Succinct and Fast Tiny Pointer Hash Tables

Xilin Tang
Cornell University
New York, NY, USA
xilin@cs.cornell.edu

Yuqi Mai
Cornell University
New York, NY, USA
ym562@cornell.edu

William Kuszmaul
Carnegie Mellon University
Pittsburgh, PA, USA
kuszmaul@cmu.edu

Alex Conway
Cornell Tech
New York, NY, USA
me@ajhconway.com

ABSTRACT

Hash tables sit on the critical path of many systems, yet modern designs still force a trade-off between fast operations and high memory overhead. We revisit this trade-off and present Tiny Pointer Hash Tables (TPHT), a family of practical hash tables that make two ideas from theory work at system scale: compressing pointers down to a byte, and encoding keys compactly so less metadata is needed. We engineer these into two complementary designs. Chained-TPHT targets maximal space savings, and is to our knowledge the first simple and practical succinct hash table, achieving a footprint smaller than the raw key-value payload size with constant-time operations. Flattened-TPHT targets latency, keeping the common case within a single cache miss while retaining strong space efficiency. Both variants support dynamic resizing without global pauses and integrate cleanly with 64-bit keys and values.

Across YCSB and microbenchmarks, TPHT advances the latency-space Pareto frontier: Chained-TPHT reaches 105.4% space efficiency, and Flattened-TPHT achieves 83.4% space efficiency with up to 89.3% higher throughput than strong baselines. Together, these results show that techniques primarily known in theory can be turned into systems-ready hash tables that meaningfully reduce memory use while delivering state-of-the-art performance.

PVLDB Reference Format:
Xilin Tang, Yuqi Mai, William Kuszmaul, and Alex Conway. Succinct and Fast Tiny Pointer Hash Tables. PVLDB, 19(9): 2168 - 2182, 2026.
doi:10.14778/3819518.3819542

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Xilinion/TinyPtr.

1 INTRODUCTION

Hash tables underpin a vast range of data-intensive systems, from distributed databases [1, 3, 92] and in-memory key-value stores [19, 21, 30, 56] to network monitoring [51, 80], genomics analysis [68, 85, 91], and real-time analytics engines [40, 82]. Because hash tables promise expected $O(1)$ queries, they are often the default choice for high-throughput applications. However, the size of datasets has exploded and continues to grow, while the cost of DRAM limits the practical memory capacity of servers. Nonetheless, state-of-the-art low-latency hash tables use only a fraction of their memory for data, with substantial portions used by metadata and collision resolution [2, 72, 75].

This paper revisits the fundamentals of hash tables and poses two questions:

- **Space.** How close can a practical hash table drive memory usage to the information-theoretic lower bound?
- **Speed.** Can it still rival the throughput of today's state-of-the-art designs while consuming markedly less memory?

We show how to build fast space-efficient hash tables using ***tiny pointers*** [10] and ***quotienting*** [4, 46, 66], theoretical techniques to compress pointers and keys respectively. We first demonstrate how these ideas from theory can be practically implemented, and then use them in two different hash-table designs. The first replaces the pointers in a chaining hash table with tiny pointers, which together with quotienting, results in an ultra space-efficient hash table; the table can even be smaller than the raw key-value payload![1] The second uses tiny pointers and quotienting in an open-addressing hash table to pack keys and collision-resolution metadata efficiently into each cache line, which yields a space-efficient hash table with state-of-the-art performance.

Tiny Pointers. Tiny pointers are implemented through a data structure called a ***dereference table***. A tiny pointer compresses a full-word address to just $\min(\log \delta^{-1}, \log\log\log n)$ bits, where $1-\delta$ is the load factor of the dereference table. Tiny-pointer theory uses recursive multi-level dereference tables to achieve theoretically optimal space usage, but this is difficult to implement in practice. One contribution of this paper is to offer a simpler implementation based on the power of two choices, which empirically supports allocation without failure up to a load factor of 98.2%.

In our hash tables tiny pointers use 8 bits instead of the 64 bits of a standard machine word or the $O(\log n)$ bits that a straightforward index would require.

Quotienting. The core idea behind quotienting is easy to understand if we consider a chained hash table that stores fingerprints (i.e. hashed keys) instead of the keys themselves. The chain that a key

Proceedings of the VLDB Endowment, Vol. 19, No. 9 ISSN 2150-8097.
doi:10.14778/3819518.3819542

[1] Minimal and dynamic perfect hashing [8, 27] can approach even smaller footprints by compressing hash functions for fixed or slowly changing key sets, but they typically target membership or static dictionaries rather than storing full key-value pairs with general updates.

belongs to is determined by the high-order bits of its fingerprint, referred to as a ***quotient***. Therefore, these bits are already implicitly determined by each item's location, and so the table need only store the low-order bits of the fingerprint, the ***remainder***. This technique is used in quotient filters [11] to build a near optimal filter data structure.

However, hash tables need to store the key itself, not just the fingerprint, because fingerprints can collide. To solve this problem, we use a technique, one-round Feistel permutations, from the theory literature to encode the keys themselves so that they can be quotiented without losing the original key. This allows us to build real-world hash tables that use quotienting to save roughly $\log n$ bits of space per key.

Tiny-Pointer Hash Tables. We use tiny pointers and quotienting to design two novel high-performance space-efficient hash tables.

Chained-TPHT. Our first design emphasizes space efficiency by using a chained hash table design, with a head array storing (tiny) pointers to the first link in each chain. The links in the chains use tiny pointers and the keys are stored using quotienting. Perhaps surprisingly, under most configurations, Chained-TPHT simultaneously achieves lower memory usage than ***even storing the raw key-value payload*** and strong overall performance.

Chained-TPHT uses tiny pointers to reduce the space consumed by pointers in the head array and the chain links. Quotienting then recovers enough memory from the keys to fully compensate for this (much reduced) overhead.

Because the head array uses tiny pointers, its size is a small fraction of the payload size. This means that it can have sufficiently many entries that each chain is short, so that operations incur few cache misses. Ideally, in a system with a 64 MiB L3 cache, the head array for a 1 GiB table can fit entirely in cache.

Despite being one of the oldest and simplest data structures in computer science[2], chained hash tables have since been considered to be impractical in space-sensitive settings because of the many pointers that the data structure requires. Chained-TPHT turns this conventional wisdom on its head, demonstrating that, with the right algorithmic techniques, chained hashing can actually be implemented as an ***extremely*** space-efficient data structure.

Flattened-TPHT. While Chained-TPHT achieves excellent memory-efficiency, there is still a latency overhead due to the indirection of having to first access the head array and then access the data itself. Our second design, Flattened-TPHT, is less memory-efficient, but allows most operations to be completed using a single cache miss, leading to better average latency. Furthermore, operations average 1.25 cache misses and never exceed three (outside resizing); these bounds yield a strong tail-latency profile.

Flattened-TPHT accomplishes this by replacing the head array with an array of cache-line-sized home groups. Each key hashes to a home group, which itself stores 0–4 key-value pairs. When more than 4 key-value pairs hash to the same home group, some "overflow" key-value pairs are stored in a dereference table, and are referenced from the home group using tiny pointers. The home entries are accessed in a single cache miss, whereas the overflow entries require two cache misses. The memory-efficiency tradeoff occurs because the home groups will not have uniform occupancy, so some memory in home groups will go unused.

