load four keys at a time to a SIMD register from an cache-line-aligned index, using the `_mm256_load_si256` command. Then we perform a vectorized comparison on the four keys using `_mm256_cmpeq_epi64` and, in case of one successful comparison, obtain the first matching index with `_mm256_movemask_pd`.

We compare LPMult in AoS layout against LPMult in SoA layout with and without SIMD on a sparse data set. Similar to the indexing experiment of Section 5.2, we measure the throughput for insertions and lookups for load factors $50, 70, 90\%$. Due to the limited memory available on the laptop, we use the medium table capacity of $2^{27}$ slots — 2 GB. This still allows us to study the performance outside of caches, where we expect layout effects to matter most, because touching different cache lines typically triggers expensive cache misses. Figure 7 shows the results of the experiment.

**Discussion.** Let us start by discussing the impact of layout *without* using SIMD instructions, methods LPAoSMult and LPSoAMult in Figure 7. For inserts (Figure 7(a)), AoS performs up to $50\%$ better than SoA, on the lowest load factor ($50\%$). This gap is slowly closing with higher load factors, leaving AoS only 10% faster than SoA on load factor $90\%$. This result can be explained as follows. When collisions are rare (as on load factor 50), SoA touches two times more cache lines than AoS — it has to place key and value in different locations. In contrast to that, SoA can fit up to two times more keys in one cache line than AoS, which improves throughput for longer probes sequences when searching empty slots under high load factors. However, when beginning inserting into an empty hash table, we can often place the entry into its hash bucket without any further probing. Only over time we will require more and more probes. Thus, in the beginning, there is a high number of insertions where the advantage of AoS has higher impact. This is also the reason why the gap in insertion throughput between AoS and SoA significantly narrows as the load factor increases.

For lookups (Figures 7(b — d)) we noticed overall that AoS is faster than SoA on short probe sequences, i.e., especially for low load factors and low rates of unsuccessful queries. On the lowest load factor ($50\%$, Figure 7(b)), we can see that in the best case (all queries successful) AoS typically encounters half the number of cache misses compared to SoA, because keys and values are adjacent. This is reflected in a 63% higher throughput. With increasing unsuccessful lookup rate, the performance of SoA approaches AoS and the crossover point lies around 75% unsuccessful lookups. For 100% unsuccessful lookups, AoS improves over SoA by 15%. For load factor $70\%$ (Figure 7(c)), AoS is again superior to SoA for low rates of unsuccessful queries, but the crossover point at which SoA starts being beneficial shifted to $25\%$ unsuccessful queries instead of $75\%$ of the $50\%$ load factor. Interestingly, we can observe that for load factor $90\%$ (Figure 7(d)), the advantage of SoA over AoS layout is unexpectedly low — with the highest difference observed being around 30% instead of close to a factor 2 as we could expect. Our analysis obtained a combination of three different factors that explain this result. First, even in the extreme case of 100% unsuccessful lookups, the difference in touched caches lines is not a factor 2. The combination ⟨sparse distribution, Mult⟩ simulates the ideal case that every key is uniformly distributed over the hash table. Thus, we know [15] that the average number of probes in an unsuccessful search in linear probing is roughly $\frac{1}{2}\left(1+\left(\frac{1}{(1-\alpha)^2}\right)\right)$, where $\alpha$ is the load factor of the table. Thus, for $90\%$ load factor the average probe length is roughly $50.5$ (we could verify this experimentally as well). Now, in AoS we can pack four key-value pairs into a cache line, and twice as much (eight) for SoA. This means that the average number of loaded cache lines in AoS and SoA is roughly $\frac{50.5}{4}$ and $\frac{50.5}{8}$ respectively. However, *in practice* this behaves like $\lceil\frac{50.5}{4}\rceil = 13$ and $\lceil\frac{50.5}{8}\rceil = 7$ respectively — since whole cache lines are loaded. Which means that AoS loads *only* roughly $1.85\times$ more caches lines as SoA — which we were also able to verify experimentally. In addition to that, a second factor are non-uniform costs of visiting cache lines. We observed that the first probe in a sequence is typically more expensive than the subsequent linear probes because the first probe is likely to trigger a TLB miss and a page walk, which amortizes over visiting a larger amount of adjacent slots. The third factor is that, independent from the number of visited cache lines, the number of hash computations, loop iterations, and key comparisons are identical for SoA and AoS. Those parts of the probing algorithm involve data dependencies that build up a critical path in the pipeline, which is not easily hidden in the load latency by modern processors and compilers. **In conclusion, the ideal advantages of SoA over AoS are less strong in practice due to the way hardware works**.

