A Comparison of Adaptive Radix Trees and Hash Tables

Victor Alvarez [#1], Stefan Richter [#2], Xiao Chen [#3], Jens Dittrich [#4]

 # Information Systems Group, Saarland University
[1] alvarez@cs.uni-saarland.de
[2] stefan.richter@infosys.uni-saarland.de
[3] s9xochen@stud.uni-saarland.de
[4] jens.dittrich@cs.uni-saarland.de

Abstract—**With prices of main memory constantly decreasing, people nowadays are more interested in performing their computations in main memory, and leave high I/O costs of traditional disk-based systems out of the equation. This change of paradigm, however, represents new challenges to the way data should be stored and indexed in main memory in order to be processed efficiently. Traditional data structures, like the venerable B-tree, were designed to work on disk-based systems, but they are no longer the way to go in main-memory systems, at least not in their original form, due to the poor cache utilization of the systems they run on. Because of this, in particular, during the last decade there has been a considerable amount of research on index data structures for main-memory systems. Among the most recent and most interesting data structures for main-memory systems there is the recently-proposed adaptive radix tree ARTful (ART for short). The authors of ART presented experiments that indicate that ART was clearly a better choice over other recent tree-based data structures like FAST and B⁺-trees. However, ART was not the first adaptive radix tree. To the best of our knowledge, the first was the Judy Array (Judy for short), and a comparison between ART and Judy was not shown. Moreover, the same set of experiments indicated that *only* a hash table was competitive to ART. The hash table used by the authors of ART in their study was a chained hash table, but this kind of hash tables can be suboptimal in terms of space and performance due to their potentially high use of pointers. In this paper we present a thorough experimental comparison between ART, Judy, two variants of hashing via quadratic probing, and three variants of Cuckoo hashing. These hashing schemes are known to be very efficient. For our study we consider whether the data structures are to be used as a non-covering index (relying on an additional store), or as a covering index (covering key-value pairs). We consider both OLAP and OLTP scenarios. Our experiments strongly indicate that neither ART nor Judy are competitive to the aforementioned hashing schemes in terms of performance, and, in the case of ART, sometimes not even in terms of space.**

I. INTRODUCTION

In the last decade the amount of main memory in commodity servers has constantly increased — nowadays, servers with terabytes of main memory are widely available at affordable prices. This memory capacity makes it possible to store most databases completely in main memory, and has triggered a considerable amount of research and development in the area. As a result, new high performance index structures for main memory databases are emerging to challenge hash tables — which have been widely used for decades due to their good performance. A recent and promising structure in this domain is the adaptive radix tree **ARTful** [1], which we call just **ART**

from now on. This recent data structure was reported to be significantly faster than existing data structures like FAST [2] and the cache-conscious B⁺-tree CSB⁺ [3]. Moreover, it was also reported that only a hash table is competitive to ART. Thus, ART was reported to be as good as a hash table while also supporting range queries. Nonetheless, three important details were not considered during the experimental comparison of ART with other data structures that we would like to point out: (1) To the best of our knowledge, the first adaptive radix tree in the literature was the **Judy Array** [4], which we simply call **Judy** from now on. A comparison between ART and Judy was not offered by the original study [1], but given the strong similarities between the two structures, we think that there ought to be a comparison between the two. (2) The hash table used by the authors of ART for the experimental comparison was a chained hash table. This kind of hashing became popular for being, perhaps, the very first iteration of hashing, appearing back in the 50s. Nevertheless, it is still popular for being the default method in standard libraries of popular programming languages like C++ and Java. However, nowadays chained hashing could be considered suboptimal in performance and space because of the (potentially) high overhead due to pointers, and other hashing schemes are preferred where performance is sought — like quadratic probing [5], [6]. Moreover, rather recently, Cuckoo hashing [7] has seen a considerable amount of research [8], and it has been reported to be competitive [7] in practice to, for example, quadratic probing. Thus, we believe that the experimental comparison between ART and hashing was not complete. This brings us to our last point, hash functions. (3) Choosing a hash function should be considered as important as choosing a hashing scheme (table), since it highly determines the performance of the data structure. Over decades there has been a considerable amount of research focusing *only* on hash functions — sometimes on their theoretical guarantees, some other times on their performance in practice. The authors of ART chose Murmur [9] as a hash function — presumably due to the robustness (ability of shuffling data) shown in practice, although nothing is known about its theoretical guarantees, to the best of our knowledge. In our own experiments we noticed that Murmur hashing is indeed rather robust, but for many applications, or at least the ones considered by the authors of ART, that much robustness could be seen as an overkill. Thus, it is interesting to see how much an easier (but still good) hash function changes the picture.

A. Our contribution

The main goal of our work is to *extend* the experimental comparison offered by the authors of ART by providing a thorough experimental evaluation of ART against Judy, two variants of quadratic probing, and three variants of Cuckoo hashing. We provide different variants of the same hashing scheme because some variants are tuned for performance, while other are tuned for space efficiency. However, it is *not* our intention to compare ART against structures already considered (covered) in the original ART paper [1] *again*. Consequently, just as in the micro-benchmarks presented in [1], we only focus on keys from an integer domain. In this regard, we would like to point out that the story could change if keys were arbitrary strings of variable size. However, a thorough study on indexing strings in main memory deserves a paper on its own, and is thus out of scope of this work.

For each considered hash table we test two different hash functions, Murmur hashing [9], for reference, completeness, and compatibility with the original study [1], and the well-known multiplicative hashing [5], [6], [10] — which is perhaps the easiest-to-compute hash function with still good theoretical guarantees. Our experiments strongly indicate that neither ART nor Judy are competitive in terms of performance to well-engineered hash tables, and in the case of ART, sometimes not even in terms of space. For example, for one billion indexed keys, *one* non-covering variant of Cuckoo hashing is *at least* $4.8\times$ faster for insertions than ART, *at least* $2.8\times$ faster for lookups, and it sometimes requires just half the space of ART, see Figures 2, 3, and 4. We also hope to convey more awareness as of how important it is to consider newer hashing approaches (hashing schemes *and* hash functions) when throughput performance and/or memory efficiency are crucial.

The remainder of the paper is organized as follows. In Section II we give a general description of adaptive radix trees — highlighting key similarities and differences between ART and Judy. In Section III we give a detailed description of the hashing schemes and hash functions used in our study. In IV we present our experiments. Finally, in Section V we close the paper with our conclusions. Our presentation is given in a self-contained manner.

II. RADIX TREES

In this section we give a general description of the (adaptive) radix trees included in our study. In general, a radix tree [6] (also called prefix tree, or trie) is a data structure to represent ordered associative arrays. In contrast to many other commonly used tree data structures such as binary search trees or standard B-Trees, nodes in radix trees do not cover complete keys; instead, nodes in a radix tree represent partial keys, and only the full path from the root to a leaf describes the complete key corresponding to a value. Furthermore, operations on radix trees do not perform comparisons on the keys in the nodes but rather, operations like looking up for a key work as follows: (1) Starting from the root, and for each inner node, a partial key is extracted on each level. (2) This partial key determines the branch that leads to the next child node. (3) The process repeats until a leaf or an empty branch is reached. In the first case, the key is found in the tree, in the second case, it is not.

In a radix tree, the length of the partial keys determines the fan-out of the nodes because *for each* node there is *exactly one* branch *for each* possible partial key. For example, let us assume a radix tree that maps 32-bit integer keys to values of the same type. If we chose each level to represent a partial key of one byte, this results in a 4-level radix tree having a fan-out of 256 branches per node. Notice that for all levels, all keys under a certain branch have a common prefix and unpopulated branches can be omitted. For efficiency, nodes in a radix tree are traditionally implemented as arrays of pointers to child nodes; when interpreting the partial key as an index to the array of child pointers, finding the right branch on a node is as efficient as one array access. However, this representation can easily lead to excessive memory consumption and bad cache utilization for data distributions that lead to many sparsely populated branches, such as uniform random distribution. In the context of our example, each node would contain an array of 256 pointers, even if only a single child node exists; leading to high memory overhead. This is the reason why radix trees have usually been considered as a data structure that is only suitable for certain use cases, e.g., textual data, and not for general purposes. For example, radix trees are often used for prefix search on skewed data; like in dictionaries. Still, radix trees have many interesting and useful properties: (1) Shape depends *only* on the key space and length of partial keys, but not on the contained keys or their insertion order. (2) Do not require rebalancing operations. (3) Establish an order on the keys and allow for efficient prefix lookups. (4) Allow for prefix compression on keys.