Consistent with prior work [44, 59, 63, 70, 72], both variants support 64-bit keys and values, deletions, and online resizing. As in previous hash-table designs, they can be straightforwardly extended to variable-length keys and values by storing pointers to the actual keys and values in the hash table.

Results. Chained-TPHT attains 105.4% space efficiency, reducing memory by 38.6% relative to state-of-the-art baselines. Flattened-TPHT delivers 83.4% space efficiency and cuts memory by 22.3% while running 89.3% faster than the same baselines.

Contributions. This paper contributes the following:

- **Practical tiny pointers.** We present a simple and practical implementation of tiny pointers, which empirically results in excellent compression with efficient dereferencing.
- **Practical key quotienting.** We incorporate quotienting into real-world hash tables, saving $\log n$ bits per key.
- **Beyond 100% space efficiency.** Using these two techniques, Chained-TPHT is 1.37-20.13× the speed of prior hash tables whose footprint is below the raw payload size.
- **State-of-the-art performance with less space.** Flattened-TPHT pairs the same techniques with a cache- and SIMD-aware layout to surpass prior tables in performance while using less memory.

The paper also comes with a contribution of theoretical interest. From a theory perspective, Chained-TPHT is a ***succinct hash table***, a hash table that uses space within a factor of $(1 + o(1))$ of the information-theoretic optimum, while offering $O(1)$ expected-time operations.[3] Such hash tables have existed in the theory literature (e.g., [4, 12, 13, 77]), but are considered too complicated (and with too poor of constants) to be practical. (As we will discuss in Section 2, there are practical ***compact hash tables*** [23, 47, 48, 74], which can achieve $(1 + o(1))$ of the space optimum but not with $O(1)$ operations.) Chained-TPHT, in addition to being practical, is arguably also the simplest succinct hash-table design to date.

2 BACKGROUND

Most hash-table designs can broadly be divided into two categories, ***separate chaining*** and ***open addressing***.

Separate chaining. A separate chaining hash table hashes each key to a bucket, which is then stored as a linked list. Each list is referred to as a ***chain***. The canonical design uses a ***head array***, each entry of which stores a pointer to the first link in its chain. Variations on this design optimize for resizing and performance bottlenecks. Split-Ordered Lists [79] enable lock-free incremental resizing while preserving bucket semantics, and CLHT [26] improves cache locality by packing multiple entries into cache-line-sized mini-buckets.

A major source of memory overhead in separate chaining are the pointers in the chains. Each item must store a pointer, so for example if the hash table stores 64-bit keys and values and uses

[2]First introduced in 1953 by Hans Peter-Luhn, chained hash tables predate both unbalanced and balanced binary trees. In fact, according to Knuth, the chained hash table may be the first ever use of a ***linked list*** in a computer program [46]

[3]In the standard parameter regime where keys are of size $w = (1 + \Theta(1)) \log n$ bits, Chained-TPHT uses space $B + O(n \log \log n) = (1 + O(\log \log n / \log n))B$ bits, where B is the information-theoretic optimum.

64-bit pointers, this immediately leads to a 50% memory overhead. However, additionally, assuring performance in the table often requires even more memory overheads.

Separate chaining performance is dominated by cache misses incurred during operations: queries must access the head pointer and traverse the chain until we find the key. Therefore, for performance reasons, it's important to have enough buckets that the chains are short. However, doing so introduces more null pointers in the head array, causing additional memory overhead. Some designs store key-value pairs in the entries in the head array. This removes the indirection but further inflates entry size and leaves many entries empty.[4] In CLHT, it actually packs each link as a cache-line mini-bucket (up to three 64-bit tuples inside). This improves the locality of chain traversals, since multiple items can be retrieved in a single cache line. However, the imperfect load balancing typically leaves over 62%[72] of memory unused.

So in terms of memory efficiency, separate chaining starts with a large overhead from pointers and then must decide how much additional memory overhead to trade for performance reasons.

Open addressing. Open-addressing hash tables store all entries directly in a single array, using a probe sequence to resolve collisions. The most common variant, linear probing, searches sequentially from the hash location until finding an empty slot or the target key [18, 32]. Other schemes like quadratic probing [50], double hashing [36], and cuckoo hashing [71] use different probe sequences to distribute keys and reduce clustering effects.

The primary advantages of open addressing are data locality and implementation simplicity. However, performance degrades rapidly as the load factor increases due to longer probe sequences and increased cache misses. To maintain good performance, most open addressing schemes operate at load factors, typically below 75% [2, 46], leaving substantial memory unused.

Recent designs attempt to mitigate these issues through metadata optimizations. Google's Swiss Tables [2] and Meta's F14 Hash Table [29] use metadata bytes to quickly skip over occupied slots during probing.

Compact Hash Tables. Recall that a hash table is ***compact*** if it uses space within a factor of $1 + \delta$ of optimal, for some $\delta \in (0, 1)$, while supporting operation times that are a function of δ^{-1}. This is a weaker guarantee than being ***succinct***, which means that the hash table can support $\delta = o(1)$, with $O(1)$-time operations. Whereas (until this work) succinct hash tables have been theoretical only, several designs of compact hash tables have been shown to be practically feasible. (See, also, Section 5.3 and Section 8.3.1 for comparisons with Chained-TPHT.)

Like separate chaining, compact bucketing [47, 48] uses variably sized buckets to store entries, but instead of storing them as linked lists, they store them on the heap and reallocate them as necessary. This allows the hash table to be space efficient, but the cost of reallocation significantly hurts update performance and causes memory fragmentation.

Compact linear-probing schemes [23] use quotienting together with metadata bits to store partial key information, but they suffer a performance penalty at high load factors due to long probe sequences. Layered versions [74] use multiple layers to reduce probe distances and allow for better performance at high load factors.

[4] With n elements and n buckets, about $1/e \approx 0.37$ of buckets are empty.

3 PRACTICAL TINY POINTERS

Tiny pointers offer a path to compressing pointers in data structures.

3.1 Tiny Pointers and Dereference Tables

The tiny-pointer user interface. Tiny pointers are made possible through a data structure called a ***dereference table***, which functions as a specialized memory allocator that will contain the objects (in our case, nodes in the hash table) that the tiny pointers will point at. Given a ***capacity*** n and an ***object size*** s, we create a dereference table of size roughly $n \cdot s$ bytes, capable of storing up to n objects of s bytes each. The dereference table provides the following user interface for tiny pointers:

(1) `Allocate`(k) allocates s bytes in the dereference table, and returns a tiny pointer p. The parameter k is the ID for the object, and must be unique among all other IDs that are currently active. The same ID k must be provided in order to dereference or free p in the future.
(2) `Dereference`(k, p) returns a (non-tiny) pointer to the object with ID k. The prerequisite is that the ID k is active with tiny pointer p.
(3) `Free`(k, p) deallocates the object with ID k. The prerequisite is that k is active with tiny pointer p.

Differences between traditional and tiny pointers. Tiny pointers go hand-in-hand with the ***dereference table***, which functions as a specialized memory allocator. Whereas standard memory allocators allocate objects of different sizes and are primarily concerned with fragmentation, dereference tables allocate objects of fixed size s and don't face this issue. However, this means that they cannot be used to allocate variable-sized objects without using an additional level of indirection. On the other hand, objects in a dereference table must be able to be referenced via a 1-byte tiny pointer, whereas standard allocators can use 8-byte pointers. That this is possible is non-obvious and requires significant theoretical techniques [10].