We now proceed to discuss the impact of SIMD instructions in both layouts. In general, SIMD allows us to compare up to four 8-byte keys (or half a cache line) in parallel, with one instruction. However, this parallelism typically comes at a small price because loading keys into SIMD registers and generating a memory address from the result of SIMD comparison (e.g., by performing count-

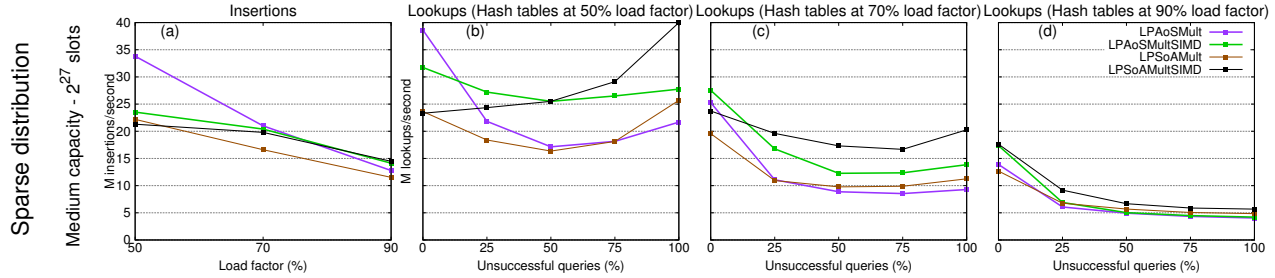


***Figure 7:*** *Effect of layout and SIMD in performance of LPMult at load factors* $50, 70, 90\%$ *w.r.t.* $2^{27}$ *under a sparse distribution of keys. Higher is better.*

trailing-zeros on a bit mask) potentially introduce a small overhead in terms of instructions. In case of writes that depend on address calculation based on the result of SIMD operations, we could even observe expensive pipeline stalls. Hence, in certain cases, SIMD can actually make execution slower, e.g., see Figure 7(a). For lower load factors, using SIMD for insertions can decrease performance significantly for both AoS and SoA layout, by up to $64\%$ in the extreme case. However, there is a crossover point between SIMD and non-SIMD insertions around $75\%$ load factor. We found that in such cases, SIMD is up to $12\%$ faster than non-SIMD.

For lookups, we can observe that SIMD improves performance in almost all cases. We notice that in general, the improvement of SIMD is higher for SoA than for AoS. As mentioned before, SoA layout simplifies loading keys to a SIMD register, whereas AoS requires us to gather the interleaved keys in a register. We observed that on the Haswell architecture, gathering is still a rather expensive operation and this difference gives SoA an edge over AoS for SIMD. As a result, we find SoA-SIMD superior to plain SoA in all cases for lookups, with improvement of up to up to $81\%$ (Figure 7(b)). We observed that AoS-SIMD can be up to $17\%$ harmful for low load factors, but beneficial for high load factors.

In general, we could observe in this experiment that AoS is significantly superior to SoA for insertions — even up to very high load factors. **Our overall conclusion is that AoS outperforms SoA by a larger margin than the other way around. Inside caches (not shown), both methods are comparable in terms of lookup performance, with AoS performing slightly better. When using SIMD, SoA has an edge over AoS — at least on current hardware — because keys are already densely packed**.

# 8. CONCLUSIONS AND FUTURE WORK

Due to the lack of space, we stated our conclusions in an inline fashion throughout the paper. All the knowledge we gathered leads us to propose a decision graph, Figure 8, that we hope can help practitioners to decide more easily what hash table to use in practice under different circumstances. Obviously, no experiment can be complete enough to fully capture the true nature of each hash table in *every* situation. Our suggestions are, nevertheless, educated as a result of our large set of experiments, and we are confident that they represent very well the behavior of the hash tables. We also hope that our study makes practitioners more aware about trade-offs and consequences of not carefully choosing a hash table.