The aforementioned memory overheads that traditional radix trees potentially suffer from leads to the natural question of whether the situation can be somehow alleviated. To the best of our knowledge, the **Judy Array** [4] is the first variant of a radix tree that adaptively varies its node representation depending on the key distribution and/or cardinality of the contained data. Judy realizes adaptivity by introducing several compression techniques. These techniques prevent excessive memory footprints on sparsely populated trees, and improve cache utilization. According to the inventors, Judy offers performance similar to hash maps, supports efficient range queries like a (comparison-based) tree structures, and prefix queries like traditional radix trees. All this while also providing better memory efficiency than *all* aforementioned data structures.

Very recently, in 2013, the **ARTful** index [1] was introduced as a new index structure for main memory database systems. ART is also an adaptive radix tree, and has similar purposes as Judy — high performance at low memory cost. However, unlike Judy, ART was not designed as an associative array, but rather ART is tailored towards the use case of an index structure for a database system — on top of a main memory storage. In the following we will discuss both, Judy arrays and ART, highlighting their similarities and differences.

A. Judy

Judy can be characterized as a variant of a 256-way radix tree. There are three different types of Judy arrays: (1) **Judy1**: A bit array that maps integer keys to true or false and hence can be used as a set. (2) **JudyL**: An array that maps integer keys to integer values (or pointers) and hence can be used as an integer to integer map. (3) **JudySL**: An array that maps string keys of
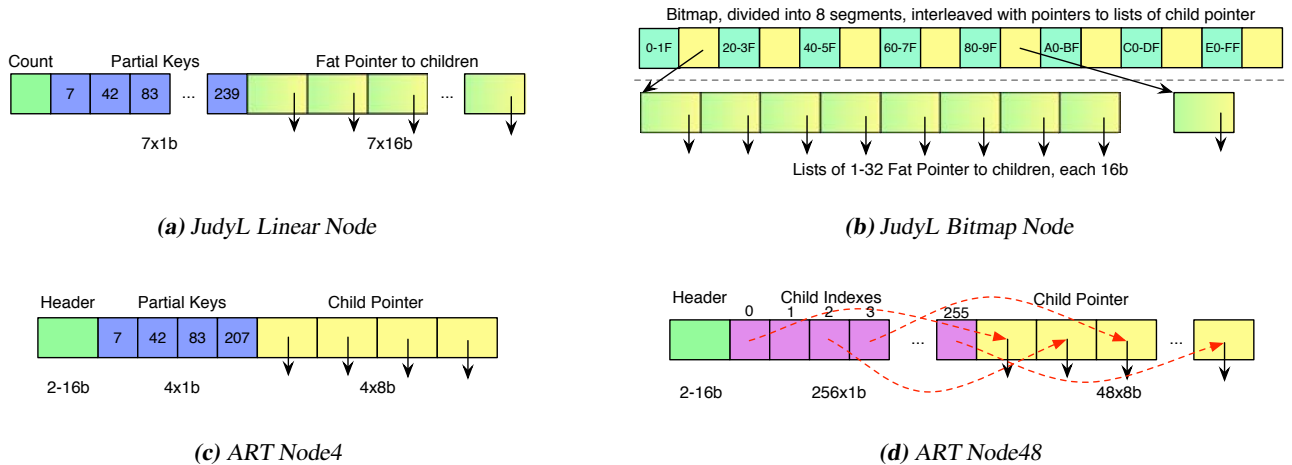
(a) *JudyL Linear Node*

(b) *JudyL Bitmap Node*

(c) *ART Node4*

(d) *ART Node48*

Figure 1: *Comparison of node types (64-bit).*

arbitrary length to integer values (or pointers) and hence can be used as a map from byte sequences to integers.

For a meaningful comparison with the other data structures considered by us, we will only focus on **JudyL** for the remainder of this work, and thus consider Judy and JudyL as synonyms from now on. In the following, we give a brief overview of the most important design decisions that contribute to the performance and memory footprint of JudyL.

The authors of Judy observed that cache misses have a tremendous impact on the performance of any data structure, up to the point where cache miss costs dominate the runtime. Hence, to offer high performance across different data distributions, one major design concern of Judy was to avoid cache-line fills (which can result in cache misses) at almost any cost. Observe that the maximum number of cache-line fills in a radix tree is determined by the number of tree levels. Moreover, the maximum number of tree levels is determined by the maximum key length divided by the partial key size. For every tree level in a standard radix tree, we need to access exactly one cache line that contains the pointer to the child node under the index of that partial key.

Judy addresses memory overheads of traditional radix trees under sparse data distributions and simultaneously avoids cache-line fills through a combination of more than 20 different compression techniques. We can roughly divide these techniques into two categories: *horizontal compression* and *vertical compression*. Due to space constraints, we only give a brief overview of the most important ideas in Judy. A full description of the ideas can be found in [4].

Horizontal compression. Here the problem of many large, but sparsely populated nodes, is addressed. The solution offered by Judy is to adapt node sizes dynamically and individually with respect to the actual population of the subtree underneath each node. Hence, Judy can compress unused branches out of nodes. For example, Judy may use smaller node types that have e.g., only seven children. However, in contrast to uncompressed (traditional) radix nodes with 256 branches, the slots in compressed nodes are not directly

addressable through the index represented by the current partial key. Consequently, compressed nodes need different access methods, such as comparisons, which can potentially lead to multiple additional cache-line fills. Judy minimizes such effects through clever design of the compressed nodes. There are two basic types of horizontally compressed nodes in Judy: linear nodes and bitmap nodes, which we briefly explain: (1) A **linear node** is a space efficient implementation for nodes with very small number of children. In Judy, the size of linear nodes is limited to one cache line.[1] Linear nodes start with a sorted list that contains only the partial keys for branches to existing child nodes. This list is then followed by a list of the corresponding pointers to child nodes in the same order, see Figure 1a. To find the child node under a partial key, we search the partial key in the list of partial keys and follow the corresponding child pointer if the partial key is contained in the list. Hence, linear nodes are similar to the nodes in a B-tree w.r.t. structure and function. (2) A **bitmap node** is a compressed node that uses a bitmap of 256 bits to mark the present child nodes. This bitmap is divided into eight 32-bit segments, interleaved with pointers to the corresponding lists of child pointers, see Figure 1b. Hence, bitmap nodes are the only structure in Judy that involve up to two cache-line fills. To lookup the child under a partial key, we first detect if the bit for the partial key is set. In that case, we count the leading set bits in the partial bitmap to determine the index of the child pointer in the pointer list. Bitmap nodes are converted to uncompressed nodes (256 pointers) as soon as the population reaches a point where the additional memory usage amortizes.[2]

To differentiate between node types, Judy must keep some meta information about every node. In contrast to most other data structures, Judy does not put meta information in the header of each node, because this can potentially lead to one additional cache-line fill per access. Instead, Judy use what the authors of Judy call *Judy pointers*. These pointers are

---

[1] Judy's 10-year-old design assumes cache-line size of 16 machine words, which is not the case for modern main-stream architectures.

[2] The concrete conversion policies between nodes types are out of the scope of this work.

fat pointers of two machine words size (i.e., 128bit on 64bit architectures) that combine the address of a node with the corresponding meta data, such as: node type, population count, and key prefix. Judy pointers avoid additional cache-line fills by densely packing pointers with the meta information about the object they point to.

Vertical compression. In Judy arrays vertical compression is mainly achieved by skipping levels in the tree when an inner node has only one child. In such cases, the key prefix corresponding to the missing nodes is stored as decoding information in the Judy pointer. This kind of vertical compression is commonly known in the literature as **path compression**. Yet another technique for vertical compression is **immediate indexing**. With immediate indexing, Judy can store values immediately inside of Judy pointers instead of introducing a whole path to a leaf when there is no need to further distinguish between keys.