Notably, even though each object in the dereference table is given a user-assigned ID k, these IDs are not actually ***stored*** in the dereference table. (Indeed, the entire dereference table consists of roughly $n \cdot s$ bytes, which is only enough space to store the allocated n objects.) Rather, it is the ***user***'s job to provide the user-assigned ID k whenever a tiny pointer is dereferenced or freed.

Finally, tiny pointers ***preserve*** a key property of traditional pointers: following a pointer requires a single memory access. As we will see later on in our proposed implementation, the translation of tiny pointer to pointer (via a call to `Dereference`(k, p)) incurs no additional memory accesses.

3.2 Practical Implementation

Allocating tiny pointers with two-choice bucketing. Recall that a dereference table is initialized with a maximum capacity n and an object size s. The dereference table will consist of $N \cdot s$-byte slots, where N is selected to be slightly larger than n (say, $n \approx 0.95N$). As we shall see, the choice to have N slightly larger than n (even

just, say, 5% larger) will be critical for good pointer compression, as it allows flexibility in which slot to use for each allocation.

The dereference table consists of two arrays:

- **Meta table** ($2n$ bytes). Each of the n ***bin***s stores a two-byte header: the current free-slot count and the offset of the first free slot, i.e., the list head of the free list.
- **Data table** ($\approx n$ entries). Each consecutive block of $(2^7 - 1)$ entries form a bin; free slots inside a bin are threaded into a free list via 7-bit offsets.

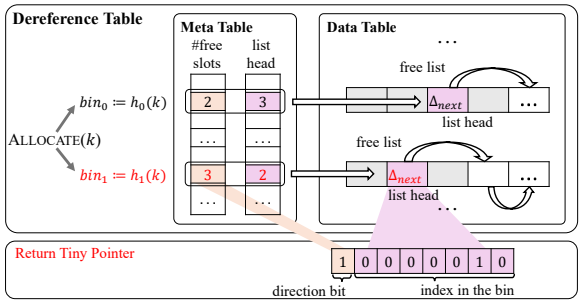


Figure 1: Meta layout, data bins, and the steps of `Allocate`(k): two hashes, choose emptier bin, pop head, return 8-bit p.

`Allocate`(k) as shown in Figure 1, compares the free-slot counters of bins $h_0(k)$ and $h_1(k)$, pops a free slot $j \in \{1, \ldots, 127\}$ from the emptier bin $h_i(k)$, and returns the 8-bit tiny pointer $i \circ j$, matching the layout above with 0 reserved for null. `Dereference`(k, p) interprets p as $i \circ j$ and returns the address of j-th slot of bin $h_i(k)$. Surprisingly, dereference is pure arithmetic and needn't a memory access. `Free`(k, p) decodes the same fields, increments that bin's free counter, and threads j back into its free list.

The main challenge: avoiding allocation failures. In this basic design, an allocation may fail if both bins $h_0(k)$ and $h_1(k)$ are full. The key design principle is to allocate tiny pointers so that these failures almost never happen.

To minimize allocation failures, our proposed design relies heavily on a probabilistic phenomenon known as the ***power of two choices*** [6, 7, 14, 81]. This says that, if a balls are placed into b bins, and if each ball is placed in the emptier of two ***random*** bins, then (with very high probability) all of the bins will have very similar loads (no bin will have more than $a/b + \log\log b + O(1)$ balls). Unlike many theoretical phenomena, the power-of-two choices has very good real-world constants. Importantly, allocation failure probability is not simply a per-operation constant; it guarantees that no failure occurs w.h.p. for polynomially many operations. Our implementation reaches sustained load factors well beyond the 95% suggested by the analysis in Section 8.6.

Difference between our implementation and the theory. In [10], dereference tables are designed as a complex structure with multiple hash-table-like levels. The bits of each tiny pointer are used to both identify the hash functions and the index in the substructure of hash tables. This design involves complex multi-level load-balancing as well as careful partitioning of the tables, which makes it difficult and inefficient to implement in practice. It further uses exhaustive tabulation (also known as the method of Four Russians) to implement some operations in constant time, which does not work well in practice[5].

3.3 Example: Chained Hash Tables

We now demonstrate how tiny pointers can be used in the context of a simple chained hash table. This example will take us part of the way towards understanding Chained-TPHT, which will go further by integrating quotienting.

Having initialized the dereference table, we can now allocate nodes for the hash table. In a chained hash table, each node lies in a linked list and is referenced by a unique tiny pointer. To allocate a node, we call `Allocate` with a unique ID for that node, and store the returned tiny pointer in the appropriate place (e.g., the head array for the bucket, or the next pointer of another node).

One might be tempted to use the key k stored in the node as the unique ID. This works when k is the first node in the chain. However, if another key ℓ is stored in the first node, the head tiny pointer p will be allocated with ID ℓ, so calling `Dereference`(k, p) will not return the correct node (or even necessarily a valid node). Therefore it is important that the IDs for the tiny pointers in each chain be computable from the bin index and the chain prefix. Moreover, the ID must be unique across objects in the dereference table, and is required whenever operating on the tiny pointer, e.g., `Dereference`(k, p) to obtain a 64-bit pointer to u or `Free`(k, p) to release u. This highlights an important general challenge to using tiny pointers: determining unique and computable IDs.

In this hash-table example, we solve this challenge by using the address A of each tiny pointer p as its unique ID k. This means that the two bins where the node pointed to by p can be allocated are $h_0(A)$ and $h_1(A)$, where h_0 and h_1 are the two hash functions used in our dereference table implementation. Notice that this ID satisfies both the uniqueness and computability requirements: each tiny pointer has a unique address, and the address is known whenever the tiny pointer is used. See Figure 2 for an example query.

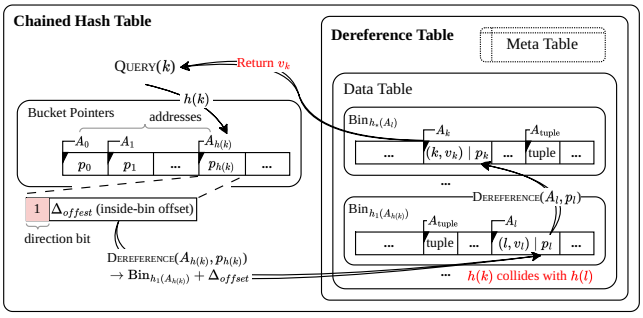


Figure 2: Query for key k in the chained hash table with tiny pointers. The direction bit of the first tiny pointer is 1 in this example, and leads to the bin corresponding to $h_1(A_{h(k)})$ in our dereference table implementation. The meta table is not used during queries.

[5]Our implementation's average throughput is 2.17×–3.69× that of a prior third-party implementation [78] directly following the theoretical work [10].

4 QUOTIENTING

The idea behind quotienting is to use the bin index to implicitly store part of each key. If the keys themselves are random or if it's sufficient to store fingerprints rather than the original keys, then the high-order bits of the random key or fingerprint can be directly used to determine the bin index. Then these high-order bits do not need to be stored themselves and can be reconstructed from the bin index when needed.