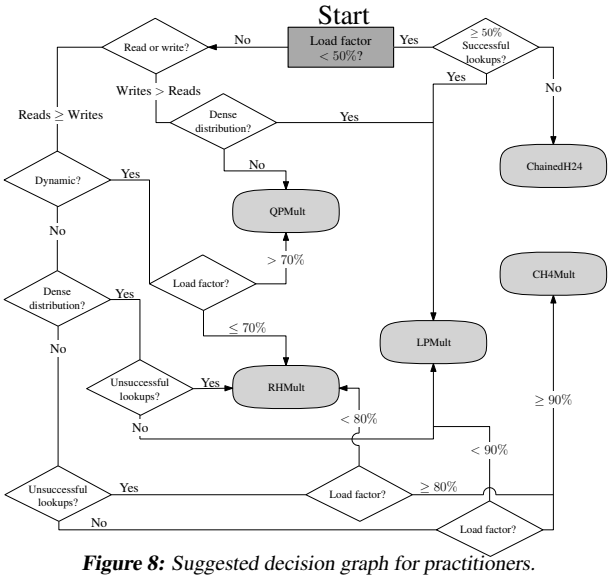


***Figure 8:*** *Suggested decision graph for practitioners.*

# 9. REFERENCES


[1] V. Alvarez, S. Richter, X. Chen, and J. Dittrich. A comparison of adaptive radix trees and hash tables. In *31st IEEE ICDE*, April 2015.

[2] A. Appleby. Murmurhash3 64-bit finalizer. Version 19/02/15. `https://code.google.com/p/smhasher/wiki/MurmurHash3`.

[3] C. Balkesen, J. Teubner, G. Alonso, and M. Ozsu. Main-memory hash joins on modern processor architectures. *IEEE TKDE*, 2014.

[4] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory-efficient hash joins. *VLDB*, 8(4), 2014.

[5] P. Celis. *Robin Hood Hashing*. PhD thesis, University of Waterloo, 1986.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 1990.

[7] M. Dietzfelbinger. Universal hashing and $k$-wise independent random variables via integer arithmetic without primes. In *STACS*, pages 569–580, 1996.

[8] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19 – 51, 1997.

[9] M. Dietzfelbinger and R. Pagh. *Succinct Data Structures for Retrieval and Approximate Membership*, volume 5125, pages 385–396. LNCS, 2008.

[10] M. Dietzfelbinger and U. Schellbach. On risks of using cuckoo hashing with simple universal hash classes. In *SODA*, pages 795–804, 2009.

[11] M. Drmota and R. Kutzelnigg. A precise analysis of cuckoo hashing. *ACM Trans. Algorithms*, 8(2):11:1–11:36, Apr. 2012.

[12] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Comp. Sys.*, 38(2):229–248, 2005.

[13] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: Fast architecture sensitive tree search on modern cpus and gpus. In *ACM SIGMOD*, pages 339–350, 2010.

[14] D. Knuth. Notes on "open" addressing. *Unpublished Memorandum*, 1963.

[15] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley, 1998.

[16] H. Lang, V. Leis, M.-C. Albutiu, T. Neumann, and A. Kemper. *Massively Parallel NUMA-Aware Hash Joins*, pages 3–14. LNCS, 2015.

[17] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *29th IEEE ICDE*, pages 38–49, April 2013.

[18] M. Mitzenmacher. *Some Open Questions Related to Cuckoo Hashing*, volume 5757, pages 1–10. LNCS, 2009.

[19] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[20] M. Pătraşcu and M. Thorup. The power of simple tabulation hashing. *J. ACM*, 59(3):14:1–14:50, June 2012.

[21] B. Schlegel, R. Gemulla, and W. Lehner. k-ary search on modern processors. In *DaMoN Workshop*, pages 52–60. ACM, 2009.

[22] M. Thorup. String hashing for linear probing. In *20th ACM-SIAM SODA*, pages 655–664, 2009.

[23] A. Viola. *Analysis of hashing algorithms and a new mathematical transform*. University of Waterloo, 1995.