B. ART

This newer data structure shares many ideas and design principles with Judy. In fact, ART is also a 256-radix tree that uses (1) different node types for horizontal compression, and (2) vertical compression also via path compression and immediate indexing — called lazy expansion in the ART paper. However, there are two major differences between ART and Judy: (1) There exist four different node types in ART in contrast to three types in Judy. These node types in ART are labeled with respect to the maximum amount of children they can have: `Node4`, `Node16`, `Node48`, and the uncompressed `Node256`. Those nodes are also organized slightly different than the nodes in Judy. For example, the meta information of each node is stored in a header instead of a fat pointer (Judy pointer). Furthermore, ART nodes take into account the latest changes and features in hardware design, such as SIMD instructions to speedup searching in the linearly-organized `Node16`. It is worth pointing out that we can not find any consideration of that kind of instructions in the decade-old design of Judy. (2) ART was designed as an index structure for a database, whereas Judy was designed as a general purpose associative array. As a consequence, Judy owns its keys and values and covers them both inside the structure. In contrast to that, ART does not necessarily cover full keys or values (e.g., when applying vertical compression) but rather stores a pointer (as value) to the primary storage structure provided by the database — thus ART is primarily used as a non-covering index. At lookup time, we use a given key to lookup for the corresponding pointer to the database store containing the complete ⟨key, value⟩ pair.

Finally, and for completeness, let us give a more detailed comparison of the different node types between Judy and ART. `Node4` and `Node16` of ART are very much comparable to a linear node in Judy except for their sizes, see Figures 1a and 1c. `Node16` is just like a `Node4` but with 16 entries. Uncompressed `Node256` of ART is the same as the uncompressed node in Judy, and thus also as in plain radix trees. `Node48` of ART consists of a 256-byte array (which allows direct addressing by a partial key) follow by an array of 48 child pointers Up to 48 locations of the 256-byte array can be occupied, and each occupied entry stores the index in the child pointer array holding the corresponding pointer for the partial

key, see Figure 1d. `Node48` of ART and the bitmap node of Judy fill in the gap between small and large nodes.

III. HASH TABLES

In this section we elaborate on the hashing schemes and the hash functions we use in our study. In short, the hashing schemes are (1) the well-known quadratic probing [6], [5], and (2) Cuckoo hashing [7]. As for hash functions we use 64-bit versions of (1) Murmur hashing [9], which is the hash function used for the original study [1], and (2) the well-known, and somewhat part of the hashing folklore, multiplicative hashing [5], [6], [10]. In the rest of this section we consider each of these parts in turn.

A. Quadratic probing

Quadratic probing is one of the best-known open-addressing schemes for hashing. In open-addressing, every hashed element is contained in the hash table itself, i.e., every table entry contains either an element or a special character denoting that the corresponding location is unoccupied. The hash function in quadratic probing is of the following form:

$$h(x,i) = (h'(x) + c_1 \cdot i + c_2 \cdot i^2)$$

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.

What makes quadratic probing attractive and popular is: (1) It is easy to implement. In its simplest iteration, the hash table consists of a single array only. (2) In the particular case that the size of the hash table is a power of two, it can be proven that quadratic probing will examine *every* single location of the table in the worst case [5]. That is, as long as there are available slots in the hash table, this particular version of quadratic probing will *always* find them, at the expense of an increasing number of probes.

Quadratic probing is, however, not bulletproof. It is known that it could suffer from *secondary clustering*. This means that if two different keys collide in the very first probe, they will also collide in all sub-sequent probes. Thus, choosing a good hash function is of primary concern.

The implementations of quadratic probing used in this study are the ones provided by Google dense and sparse hashes [11]. These `C++` implementations are well-engineered for general purposes[3], and are readily available. Furthermore, they are designed to be used as direct replacements of `std::unordered_map`[4]. This reduces integration in existing code to the minimal effort. These Google hashes come in two variants, dense and sparse. The former is optimized for (raw) performance, potentially sacrificing space, while the latter is optimized for space while potentially sacrificing performance. In this study we consider both variants, and, for simplicity, we will refer to Google dense and sparse hashes simply as **GHFast** (for performance) and **GHMem** (for memory efficiency) respectively.

---

[3]This does not necessarily imply optimal performance in certain domains. That is, it is plausible that specialized implementations could be faster.

[4]Whose implementation happens to be hashing with chaining just as the ones used in the original ART paper.

B. Cuckoo hashing

Cuckoo hashing is a relatively new open-addressing scheme [7], and somewhat still not well-known. The original (and simplest) version of Cuckoo hashing works as follows: There are two hash tables T_0, T_1, each one having its own hash function h_0, h_1. Every inserted element x is stored at either $T_0[h_0(x)]$ or $T_1[h_1(x)]$ but *never* in both. When inserting an element x, location $T_0[h_0(x)]$ is first probed, if the location is empty, x is store there, otherwise, x kicks out the element y already found at that location, x is stored there, but now y is out of the table and has to be inserted, so location $T_1[h_1(y)]$ is probed. If this location is free, y is stored there, otherwise y 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. How this rehash is done is a design decision: it is not necessary to allocate new tables, one can reuse the already allocated space by deleting and reinserting every element already found in the table. However, if the set of elements to be contained in the table increases over time, then perhaps increasing the size of the table when the rehash happens is a better policy for future operations. It has been empirically observed [7], [12] that in order to work, and obtain good performance, the load factor of Cuckoo hashing should stay slightly below 50%. That is, it requires at least twice as much space as the cardinality of the set to be indexed. Nevertheless, it has also been observed [12] that this situation can be alleviated by generalizing Cuckoo hashing 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%, at the expense of performance. Thus, as for the Google hashes mentioned before, we can consider two versions of Cuckoo hashing, one tuned for performance, when $k = 2$, and the other tuned for space-efficiency, when $k = 4$.

Finally, we include in this study yet another variant of Cuckoo hashing. This variant allows more than one element per location in the hash table [13], as opposed to the original Cuckoo hashing where *every* location of the hash table holds *exactly* one element. In this other variant, we use only two tables T_0, T_1, just as the original Cuckoo hashing, but every location of the hash table is a bucket of size equal to the cache-line size, 64 bytes for our machines. This variant works essentially as the original Cuckoo hashing, when inserting an element x, it checks whether there is a free slot in the corresponding bucket, if yes, then x is inserted, otherwise a random element y of that bucket is kicked out, x is left in its place, and we start the Cuckoo cycles. We decided to include this variant of Cuckoo hashing because when a location of the hash table is accessed, this location is accessed through a cache line, so by aligning these buckets to cache lines boundaries we hope to have better data locality for lookups; at the expense of making more comparisons to find the given element in the bucket. This comparisons happen, nevertheless, only among elements that are already on cache (close to the processor).

For simplicity, we will refer to standard Cuckoo hashing using two and four tables as **CHFast** (for performance) and **CHMem** (for memory efficiency) — highlighting similarities of each of these hashes with Google's GHFast and GHMem, respectively, mentioned before. The last variant of Cuckoo hashing described above, using 64-byte buckets, will be simply referred to as **CHBucket**.

Let us now explain the reasons behind our decision to include Cuckoo hashing in our study. (1) For lookups, traditional Cuckoo hashing requires at most two tables accesses, which is in general optimal among hashing schemes using linear space. In particular, it is independent of the current load factor of the hash table — unlike other open-addressing schemes, like quadratic probing. (2) It has been reported to be competitive with other good hashing schemes, like quadratic probing or double hashing [7], and (3) It is easy to implement.

Like quadratic probing, Cuckoo hashing is not bulletproof either. It has been observed [7] that Cuckoo hashing is sensitive to what hash functions are used [14]. With good (and robust) hash functions, the performance of Cuckoo hashing is good, but with hash functions that are not as robust, performance deteriorates; we will see this effect in our experiments.

C. Hash functions

Having explained the hashing schemes used in our study, we now turn our attention to the hash functions used. We pointed out before that both used hashing schemes are highly dependent on the hash functions used. For our study we have decided to include two different hash functions: (1) MurmurHash64A [9] and the well-known multiplicative hashing [6]. The first one has been reported to be efficient and robust[5] [9], but more importantly, it is included here because it is the hash function that was used in the original ART paper [1], and we wanted to make our study equivalent.