The use of quotienting in hash tables was suggested by Knuth [46], and has been applied in quotient filters [11, 28, 34, 73], which can be interpreted as quotienting hash tables which store fingerprints rather than keys. There is a literature of hash tables that use quotienting in theory to achieve memory-efficiency [4, 9, 34], and some of these have practical implementations [23, 37, 47, 48, 74].

The difficulty is that keys sharing a bin need not share a prefix. One fix is to use an invertible hash function—a random permutation—to map each key to a fingerprint that preserves its information. This would, in principle, allow quotienting as described above. The problem is that no known techniques are practical, and fully random permutations are hard to realize even in theory [20].

To get around this, we use a construction called a one-round Feistel permutation [66]. Rather than trying to emulate a fully random permutation hash function, this construction makes a specified bit range of the key "look random" and then uses that range for quotienting. Feistel permutations have been used in theoretical hash-table designs [4], as well as in a GPU implementation [37].

4.1 One-Round Feistel Permutations

For a key k, set $k = q^{\text{pre}} \circ r$, where q^{pre} is the prefix of k that we will quotient off, r is the remainder suffix. Let w be the word length and $2^{w-\ell}$ be any suitable divisor. Starting from any hash family $\mathcal{H}$ of functions $h : [2^w] \to [2^\ell]$, we define $\mathcal{H}_q = \{h' \mid h'(q^{\text{pre}} \circ r) = h(r) \oplus q^{\text{pre}}, h \in \mathcal{H}\}$, where $\oplus$ denotes XOR. Intuitively, h' infuses randomness from $h(r)$ into the prefix of the key. Importantly, if two keys have the same h' value, they can be compared using only the stored remainder, and each key can be reconstructed from the remainder and h'. We realize the one-round Feistel permutation when calculating $h(r) \oplus q^{\text{pre}}$, which we refer to as the ***quotient***. Algorithm 1 shows this quotienting process.

Algorithm 1 One-Round Feistel Quotient

Require: Key $k = q^{\text{pre}} \circ r$, hash function h

1: **procedure** `Quotient`(k) **return** $q \leftarrow h(r) \oplus q^{\text{pre}}$

2: **procedure** `RecoverKey`(q, r) **return** $k \leftarrow (q \oplus h(r)) \circ r$

Note that $\mathcal{H}_q$ need not preserve the independence properties of $\mathcal{H}$. In particular, even if $\mathcal{H}$ is fully random, $k_1 = q_1^{\text{pre}} \circ r$ and $k_2 = q_2^{\text{pre}} \circ r$ will never collide if $q_1^{\text{pre}} \neq q_2^{\text{pre}}$. Therefore $\mathcal{H}_q$ is not even pairwise independent. Following [61], we show that $\mathcal{H}_q$ is universal and has strong distributional properties—in particular, Chernoff bounds hold and so its max load is the same as $\mathcal{H}$'s.

Claim 1. *$\mathcal{H}_q$ is universal when $\mathcal{H}$ is pairwise independent.*

Proof. For distinct keys $k = q_1^{\text{pre}} \circ r_1$ and $k' = q_2^{\text{pre}} \circ r_2$, pairwise independence of $\mathcal{H}$ implies

$$\Pr_{h' \sim \mathcal{H}_q}\left[h'(k) = h'(k')\right] = \Pr_{h \sim \mathcal{H}}\left[h(r_1) = h(r_2) \oplus q_1^{\text{pre}} \oplus q_2^{\text{pre}}\right] \leq 2^{-\ell}. \quad \square$$

Claim 2. *If $\mathcal{H}$ is mutually independent, then $\mathcal{H}_q$ has expected max load $\Theta(\log n / \log\log n)$.*

Proof. The indicators $X_i^{(b)} = \mathbf{1}[h'(k_i) = b]$ (with $h' \sim \mathcal{H}_q$) are negatively associated, so standard Chernoff bounds apply to $X^{(b)} = \sum_i X_i^{(b)}$ [87].

Thus, by the standard result [76], the Chernoff bounds imply that the expected max load is $\Theta(\log n / \log\log n)$. $\square$

In practice we instantiate $\mathcal{H}$ with xxHash [24].

5 CHAINED TINY POINTER HASH TABLE

In this section, we present a chained hash table that fuses ***tiny pointers*** with ***quotienting***. The resulting structure is highly space-efficient; as demonstrated in Section 5.3, its footprint can even fall below the raw key-value payload size.

At a high level, Chained-TPHT is a chained hash table that stores all nodes in a single high-load-factor dereference table. Further, Chained-TPHT uses quotienting to reduce the size of stored keys.

5.1 Design of Chained-TPHT

Let n denote the number of stored tuples and let $2^{w-\ell}$ be the divisor chosen for quotienting. We refer to the set of keys with a given quotient q as q's ***quotient group***. In Chained-TPHT, chains precisely correspond to quotient groups, so we will sometimes refer to chains and quotient groups interchangeably. The data structure comprises two tables:

- **Head array** (2^ℓ entries). Each cell holds a tiny pointer to the first node in the chain for the corresponding quotient group (or `NULL` if the group is empty).
- **Dereference table** ($\approx n$ slots). Each slot stores a triple (p, r, v) consisting of the tiny pointer that links to the next element, the quotiented key (remainder), and the value. In the ideal case this takes $8-\ell+2w$ bits per tuple; we provision a small slack to stay below the critical load.

Chain identification. Figure 3 illustrates the data path: for a key k we compute its quotient using the hash function from Section 4.1. The quotient is used to determine which chain k belongs to, and in particular is used as the index in the head array. When accessing keys in that chain, we recover the original prefix as $q \oplus h(r)$ and concatenate it with the stored remainder r.

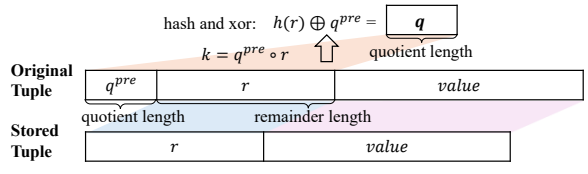


Figure 3: Quotienting pipeline: the hash determines the quotient and only the remainder is stored in the chain.

Monolithic dereference table. As in a traditional chained hash table, tuples in a chain (quotient group) are organized as a linked list. All chain nodes across different chains are to be stored in a flat memory pool, which is a single dereference table. So, even though the chains are logically partitioned, they are physically intermingled in a single table.

Tiny pointer IDs in Chained-TPHT. As discussed in Section 3.1, assigning IDs to tiny pointers in a chained hash table is subtle. Recall that when allocating a node, we cannot use the stored key k as the ID. Instead, we use the address of the unique [tiny] pointer in the list that points to the node. Importantly, these IDs are unique and known at dereference time, since we obtain the tiny pointer from its address.

Hash table operations. The procedure for inserting a key-value pair (k, v) is detailed in Algorithm 2. First, the key k is split into a prefix q^{pre} and a suffix r to compute a bucket index (quotient) $q = h(r) \oplus q^{\text{pre}}$. Denoting the head array by B, we then traverse the linked list starting from $B[q]$, using the in-place address of each tiny pointer's location as its ID for dereferencing. Upon reaching the end of the chain, a new entry is allocated in the dereference table and linked into the list. The new entry is then populated with the quotient, the value, and a `NULL` tiny pointer.

Queries, updates, and deletes follow a similar procedure to locate the target entry by traversing the appropriate chain. Once found, the operation is completed by interacting with the dereference table via the interfaces from Section 3.2.