The second hash function, multiplicative hashing, is *very* well known [5], [6], [10], and it is given here:

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

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: $a \text{ div } b = \lfloor a/b \rfloor$. What makes this hash function highly interesting is: (1) It can be implemented extremely efficiently by observing that the multiplication $x \cdot z$ is per se already done modulo 2^w, and the operator div is equivalent to a right bit shift by $w - d$ positions. (2) It has also theoretical guarantees. It has been proven [10] 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:

$$Pr[h_z(x) = h_z(y)] \leq \frac{2}{2^d} = \frac{1}{2^{d-1}}$$

This probability is twice as large as the ideal probability that, for a hash function, *every* location of the hash table is equally likely. 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 perfect candidate for simple and somewhat robust hash functions.

---

[5]Although, to the best of our knowledge, no theoretical guarantee of this has been shown.

As we will see in our experiments, MurmurHash64A is indeed more robust than multiplicative hashing, but this robustness comes at a very high performance degradation. In our opinion multiplicative hashing showed to be robust enough in all our scenarios. From now on, and for simplicity, we will refer to MurmurHash64A simply as **Murmur** and to multiplicative hashing just as **Simple**.

IV. Main Experiments

In this section we experimentally confront the adaptive radix tree ART [1] with all other structures previously mentioned: (1) Judy [4], which is another kind of adaptive radix tree highly space-efficient — discussed in Section II and (2) Quadratic probing [11] and Cuckoo hashing [7] — discussed in Section III. The experiments are mainly divided into three parts.

In IV-C we first show experiments comparing ART only against Cuckoo hashing under the following metrics: **insertion throughput**, **point query throughput**, and **memory footprint**. The reason why we only compare ART against Cuckoo hashing is the following: ART, as presented and implemented in [1] was designed as a non-covering indexing data structure for databases. That is, as mentioned in Section II-B, ART will index a set of ⟨key, value⟩ pairs already stored and provided by a database. Thus, ART will, in general, neither cover the key nor the value[6], but it will rather use the key to place a pointer to the location in the database where the corresponding pair is stored. Thus, when looking up for a given key, ART will find the corresponding pointer (if previously inserted) and then follow it to the database store to retrieve the corresponding ⟨key, value⟩ pair. The semantics of the freely available implementations of Judy arrays [4] and Google hashes [11] are that of a map container (associative array), i.e., self-contained general-purpose indexing data structures (covering both the key and the value). We could have compared ART against these implementations as well but we think the comparison is slightly unfair, since inserting a pointer in those implementations will still cover the key, and thus the data structure will per se require more space. This is where our own implementation of Cuckoo hashing enters the picture. For the experiments presented in IV-C, Cuckoo hashing uses the key to insert a pointer to the database store, exactly just as ART — making an apple-to-apple comparison. For these experiments we assume that we *only* know upfront the number n of elements to be indexed. This is a valid assumption since we are interested in indexing a set of elements already found in a database. With this in mind, the hash tables are prepared to be able to contain at least n elements. Observe that the ability of pre-allocate towards certain size is (trivially) inherent to hash tables. In contrast, trees require knowledge not only about their potential sizes, but also the actual values and dedicated (bulk-loading) algorithms. This kind of workload (IV-C) can be considered static, like in an OLAP scenario.

In IV-D we test the structures considered in IV-C under TPC-C-like dynamic workloads by mixing insertions, deletions, and point queries. This way we simulate an OLTP scenario. The metric here is only **operation throughput**. In

this experiment the data structures assume nothing about the amount of elements to be inserted or deleted, and thus we will be able to observe how the structures perform under fully dynamic workloads. In particular, we will observe how the hash tables handle growth (rehashing) over time.

In IV-E we consider ART as a standalone data structure, i.e., a data structure used to store (cover) ⟨key, value⟩ pairs, and we compare it this time against Judy array, Google hashes, and Cuckoo hashing under the same metrics as before. As ART was not originally designed for this purpose, we can go about two different ways: (1) We endow ART with its own store and we use the original implementation of ART, or (2) We endow ART with explicit leaf nodes to store the ⟨key, value⟩ pairs. We actually implemented both solutions but we decided to keep for this study only the first one. The reason for this is that for the second option we observed mild slowdowns for insertions and mild speedups for lookups (just as expected), but space consumption increases significantly as the size of the set to be contained also increases; the reason for this is that a leaf node requires more information (the header) than simply storing only ⟨key, value⟩ pairs in a pre-allocated array. For these experiments, the hash tables and the store of ART are prepared to be able to store at least n elements, where n is the number of elements to be indexed.

A. Experimental setup

All experiments are **single-threaded**. The implementations of ART, Judy arrays, and Google hashes are the ones freely available [15], [4], [11]. No algorithmic detail of those data structures was touched except that we implemented the missing range-query support in ART. All implementations of Cuckoo hashing are our own. All experiments are in **main memory** using a **single core** (one NUMA region) of a dual-socket machine having two hexacore Intel Xeon Processors X5690 running at 3.47 GHz. The L1 and L2 cache sizes are 64 and 256 KB respectively per core. The L3 cache is shared and has a size of 12 MB. The machine has a total of 192 GB of RAM running at 1066 MHz. The OS is a 64-bit Linux (3.4.63) with the default page size of 4 KB. All programs are implemented in C/C++ and compiled with the Intel compiler icc-14 with optimization -O3.

B. Specifics of our workloads

In our experiments we include two variants of ART, let us call them unoptimized and optimized. The difference between the two of them is that the latter applies path compression (one of the techniques for vertical compression mentioned in Section II-B) to the nodes. By making (some) paths in the tree shorter, there is hope that this will decrease space and also speedup lookups. However, path compression clearly incurs into more overheads at insertion time; since at that time it has to be checked whether there is opportunity for compression and then it must be performed. In our experiments we denote the version of ART with path compression by **ART-PC**, and the one without it simply by **ART**. From now on, when we make remarks about ART, those remarks apply to *both* variants of ART, unless we say otherwise and point out the variant of ART we are referring to.

---

[6]The only exception to this happens when the key equals the value; effectively making ART a set container.

In the original ART paper [1] *all* micro-benchmarks are performed on 32-bit integer keys because some of the structures therein tested are 32-bit only. The authors also pointed out that for such short keys, path compression increases space instead of reducing it, and thus they left path compression out of their study. In our study we have no architectural restrictions since *all* herein tested structures support 32- and 64-bit integer keys. Due to the lack of space, and in order to see the effect of path compression, we have decided to (only) present **64-bit** integer keys.

We perform the experiments of IV-C and IV-E on two different key distributions on three different dataset sizes — for a total of six datasets. The two key distributions considered are the ones also considered in the original paper [1] and these are: (1) **Sparse** distribution; where *each* indexed key is *unique* and chosen uniformly at random from $[1, 2^{64})$, and (2) **Dense** distribution; where *every* key in $1, \ldots, n$ is indexed[7] (n is the total number of elements to be indexed by the data structures). As for datasets, for each of the aforementioned distributions we considered three different sizes: **16**, **256**, and **1000 million**. Two out of these three datasets (16M, 256M) were also considered in the original ART paper, along with a size of 65K. We would like to point out that 65K pairs of 16 bytes each is rather small and fits comfortably in the L3 cache of a modern machine. For such a small size whether an index structure is needed is debatable. Thus, we decided to move towards "big" datasets, and include the one billion size instead. Finally, the shown performance is the average of three independent measurements; for the sparse distribution of keys each measurement has a different input set.

C. Non-covering evaluation

In this very first set of experiments we test ART against Cuckoo hashing under the workload explained in IV-B. Lookups are point queries and each one of them looks up for an existing key. After having inserted all keys, the set of keys used for insertions is permuted uniformly at random, and then the keys are looked up in this random order; this guarantees that insertions and lookups are independent from each other. Insertion and lookup performance can be seen in Figures 2 and 3 respectively; each is presented in millions of operations per second. In Figure 4 we present the effective memory footprint of each structure in megabytes; this size accounts *only* for the size of the data structure, i.e., everything except the store.