Algorithm 2 INSERT

Require: Global head array B, key $k = q^{\text{pre}} \circ r$, value v

1: $q \leftarrow h(r) \oplus q^{\text{pre}}$ ▷ Quotient group identifier
2: $p^{\text{tp}} \leftarrow \text{addr}(B[q])$ ▷ Pointer to the tiny pointer
3: $k^{\text{deref}} \leftarrow \text{addr}(B[q])$ ▷ In-place key for dereference
4: **while** $*p^{\text{tp}} \neq$ `NULL` **do** ▷ Follow to the end of the chain
5: $e \leftarrow$ `Dereference`$(k^{\text{deref}}, *p^{\text{tp}})$
6: $p^{\text{tp}} \leftarrow \text{addr}(e.p)$; $k^{\text{deref}} \leftarrow \text{addr}(e.p)$ ▷ Advancing
7: e, $*p^{\text{tp}} \leftarrow$ `Allocate`(k^{deref}) ▷ Reserve new entry and link it
8: $e.k \leftarrow r$; $e.v \leftarrow v$; $e.p \leftarrow$ `NULL`

5.2 Performance Advantages

Beyond space efficiency, Chained-TPHT offers two key performance advantages over conventional chaining methods: enhanced cache locality and shorter average chains.

Enhanced cache locality. A conventional chained hash table uses a head array of 8-byte pointers to locate the head of each chain. Replacing these with 1-byte tiny pointers shrinks the head array by a factor of eight. This greatly improves the chances that the array stays in CPU cache—in the best case, the entire head array fits in cache. Consequently, a higher fraction of head lookups become cache hits, reducing latency.

Shorter chains. The average chain length is determined by the ratio of elements to key groups. In traditional designs, enlarging the head array to add key groups incurs substantial space overhead and impacts cache efficiency. The minimal footprint of tiny pointers essentially eliminates this trade-off. With Chained-TPHT, it is inexpensive to size the head array so that $\lambda = n/2^{\ell} \leq 1$. Under the standard Poisson approximation, a successful lookup for a uniformly random stored key finds its target at the first node in its chain with probability approximately $(1 - e^{-\lambda})/\lambda$, which is at least $1 - e^{-1} \approx 63.2\%$ when $\lambda \leq 1$. A minor limitation is diminishing returns from further enlarging the head array: while multi-element chains shrink slowly, a larger array can exceed cache and erode locality.

A remark on the symbiosis between tiny pointers and quotienting. Part of the reason we can practically use quotienting in Chained-TPHT is that chained hash tables, ***a priori***, are much easier to quotient than open-addressed hash tables (which require carefully managed metadata for quotienting to be possible). Historically, this compatibility was viewed as essentially pointless, since chaining pays so much space overhead for pointers. With tiny pointers, however, we get the best of both worlds—the low metadata overhead of open addressing with the convenient quotienting of chaining.

5.3 Space Analysis

We now show that Chained-TPHT is nearly space optimal.

Consider the word-RAM model with a word size of w bits ($w = 64$ on most contemporary machines). Storing n key-value pairs verbatim consumes $2n$ words, or $2nw$ bits. Yet a hash table represents only the ***set*** of keys without regard to their order.

Assuming all keys are distinct, the number of possible key sets is $\binom{2^w}{n}$. Any representation therefore requires at least

$$\log \binom{2^w}{n} \approx n\big(w - \log n\big) + n \log e + o(n)$$

bits, where the approximation follows from Stirling's formula. It holds with approximately 1% error when $1000n \leq 2^w$. Relative to the naïve nw-bit allocation, the entropy bound saves roughly $\log n$ bits ***per element***.

For Chained-TPHT, putting the pieces together, besides each w-bit value, each key is stored as a remainder of $w - \log n$ bits (its high-order bits being determined by the bin index via quotienting) together with one 8-bit tiny pointer; the head array contributes $8n$ bits and the meta table $16n/127$ bits. Accounting for load factor $1 - \delta$ (empirically $\delta \approx 2\%$), the total footprint is

$$\frac{n\big(2w - \log n + 8 + 16/127\big)}{1 - \delta} + 8n \text{ bits},$$

which is nearly optimal relative to the entropy bound while preserving constant-time operations. We remark that, from a theoretical perspective, tiny pointers of $O(\log \log n)$-bits support a dereference table with load factor of $1 - \delta = (1 - 1/\log n)$ [10]. This means that, in the standard parameter regime of $w = O(\log n)$, Chained-TPHT uses $B + O(n \log \log n)$ bits, where $B = \log \binom{2^w}{n}$ is the information-theoretically optimal space usage.

Table 1 summarizes the space and operation complexity for compact hash tables. Here, instantiation means the practical structure that results from adapting the theoretical design in parameters and structure (e.g., fitting a $\log \log n$ component into a byte). Notably, at load factor $1 - \delta$, the insertion (and sometimes query) cost of prior compact hash tables depends inversely on δ, so at near-optimal

space $(1+o(1))B$ they incur $\omega(1)$ or $O(\log n)$ operation. Chained-TPHT, by contrast, achieves both $(1+o(1))B$ space and $O(1)$ operations, making it a simple and practical ***succinct*** hash table.

Table 1: Asymptotic and instantiation space redundancy over the approximate information-theoretic optimum $n(2w-\log n)$ in bits and expected operation complexity for compact hash tables. In this table only, load factor is written as $1/(1+\bar{\delta})$ (elsewhere $1-\delta$) to keep the formulas compact. Here $w = O(\log n)$. For Chained-TPHT, $\bar{\delta}^* = o(1)$, distinct from the $\bar{\delta} = \Theta(1)$ used for the other methods.

Method	Space Redundancy	Instantiation Redundancy	Insert	Query
Chained-TPHT	$O(n \log\log n)$	$\bar{\delta}^*(w+8+16/127)n+8n$	$O(1)$	
Cleary [23]	$O(\bar{\delta}n \log n)$	$\bar{\delta}(w+5)n$	$O(\bar{\delta}^{-2})$	
Layered [74]	$O(\bar{\delta}n \log n)$	$\bar{\delta}(w+3+14/9+o(1))n$	$O(\bar{\delta}^{-2})$	$O(\bar{\delta}^{-1})$
Group [48]	$O(n)$	$(4+1+1)n$	$O(\log n)$	$O(1)$
Bucket [48]	$O(n \log\log n)$	$(8+1)n$	$O(\log n)$	

6 FLATTENED TINY POINTER HASH TABLE

Flattened-TPHT pushes the speed-space Pareto frontier by trading some compression headroom for locality: it eliminates chain indirection, flattens collision chains into a cache-friendly layout, and vectorizes fingerprint comparisons, retaining space efficiency while matching or exceeding the throughput of strong baselines.

6.1 Flattened-TPHT Design

To minimize memory access latency, Flattened-TPHT employs a cache-efficient flattened data layout. Specifically the layout is designed with three goals in mind: eliminating head pointer indirection, maximizing cache line locality, and flattening collision chains.

Head pointer elimination. Accessing a quotient group in Chained-TPHT via its base pointer array incurs an extra memory indirection. In Flattened-TPHT, rather than using a head array, we use a ***home array***. In the home array, entries are ***home blocks***, which store up to a fixed number of ***home tuples***, as well as pointers to ***overflow tuples***. One can think of this as removing the home tuples from the dereference table and storing them directly in the array. As in Chained-TPHT's home array, each home block corresponds to a quotient q, so that all keys with quotient q can be found in (as a home tuple) or via (as an overflow tuple) that home block.