Before analyzing the results of our experiments, let us state beforehand our conclusion. The adaptive radix tree ART was originally reported [1] to have better performance than other well-engineered tree structures (of both kinds, comparison-based and radix trees). It was also reported that only hashes were competitive to ART. Our own experience indicates that well-engineered performance-based hash tables are not only competitive to ART, but actually significantly better. For example, CHFast-Simple is *at least* $2\times$ faster for insertions and lookups than ART throughout the experiments. Moreover, this difference gets only worse for ART as the size of the set to be indexed increases; for one billion CHFast-Simple is *at least* $4.8\times$ faster than ART for insertions and *at least* $2.8\times$ faster

---

7Dense keys are randomly shuffled before insertion.

for lookups, and CHBucket-Simple is *at least* $4\times$ faster than ART for insertions, and *at least* $2\times$ faster for lookups.

Having stated our conclusion, let us now dig more into the data obtained by the experiments. First of all (1) we can observe that using a simple, but still good hash function, has in practice an enormous advantage over robust but complicated hash functions; CHFast-Simple is throughout the experiments roughly $1.7\times$ faster for insertions than CHFast-Murmur, and also roughly $1.93\times$ faster for lookups. This difference in performance is intuitively clear, but quantifying and seeing the effect makes an impression stronger than initially expected. (2) With respect to memory consumption, see Figure 4, multiplicative hashing seems to be robust enough for the two used distributions of keys (dense and sparse). In all but one tested case, see Figure 4a, multiplicative hashing uses as much space as Murmur hashing — which has been used in the past for its robustness. The discrepancy in the robustness of both hash functions suggests that a dense distribution pushes multiplicative hashing to its limits, and this has been pointed out before [14]. In our opinion, however, multiplicative hashing remains as a strong candidate to be used in practice. Also, and perhaps more important, it is interesting to see that the memory consumption of either version of ART is competitive only under the dense distribution of keys, although not better than that of CHMem. This is where the adaptivity of ART plays a significant role; as in contrast to the sparse distribution, where ART seems very wasteful w.r.t. memory consumption. (3) 64-bit integer keys are (again) still too short to notice the positive effect of path compression in ART — both versions of ART have essentially the same performance, but the version without path compression is in general more space-efficient; the same effect was also reported in the original ART paper [1]. (4) With respect to performance (insertions and lookups) we can see that the performance of all structures degrades as the size of the index increases. This is due to caching effects (data and TLB misses) and it is expected, as it was also observed in the original ART paper [1]. When analyzing lookup performance, we go into more detail on these caching effects. We can also observe that as the size of the index increases, the space-efficient variant of Cuckoo hashing, CHMem-Simple, gains territory to ART. Thus, a strong argument in favor of CHMem-Simple is that it has similar performance to ART but it is in general more space-efficient.

Let us now try to understand the performance of the data structures better. Due to the lack of space we will only analyze lookups on two out of three datasets, and comparing the variant of ART without path compression against the two fastest hash tables (CHFast-Simple and CHBucket-Simple).

Lookup performance. Tables I and II show a basic cost breakdown *per lookup* for 16M and 256M respectively. From these tables we can deduce that the limiting factor in the (lookup) performance of ART is a combination of long latency instructions plus the complexity of the lookup procedure. For the first term (long latency instructions) we can observe that the sum of L3 Hits + L3 Misses is considerably larger than the corresponding sum of CHFast-Simple and CHBucket-Simple. The L3-cache-hit term is essentially non-existent for the hashes, which is clear, and the L3-cache-miss term of ART rapidly exceeds that of the hashes as the index size increases. This makes perfect sense since ART decomposes a key into bytes
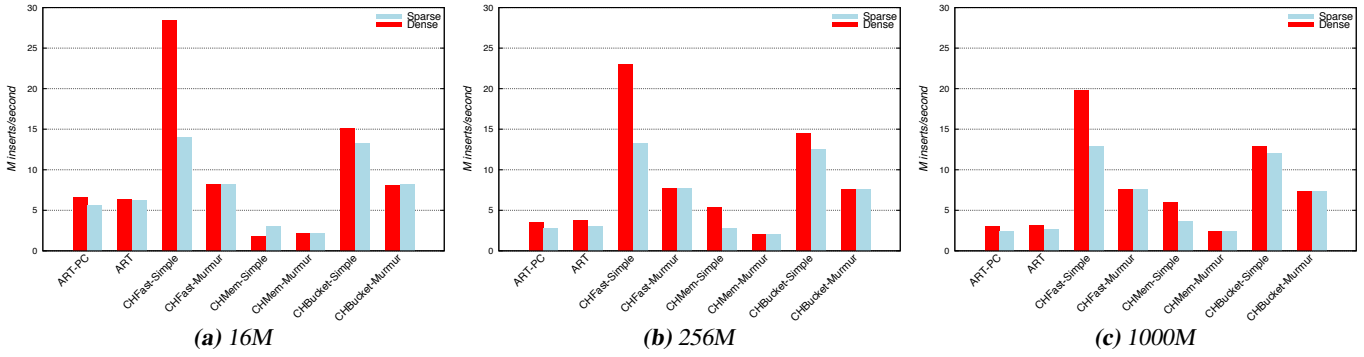
Figure 2: Insertion throughput (non-covering). Higher is better.
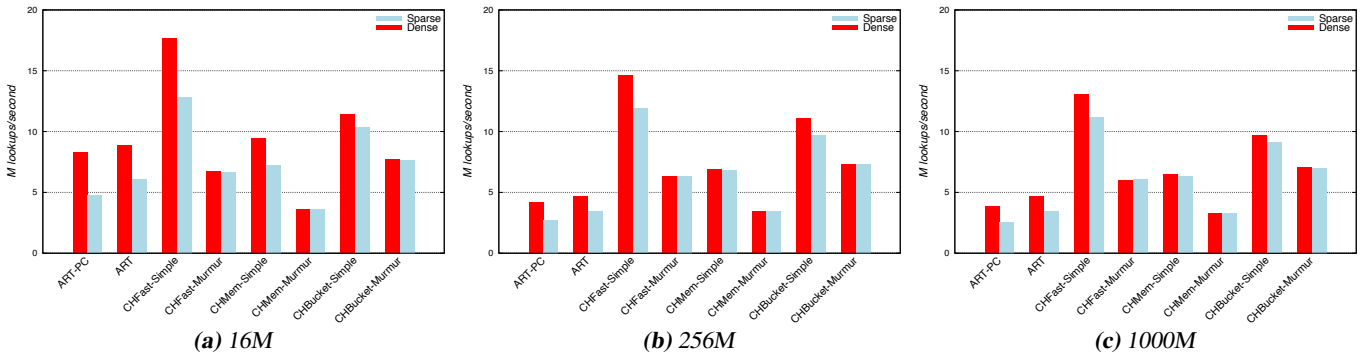

Figure 3: Lookup throughput (non-covering). Higher is better.
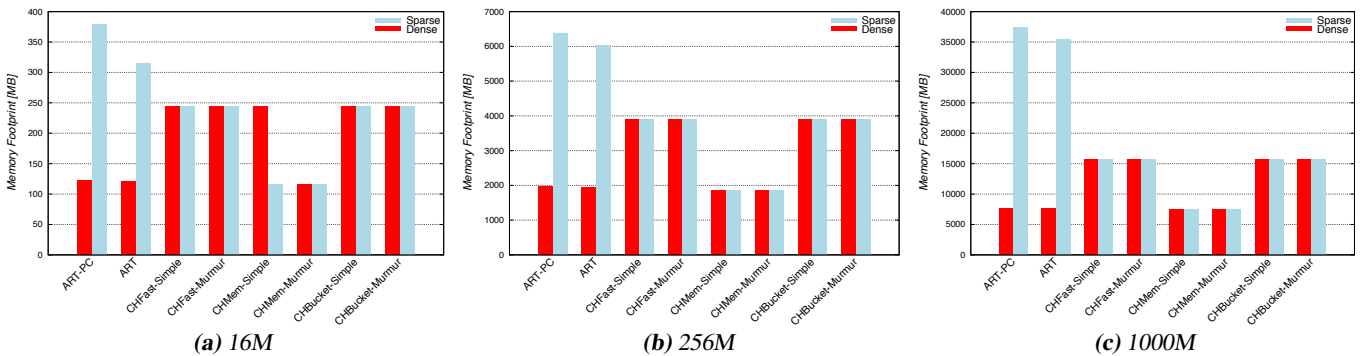