Cache line locality. Each aligned 64-byte home block fits into a cache line. To leave room for metadata and tiny pointers to overflow tuples alongside up to four home tuples, we shrink each stored key by quotienting off at least one byte; this requires only $\ell - 2 \geq 8$, i.e., a reasonable minimum of 2^{10} inputs. The design improves locality but costs space in two ways: underfull home blocks leave unused bytes, and averaging four keys per quotient group means quotienting two fewer bits than in Chained-TPHT. Sharing one home block across quotient groups would recover those bits but would require extra metadata and prevent the chain-flattening optimization below.

Chain flattening. Overflow tuples are not linked as in Chained-TPHT; instead, their tiny pointers are stored directly in the home block, so any lookup needs at most one extra cache miss and avoids pointer chasing. If a quotient group outgrows the home block, one resident tuple is migrated to the dereference table and replaced by a tiny pointer that also references the evicted tuple. With at least two quotient bytes, four home tuples leave six bytes for three fingerprint-tiny-pointer pairs, so migration is needed only above seven tuples per quotient group, which occurs with probability 5.1% according to a Poisson approximation.

6.2 Vectorized Lookups

We now alter the home-block structure to leverage SIMD on modern processors.

Our vectorized approach extracts a one-byte fingerprint from each stored tuple. Each home block stores these fingerprints at the start of its cache line, one per tuple, including both home and overflow tuples.

During a lookup, we compare the query's fingerprint against all fingerprints in the array using a single SIMD instruction (e.g., `_mm_cmpeq_epi8`). A full key comparison is then performed only for those tuples whose fingerprints match. This two-phase process prunes the search space within a home block, sharply reducing key comparisons and pointer traversals.

An important design consideration is the generation of these fingerprints. Extracting the fingerprint directly from the key's remainder (e.g., its most significant byte) would yield many collisions when keys share high-order bits. Therefore, we enlarge the quotient by one byte when applying the Feistel permutation in Section 4.1 and truncate the most significant byte for the fingerprint.

6.3 Structure and Operations

The layout of Flattened-TPHT is shown in Figure 4, for cases with 4 home tuples and 3 home tuples (after migration due to overflow).

The layout includes the following components:

- **Home array** ($n/4$ cache lines): Array of 64B home blocks. Each home block entry compacts fingerprints, tiny pointers, and quotiented key-value pairs together with metadata consisting of the number of fingerprints and tiny pointers, and concurrency control information.
- **Dereference table** ($\approx n/4$ slots by Section 6.4). This is unchanged from Chained-TPHT (Section 5.1), except that each entry is smaller because an additional byte is quotiented.

The procedure for inserting a tuple (k, v) into Flattened-TPHT is detailed in Algorithm 3. Query, update, and deletion operations are similar, using vectorized SIMD instructions to scan fingerprints and identify candidate entries before performing a full key comparison.

6.4 Failure Rate Analysis

While this flattened design improves cache performance, it introduces two new failure risks. First, each home block has a fixed size and must accommodate all the data it stores either as home tuples or as tiny pointers to overflow tuples. Second, the dereference table must be sized so that it can accommodate all overflow tuples, while maintaining high space utilization.

While bad events such as home block overflows and dereference-table exhaustion are possible, our probabilistic analysis verifies that these events do not occur with high probability across all workloads. Specifically, we prove that home blocks operate well

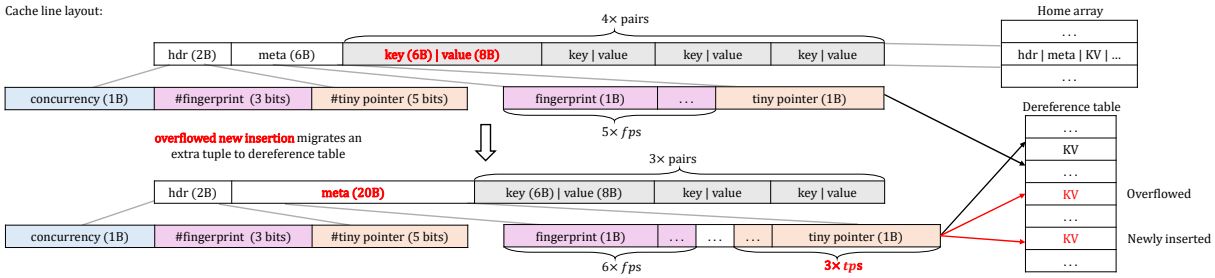


Figure 4: Flattened Data Layout of Flattened-TPHT

Algorithm 3 INSERT

Require: home array C, key $k = q_0^{\text{pre}} \circ r_0$, value v

$b \leftarrow h(r_0) \oplus q_0^{\text{pre}}$ ▷ Target home block index
$q_1^{\text{pre}} \leftarrow r_0 \div 2^{w-\ell-8}$; $r_1 \leftarrow r_0 \bmod 2^{w-\ell-8}$
$fp \leftarrow h(r_1) \oplus q_1^{\text{pre}}$ ▷ Fingerprint of the key
$L \leftarrow C[b]$
AddFingerprint(L, fp) ▷ Store fingerprint in the home block
if HasInCacheSpace(L) **then** ▷ Case 1: Space for in-cache KV
    AddInCacheKV(L, r_1, v)
else if HasPointerSpace(L) **then** ▷ Case 2: Space for a tiny pointer
    $e, tp \leftarrow$ Allocate$(L.tp.end())$; $e.k \leftarrow r_1$; $e.v \leftarrow v$
    AddTinyPointer(L, tp)
else ▷ Case 3: Evict an in-cache KV to make space
    $r^{\text{evict}}, v^{\text{evict}} \leftarrow$ EvictInCacheKV(L)
    $e_1, tp_1 \leftarrow$ Allocate$(L.tp.end())$; $e_1.k \leftarrow r^{\text{evict}}$; $e_1.v \leftarrow v^{\text{evict}}$
    AddTinyPointer(L, tp_1)
    $e_2, tp_2 \leftarrow$ Allocate$(L.tp.end())$; $e_2.k \leftarrow r_1$; $e_2.v \leftarrow v$
    AddTinyPointer(L, tp_2)

within their capacity limits w.h.p., and the dereference table can be conservatively sized to handle overflows without risk of exhaustion.

Home block capacity analysis. In particular, if sufficiently many tuples share a home block, even if they are all overflow tuples, there won't be enough space for their tiny pointers. We refer to this event as a ***hard overflow***. We now show that the probability of a hard overflow is negligible with a classical balls-and-bins analysis of n keys (balls) into $m = 2^{\ell-2}$ home blocks (bins).

Lemma 3 (Maximum Bin Size [76]). *Throw n balls into m bins i.u.d., where $m/\text{polylog}(m) \le n \ll m \log m$. Let M be the max load. Then, $M \le \frac{\log m}{\log\log m + \log m - \log n}(1 + o(1))$ w.h.p.*

Applying this lemma with n keys and $m = 2^{\ell-2}$ home blocks, we establish that the max load on any home block is at most $\frac{\ell}{\log \ell - 2}(1 + o(1))$ w.h.p. For any dataset with fewer than 2^{64} items, this is fewer than 24 elements. Each home block can hold up to 31 tiny pointers (considering metadata and fingerprints), so this is well within the capacity. Therefore, hard overflows are statistically negligible.

Dereference table sizing analysis. Let O denote the number of overflow tuples in an arbitrary home block. By linearity of expectation, the expected total number of overflow tuples is $(n/4)\mathbb{E}[O]$.

If a home block has x tuples, then $O = \max(0, x - 4 + \lfloor x/6 \rfloor)$. The R.V. x follows a binomial distribution, $x \sim \text{Binomial}(n, 4/n)$, which we approximate by Poisson(4) [67]. Using the fact that, for $x \sim \text{Poisson}(4)$, we have $\Pr[X = i] = 4^i e^{-4}/i!$, we can conclude that

$$\mathbb{E}[O] \approx \sum_{i=5}^{\infty} (i - 4 + \lfloor i/6 \rfloor) \frac{4^i e^{-4}}{i!} < 0.997,$$

which corresponds to an expected overflow ratio below 24.9%. This ratio improves with dataset size; when $n \ge 2^{16} = 65536$, this ratio decreases to less than 20.8%.

We now use the Azuma-Hoeffding inequality to establish concentration bounds around the expected overflow size.

Claim 4. *The overflow size of Flattened-TPHT exhibits strong concentration around its expected value w.h.p.*

PROOF. Let X_i be the home block identifier for the i-th tuple, and let $S = f(X_1, \ldots, X_n) = \sum_{i=1}^{n/4} O_i$ denote the total overflow size across all home blocks. Consider the Doob martingale $M_t = \mathbb{E}[S \mid X_1, \ldots, X_t]$ for $t = 0, \ldots, n$. Revealing one tuple can change the total overflow by at most two units, hence the martingale differences are bounded: $|M_t - M_{t-1}| \le 2$ almost surely. By Azuma-Hoeffding, for any $\lambda > 0$,

$$\Pr\big[\, S - \mathbb{E}[S] \ge \lambda \big] \le \exp\Big(-\frac{\lambda^2}{2n \cdot 2^2}\Big).$$

Choosing $\lambda = 2\sqrt{2n \ln n}$ gives a tail at most $1/n$. For practical dataset sizes (e.g., $n \ge 10^5$), this deviation is below 1% of the dataset size. Therefore, the dereference table size exhibits strong concentration around its expectation w.h.p. □

Adding up all the space overhead required for fault tolerance, the space usage of Flattened-TPHT for n tuples with w-bit keys and values is $2nw + 0.22n\big(2w - \log n\big)/(1 - \delta)$ bits, where $1 - \delta$ is the load factor of the dereference table. This analysis shows that Flattened-TPHT provides robust protection against overflow events while maintaining excellent space efficiency.

7 CONCURRENCY AND RESIZING

7.1 Concurrency

We use a fine-grained optimistic concurrency control protocol for highly parallel operations. The mechanism uses version numbers to ensure atomicity and consistency at the quotient-group granularity. For Flattened-TPHT (Figure 4), the version number is embedded in the header of each cache line, while for Chained-TPHT it is stored alongside the head-array pointers.

Specifically, we use ***sequence locks*** (seqlocks), a primitive common in e.g. the Linux kernel [83, 84]. Each group maintains a monotonically increasing integer version: even indicates no writer, odd indicates a write in progress. A writer acquires the lock by on the group atomically incrementing the version (even to odd), performs its updates, and releases by incrementing again (odd to even). Readers use optimistic validation: read version v_0; if v_0 is odd, retry; read the data; read version v_1; accept the read only if $v_0 = v_1$, otherwise retry. A similar mechanism applies to the dereference table, with version numbers on individual bins.

The version counter can wrap around after enough updates, risking the ABA problem. We use a configurable-width version counter to balance space and safety. In our experiments, an 8-bit counter suffices to make wraparound negligible for our workloads.

7.2 Resizing

We also provide a scalable resizing framework for TPHT. However, resizing is an option that can be enabled for certain applications, rather than a fallback when allocation failure occurs. Our theory and experiments (Section 6.4 and 8.6) show that failures occur extremely rarely at reasonable load factors. As in some prior hash-table designs, in the case of a failure, the system will return an error rather than initiating a resize [35, 72].

Our resizing framework avoids stop-the-world stalls through a key strategy: collaborative, in-flight resizing. Instead of being blocked, threads accessing a partition during its resize detect the operation and join as workers. This partition-based approach amortizes growth costs by resizing partitions individually when their load factor exceeds a tunable threshold. When a resize is triggered, an initiating thread coordinates the migration by dividing it into fine-grained strides, which are then collaboratively processed by the coordinator and any joining threads. This dynamic distribution of work significantly accelerates the process. The configurable load threshold and growth factor also enable a direct trade-off between space efficiency and operation latency.

An additional technique to optimize worst-case space efficiency is ***staggering***. Given a resizing factor c and base capacity b, we initialize k partitions with different capacities $b \cdot c^{i/k}$ for $i = 0, 1, \ldots, k-1$. This approach prevents the scenario where nearly all partitions resize simultaneously, which would result in significant worst-case unused space. With a doubling resizing factor and 0.7 resizing threshold, staggering achieves a worst-case space efficiency of 43.9%, compared to 35.6% for native partitioning.

8 EVALUATION

We evaluate the performance of tiny pointer hash tables, Chained-TPHT and Flattened-TPHT, against state-of-the-art general hash tables, Cuckoo [35], IcebergHT [72], Junction [75], and TBB [41], as well as strong compact hash tables, Compact Bucketing [48], Cleary [23], and Layered [74]. libcuckoo implements classical cuckoo hashing with high-performance concurrency control. IcebergHT is a cache-aware hash table that leverages the recent iceberg hashing lemma for improved performance and memory efficiency. Junction provides an efficient implementation of canonical linear probing. TBB is a production separate-chaining hash table from Intel TBB. We chose these to cover the major families: separate chaining (TBB), cuckoo (Cuckoo), linear probing (Junction), and cache-aware design (IcebergHT). For compact hash tables, Bucket and Group are bucketing variants, Cleary^{plain} uses bidirectional linear probing, Layered^{plain} builds layered design upon linear probing, and Cleary^{sparse} and Layered^{sparse} are sparse variants.

Our evaluation addresses the following key questions:

(1) Do TPHTs deliver leading throughput against the baselines? (Section 8.2)
(2) Where do they sit on the speed-space Pareto frontier: how much memory is saved at equal throughput, and what throughput is retained at fixed space efficiency? (Section 8.3)
(3) How does the performance and space usage of Chained-TPHT compare to prior compact hash tables? (Section 8.3.1)
(4) How robust is their performance under scale—with increasing dataset size and thread count? (Section 8.4)
(5) How effective is our resizing scheme? Does it avoid blocking the world during resizing? (Section 8.5)
(6) Do the core primitives meet their theoretical promises: high dereference-table load factors without failures and near-Poisson occupancy from the hash family under diverse key distributions? (Section 8.6, Section 8.7)

8.1 Experimental Setup

Environment. All experiments were conducted on a single-socket x86_64 server running Linux kernel 5.15.0–151-generic. The machine is equipped with an Intel Xeon Silver 4314 CPU (16 cores, 32 threads; 2.40 GHz base, up to 3.40 GHz) and 128 GiB of RAM, with a 24 MiB shared L3 cache and a single NUMA node.