Figure 4: Memory footprint in MB (non-covering). Lower is better.

and then uses each byte to traverse the tree. In the extreme (worst) case, this traversal incurs into at least as many cache misses as the length of the key (8 for full 64-bit integers). On the other hand, CHFast and CHBucket incur into at most two (hash) table accesses, and each access loads records from the store. Thus, CHFast incurs into *at most* four cache misses, but CHBucket could still potentially incur into more; we go into more detail on this when analyzing CHBucket. Still, for ART we can also observe that the instruction count per lookup is the highest among the three structures. This lookup procedure works as follows: at every step, ART obtains the next byte to search for. Afterwards, by the adaptivity of ART, *every* lookup

has to test whether the node we are currently at is one of four kinds. Depending on this there are four possible outcomes; in which the lightest to handle is `Node256` and the most expensive in terms of instructions is `Node16`; where search is implemented using SIMD instructions. `Node4` and `Node48` are lighter in terms of instructions than `Node16` but more expensive than `Node256`. This lookup procedure is clearly more complicated than the computation of at most two Simple hash functions (multiplicative hashing).

Let us now discuss the limiting factors in the (lookup) performance of CHFast and CHBucket. Since the amount of

L3 cache hits is negligible, and the computation of Simple hash functions is also rather efficient, we can conclude that the (lookup) performance is essentially governed by the L3 cache misses, and actually, for CHFast that is the only factor, since the lookup procedure does nothing else than hash computations (two at most) and data access. The lookup procedure of CHBucket is slightly more complicated since each location in the hash table is a bucket that contains up to eight pointers ($8 \cdot 8$ bytes $= 64$ bytes) to the database store. The lookup procedure first performs a hash computation for the given key k (using the first hash function). Once the corresponding bucket has been fetched, it computes a small fingerprint of k (relying only on the least significant byte) and *every* slot of the bucket is then tested against this fingerprint. A record from the store is then loaded *only* when there is a match with the fingerprint; so there could be false-positives. The fingerprint is used to avoid loading from the store all elements in a bucket. If the fingerprint matches, the corresponding record is fetched from the store and then the keys are compared to see whether the record should be returned or not; in the former case, the lookup procedure finishes, and in the later we keep looking for the right record in the same bucket. If the bucket has been exhausted, then a similar round of computation is performed using the second hash function. This procedure clearly incurs into more computations than that of CHFast, and it also seems to increase branch misprediction — there is at least one more mispredicted branch per lookup (on average) than in ART and CHFast. We tested an (almost) branchless version of this lookup procedure but the performance was slightly slower, so we decided to keep and present the branching version. This concludes our analysis of the lookup performance.

	ART		CHFast-Simple		CHBucket-Simple	
Distribution	Dense	Sparse	Dense	Sparse	Dense	Sparse
Cycles	405.3	590.3	200.8	277.6	339.1	373.4
Instructions	149.3	151.1	34.60	40.13	91.94	97.78
Misp. Branches	0.027	0.972	0.126	0.662	1.645	1.857
L3 Hits	2.539	3.104	0.083	0.118	0.145	0.156
L3 Misses	2.414	3.831	2.397	3.716	3.189	3.460

Table I: *Cost breakdown per lookup for 16M.*

	ART		CHFast-Simple		CHBucket-Simple	
Distribution	Dense	Sparse	Dense	Sparse	Dense	Sparse
Cycles	785.0	1119	248.4	297.5	353.9	399.4
Instructions	164.9	162.9	36.02	39.68	90.75	97.00
Misp. Branches	0.045	0.686	0.303	0.638	1.608	1.825
L3 Hits	2.435	3.235	0.081	0.075	0.090	0.107
L3 Misses	4.297	6.671	2.863	3.747	3.170	3.472

Table II: *Cost breakdown per lookup for 256M.*

There is one more detail that we would like to point out: We can see from Figures 2 and 3 that CHBucket has a performance that lies between the performance-oriented CHFast and the space-efficient CHMem, but its space requirement is equivalent to that of CHFast. Thus, a natural question at this point is: does CHBucket make sense at all? We would like to argue in favor of CHBucket. Let us perform the following experiment: we insert 1 billion dense keys on CHFast-Simple, CHFast-Murmur, and CHBucket-Simple without preparing the hash tables to hold that many keys, i.e., we allow the hash tables to grow from the rather small capacity of $2 \cdot 2^6 = 128$ locations all the way to $2 \cdot 2^{30} = 2,147,483,648$ — growth is set to happen in powers of two. We observed that CHFast-Simple grows (rehashes) at an average load factor of 39%, CHFast-Murmur grows at an average load factor of 51%, and CHBucket-

Simple is always explicitly kept at a load factor of 75%, and it always rehashes exactly at that load factor[8]. The load factor of 75% was set for performance purposes — as the load factor of CHBucket approaches 100%, its performance drops rapidly. Also, by rehashing at a load factor of 75%, we save roughly 25% of hash function computations when rehashing in comparison of rehashing at a load factor near 100%. Thus, rehashing also becomes computationally cheaper for CHBucket, and follow up insertions and lookups will benefit from the new available space (less collisions). But now, what is the real argument in favor of CHBucket? The answer is in the robustness of CHFast-Simple. If CHFast-Simple was as robust as CHFast-Murmur, the former would always rehash around a 50% load factor, just as the latter, but that is not the case. This negative effect has been already studied [14], and engineering can alleviate it, but the effect will not disappear. Practitioners should be aware of this. On the other hand, CHBucket-Simple seems as robust as CHBucket-Murmur, and it could actually be considered as its replacement. Thus, by tuning the rehashing policy we can keep CHBucket-Simple at an excellent performance; in particular, considerably better than ART for somewhat large datasets.

We would like to close this section by presenting one more small experiment. In Figure 5 the effect of looking up for keys that are skewed can be observed. The lookup keys follow a Zipf distribution [6]. This experiment tries to simulate the fact that, in practice, some elements tend to be more important than others, and thus they are queried more often. Now, if certain elements are queried more often others, then they also tend to reside more often in cache, speeding up lookups. In this experiment *each* structure contains 16M dense keys.
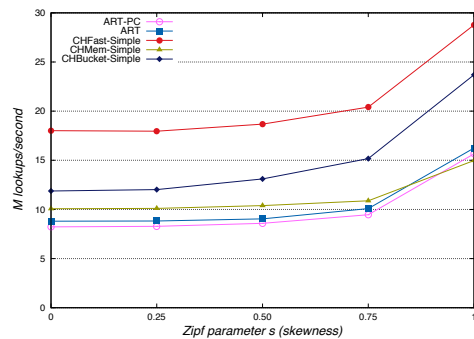

Figure 5: *Skewed (Zipf-distributed) lookups. Higher is better.*

We can see that all structures profit from skewed queries, although the relative performance of the structures stays essentially the same — CHFast-Simple and CHBucket-Simple set themselves strongly apart from ART.

D. Mixed workloads

The experiment to test mixed workloads is composed as follows: We perform one billion operations in which we vary the amount of lookups (point queries) and updates (insertions and deletions). Insertions and deletions are performed in a ratio 4:1 respectively. The distribution used for the keys is the dense distribution. Lookups, insertions and deletions are

---

[8]Without the manual 75% load factor, CHBucket-Simple rehashes on average at a load factor of 97%.

all independent from one another. In the beginning, *every* data structure contains 16M dense keys and thus, as we perform more updates, the effect of growing the hash tables will become more apparent as they have to grow multiple times. This growth comes of course with a serious performance penalty. The results of this experiments can be seen in Figure 6.
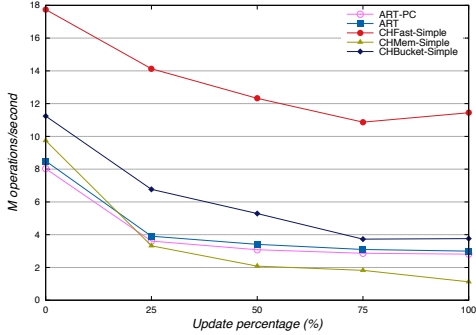

Figure 6: *Mixed workload of insertions, deletions, and point queries. Insertion-to-deletion ratio is 4:1. Higher is better.*

We can see how the performance of *all* structures decreases rapidly as more and more updates are performed. In the particular case of the hash tables, more updates mean more growing, and thus more rehashing, which are very expensive operations. Yet, we can see that CHFast-Simple remains in terms of performance indisputably above all other structures. We would also like to point out that, although the gap between ART (ART-PC) and CHBucket-Simple narrows towards the right end (only updates), the latter still performs around one million of operations per second more than the former; around 20% speedup. This can hardly be ignored.

E. Covering evaluation

In this section we confront experimentally *all* data structures considered in this study: Judy, ART, Google hashes, and Cuckoo hashing. Additionally, as B^+-trees are omnipresent in databases, we include measurements for a B^+-tree [16] from now on. As B^+-trees have been already broadly studied in the literature, we will not discuss them here any further — we just provide them as a baseline reference and to put the other structures in perspective. Unlike the experiments presented in IV-C, in this section we consider each data structure as a standalone data structure, i.e., covering ⟨key, value⟩ pairs. As we mentioned before, ART was designed to be a non-covering index, unable to cover keys and values. We also mentioned that, in order to compare ART against other data structures in this section, we endowed ART with its own store, which we now consider as a fundamental part of the data structure. For this store we chose the simplest and most efficient implementation, an array where each entry holds a ⟨key, value⟩ pair. This array, just as the hash tables, is pre-allocated and has enough space to hold at least n ⟨key, value⟩ pairs, for n = 16M, 256M, 1000M. We do all this to minimize the performance overhead contributed by the store of ART to the measurements; we simulate an ideal table storage. Therefore, we want to point out that, when it comes to ART, there is essentially no difference in the experiments presented

in Section IV-C and this section; the only actual difference is that in Section IV-C the store of ART is left out of the computation of space requirements since it is provided by the database. Here, nevertheless, this is not the case anymore.

As before, lookups are point queries and we query only existing keys. The insertion order is random, and once all pairs have been inserted, they are looked up for in a different random order — making insertions and lookups independent from each other. Insertion and lookup performance can be seen in Figures 7 and 8 respectively, and it is presented in millions of operations per seconds. The space requirement, in megabytes, of each data structure can be seen in Figure 9. Also, Tables III and IV present a basic cost breakdown *per lookup* for 16M and 256M respectively. Due to the lack of space, and the similitude of the performance counters between GHFast-Simple and CHFast-Simple, we present this cost breakdown only for JudyL, GHFast-Simple, and CHBucket-Simple. A comparison against ART can be done using the corresponding entries of Tables I and II on page 9.

Lookup performance. By just looking at the plots, Figures 7 and 8, we can see that there is clearly no comparison between JudyL and ART with GDenseHash-Simple and CDenseHash-Simple. The latter seem to be in their own league. Moreover, we can see that this comparison gets only worse for JudyL and ART as the size of the dataset increases; for one billion entries the space-efficient CHMem-Simple is now *at least* $2.3\times$ as fast as ART, and Judy, while requiring significantly less space than ART. In this regard, space consumption, JudyL is extremely competitive; for the dense distribution of keys no other structure requires less space than JudyL, and under the sparse distribution JudyL is comparable with the space-efficient hashes. However, all optimizations (compression techniques) performed by JudyL, in order to save space, come at very expensive price; JudyL is by far the structure with the highest number of instructions per lookup.

	JudyL		GHFast-Simple		CHBucket-Simple	
Distribution	Dense	Sparse	Dense	Sparse	Dense	Sparse
Cycles	623.6	931.9	94.32	140.2	116.9	141.2
Instructions	216.6	215.8	46.98	53.84	32.69	36.55
Misp. Branches	0.041	1.466	0.006	0.572	1.135	1.382
L3 Hits	3.527	4.016	0.016	0.043	0.077	0.083
L3 Misses	1.460	3.737	1.104	1.793	2.006	2.466

Table III: *Cost breakdown per lookup for 16M.*

	JudyL		GHFast-Simple		CHBucket-Simple	
Distribution	Dense	Sparse	Dense	Sparse	Dense	Sparse
Cycles	1212	1339.	94.72	143.2	126.9	146.4
Instructions	244.0	271.7	45.69	52.61	32.86	35.74
Misp. Branches	0.011	0.412	0.006	0.553	1.121	1.282
L3 Hits	4.103	3.116	0.025	0.058	0.084	0.085
L3 Misses	2.838	6.151	1.086	1.814	2.114	2.451

Table IV: *Cost breakdown per lookup for 256M.*

We can observe that the amount of long latency operations (L3 Hits + L3 Misses) of ART and JudyL are very similar; thus, we can conclude that the other limiting factor of JudyL is algorithmic, which, in the particular case of JudyL, it is also translated into code complexity — JudyL's source code is extremely complex and obfuscated.

With respect to the factors limiting the (lookup) performance of the hash tables, we can again observe that the amount of L3 cache hits is negligible, the instruction counts
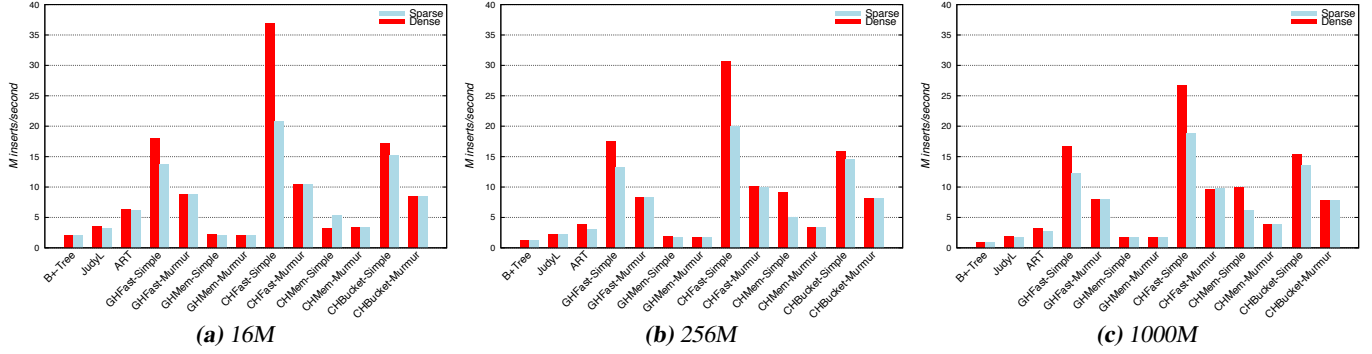
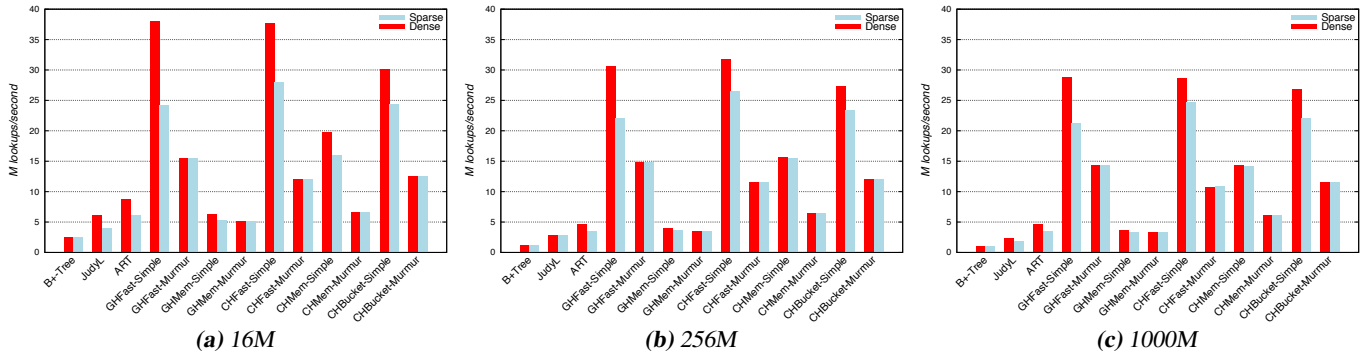***Figure 7:*** *Insertion throughput (covering). Higher is better.*

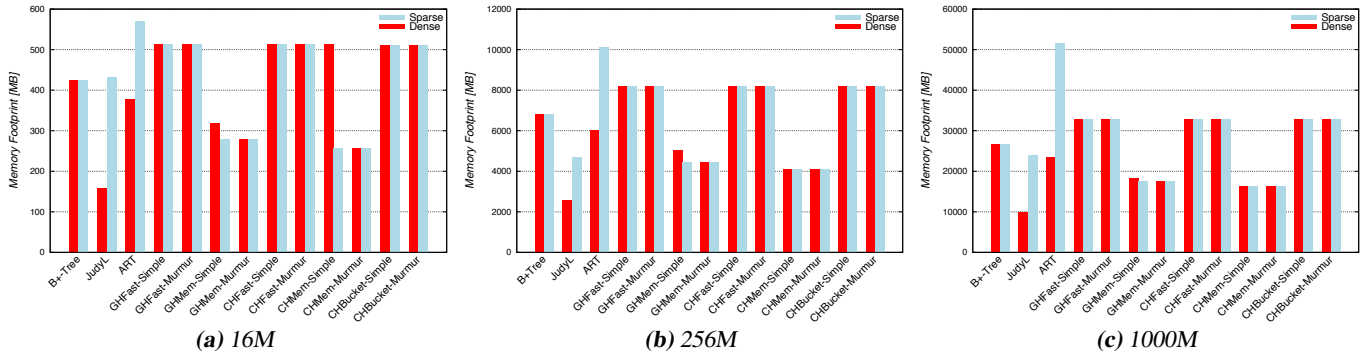***Figure 8:*** *Lookup throughput (covering). Higher is better.*

Figure 9: *Memory footprint in MB (covering). Lower is better.*

is very small, and thus, what is limiting the hash tables is essentially the amount of L3 cache misses. We can additionally observe that CHBucket-Simple incurs into more than one branch misprediction per lookup — these mispredictions are happening when looking for the right element inside a bucket. However, these mispredictions cannot affect the performance of CHBucket-Simple as they potentially do in its non-covering version (Tables I and II), since this time these mispredictions cannot trigger long latency operations due to speculative loads (usually resulting into L3 cache misses).

Range queries. So far, all experiments have considered only point queries. We now take a brief look at range queries, a longer study will be found in the extended version of this paper. Clearly, range queries are the weak spot of hash tables

since elements in a hash table are in general not stored in a particular order. However, in the *very particular* case that keys come from a small discrete universe, as in the case of the dense distribution, we could answer a range query $[a, b]$ by looking up in a hash table for *every* possible value between a and b, the whole range. Depending on the selectivity of the query, this method avoids looking up the whole hash table. For our experiment we fire up three sets of 1000 range queries, every set with a different selectivity; 10%, 1%, 0.1% respectively, on structures containing exactly 16M keys. For the sparse distribution we refrain ourselves from answering the queries using hash tables; it hardly makes sense. The results can be seen in Figure 10 below. All structures are covering versions, as the ones used in Section IV-E. As we mentioned

before, we implemented range-query support in ART, and our implementation is based on tree-traversal.
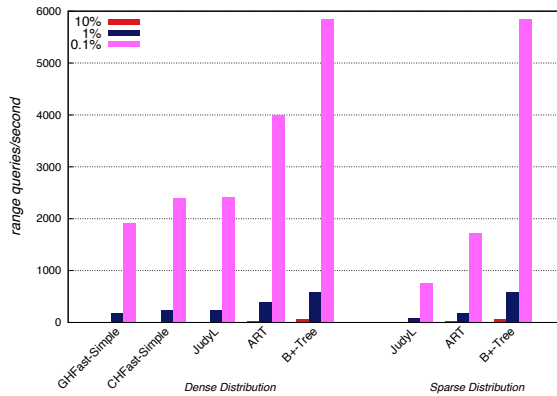

Figure 10: *Range queries over 16M dense and sparse keys. Covering versions of hash tables are only shown for dense keys. Higher is better.*

It is hard to see in the plot, but the difference in throughput between adjacent selectivities is a factor of 10. It is also very surprising to see that the hashes still perform quite good under the dense distribution. Again, the use cases for which hash tables can be used in this manner are *very* limited, but not impossible to find.

V. Conclusions

In the original ART paper [1], the authors thoroughly tested ART, and their experiments supported the claim that only a hash table was competitive (performance-wise) to ART. In our experiments we extended the original experiments by considering hashing schemes other than chained hashing. Our experiments clearly indicate that the picture changes when we carefully choose both, the hashing scheme *and* the hash function. *Our* conclusion is that a *carefully chosen* hash table is not only competitive with ART, but actually significantly better. For example, for an OLAP scenario, and for one billion indexed keys, one non-covering variant of Cuckoo hashing is *at least* $4.8\times$ faster for insertions, *at least* $2.8\times$ faster for lookups, and it sometimes requires just half the space of ART, see Figures 2, 3, and 4. For an OLTP scenario, the same variant is up to $3.8\times$ faster than ART, see Figure 6. We also tested ART against another (older) adaptive radix tree (Judy). In our experiments, ART ended up having almost $2\times$ better performance over Judy, but at the same time, it tends to also use twice as much space. This is an important trade-off to keep in mind.

Towards the very end we presented a small experiment to test performance under range queries. Here, ART was clearly outperforming Judy and all hash tables. However, ART is still slower than a B$^+$-tree by up to a factor of 3. Furthermore, we also observe that in the *very limited* case of a dense distribution coming from a small discrete universe, hash tables perform surprisingly good (comparable to Judy), and deciding whether hash tables could be use for range queries this way takes no time to the query optimizer.

Finally, in the extended version of this work we will extend our experiments by considering a 32-bit universe, by present-

ing changes to ART that slightly improve its performance, by extending the study on range queries, and by presenting interesting implementation details, that, for example, help to improve the robustness of multiplicative hashing.

VI. Acknowledgement

References

[1] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *ICDE*, 2013, pp. 38–49.

[2] 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 *SIGMOD*, 2010, pp. 339–350.

[3] J. Rao and K. A. Ross, "Making B+-trees cache conscious in main memory," in *SIGMOD*, 2000, pp. 475–486.

[4] D. Baskins, "Judy arrays," http://judy.sourceforge.net/ Version 31/07/14.

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

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

[7] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122 – 144, 2004.

[8] M. Mitzenmacher, *Some Open Questions Related to Cuckoo Hashing*. Springer Berlin Heidelberg, 2009, vol. 5757, pp. 1–10.

[9] A. Appleby, "Murmurhash64a," https://code.google.com/p/smhasher/ Version 31/07/14.

[10] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen, "A reliable randomized algorithm for the closest-pair problem," *Journal of Algorithms*, vol. 25, no. 1, pp. 19 – 51, 1997.

[11] Google Inc., "Google sparse and dense hashes," https://code.google. com/p/sparsehash/ Version 31/07/14.

[12] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis, "Space efficient hash tables with worst case constant access time," *Theory of Computing Systems*, vol. 38, no. 2, pp. 229–248, 2005.

[13] M. Dietzfelbinger and C. Weidling, "Balanced allocation and dictionaries with tightly packed constant size bins," *Theoretical Computer Science*, vol. 380, no. 1–2, pp. 47–68, 2007.

[14] M. Dietzfelbinger and U. Schellbach, "On risks of using cuckoo hashing with simple universal hash classes," in *SODA*, 2009, pp. 795–804.

[15] V. Leis, "ART implementations," http://www-db.in.tum.de/~leis/ Version 31/07/14.

[16] T. Bingmann, "STX B+-tree implementation," http://panthema.net/2007/ stx-btree/ Version 31/07/14.