Datasets. Following previous work [22, 72], we use the hash-table variant of YCSB [25, 90] to evaluate the real-world performance of the proposed hash tables. YCSB proceeds in two phases: a ***load phase*** consisting of inserts and a ***run phase*** consisting of a mix of positive reads and inserts. Following prior work, we use run phases A, B, and C, which consist of 50% reads/50% inserts, 90% reads/10% inserts and 100% reads, respectively. We also define phases A$^-$, B$^-$, and C$^-$, which mirror A, B, and C but use negative rather than positive queries, and phase X, which performs deletions. In the load phase, we insert 64M tuples into the hash table. In the run phases A, A$^-$, B, and B$^-$, we insert the same number of tuples and change the number of queries accordingly. For C, C$^-$, we query 64M tuples, and for X, we delete 64M tuples. We also run microbenchmarks over a dataset consisting of uniformly distributed keys.

Scope. We evaluate all hash tables with 64b keys and 64b values. Our benchmarks do not include duplicate keys.[6]

8.2 YCSB

To measure overall performance, we run YCSB with 16 threads. We initialize all tables with similar initial capacity (except TBB, which allocates lazily) and terminate at 70% space efficiency (50% for Run X), defined as the ratio of the inserted key-value payload size to

[6]Duplicate-key support is compatible with TPHT designs. Chained-TPHT requires no modification to support duplicate keys, since the hash table key is distinct from the identifier used for tiny-pointer dereference. Flattened-TPHT can be extended to support duplicate keys with light structural changes.

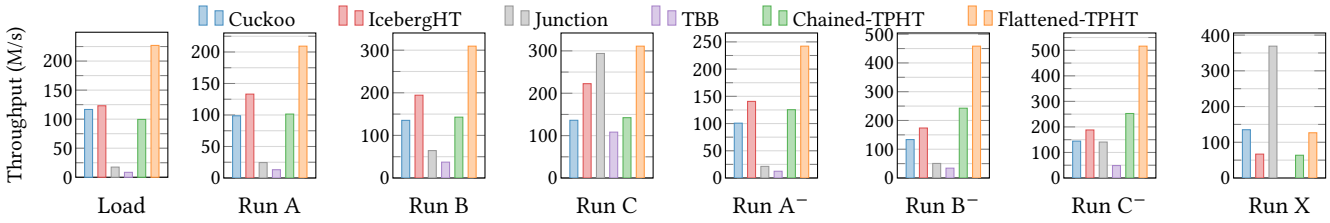
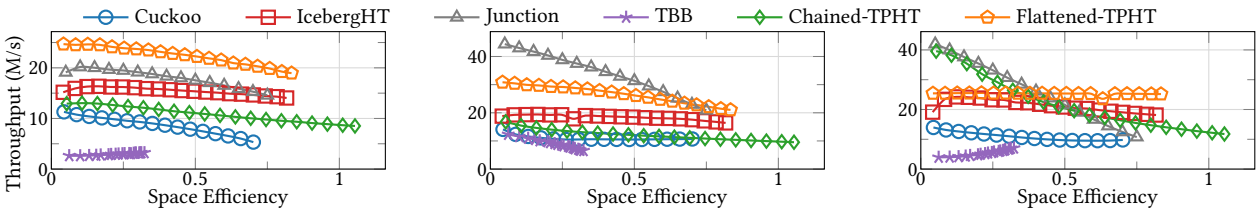


Figure 5: Performance of hash tables on YCSB workloads with 16 threads. (Throughput is Million ops/second).



(a) Insertion **(b) Positive Query** **(c) Negative Query**

Figure 6: Throughput-space efficiency tradeoff across insertion and query workloads. Each curve shows how throughput and space efficiency vary with load factor, with rightmost points indicating maximum achievable space efficiency.

the total allocated memory footprint. The throughput is shown in Figure 5.

Flattened-TPHT has the highest average throughput with 226.9 M ops/s for load phase and 310.4 M ops/s for run phases, which is 1.84 × and 1.94 × of the fastest baseline, respectively. Chained-TPHT maintains competitive performance: 99.5 M ops/s load phase and 153 M ops/s run phase, which are 80.9% and 95.7% of the fastest baseline, respectively.

Cuckoo maintains balanced performance across all workloads (116.7 M ops/s load phase, 126.3 M ops/s run phase). This stems from the cuckoo design: each key corresponds to two locations, and probes at most two cachelines per query, leading to predictable access patterns regardless of workload mix. IcebergHT uses a front yard/backyard design from recent theory literature [9]. The fingerprint layer of IcebergHT brings both pros and cons: it adds an extra level of indirection, but also allows for filtering out non-matching keys early. Its run-phase throughput is 26.6% higher than Cuckoo. Junction is a linear probing hash table with strong positive query performance, 293.9 M ops/s throughput on Run C, but its complicated concurrency control mechanism for insertions as well as the global load monitoring mechanism leads to contention and poor performance on insertion and mixed workloads. TBB maintains stable but lower throughput compared to other baselines. This likely stems from its separate-chaining design, which provides less cache locality, and RAII accessors that hold locks for the duration of access. We note that TBB lacks support for safe multi-threaded deletions, so there are no results for Run X.

8.3 Speed/Space Tradeoff

We define ***space efficiency*** as the ratio of the raw key-value payload size to the maximum allocated memory footprint during execution. Initializing each hash table's capacity to 16M tuples and inserting data in 5% increments, we record the memory footprint until reaching 99% capacity or a failure or resizing occurs. We pin to one physical core to ensure stable throughput and memory footprint measurements. Figures 6a, 6b, and 6c show the throughput for insertion, positive query, and negative query, respectively.

Chained-TPHT demonstrates the highest space efficiency of 105.4%, reducing memory usage by 22.5% to 69.3% compared with non-tiny-pointer-based baselines. Flattened-TPHT achieves the second-highest space efficiency, 83.4%, and at 50% space efficiency is 30.4% faster on insertion and 24.1% faster on negative query than the best baseline; the advantage grows above 50% space efficiency. It has the second highest positive query at 50% space efficiency, 88.5% of the best baseline, and becomes the fastest when space efficiency is above 70%. Junction (linear probing) excels at low load: at a 0.05 load factor it achieves 1.4× Flattened-TPHT's positive-query throughput. Its performance, however, drops sharply—by 48.3%—as the load factor rises to 0.7. For comparison, Chained-TPHT, Flattened-TPHT, and Cuckoo drop by 35.3%, 19.8%, and 24.8%, respectively, whereas IcebergHT remains flatter with only 3.9% decline. This stems from IcebergHT's small fallback backyard, which becomes increasingly cache-friendly as fallbacks accumulate, partially offsetting other overheads. With TBB, insertion and negative-query throughput increases as more data are loaded. TBB allocates buckets lazily upon first pointer dereference; loading more data forces allocation of more initial buckets, which raises throughput.

8.3.1 Comparison with compact hash tables. In Figure 7, we conduct the same experiments as Section 8.3 with compact hash tables. Unless a specific operation is indicated, reported throughput is averaged over all data points and operations (insert, query). Bucketing methods (Bucket and Group) [48] use frequent bucket reallocations to maintain high space efficiency (up to 110.3%). However, this results in very slow insertions (Chained-TPHT is 9.8×

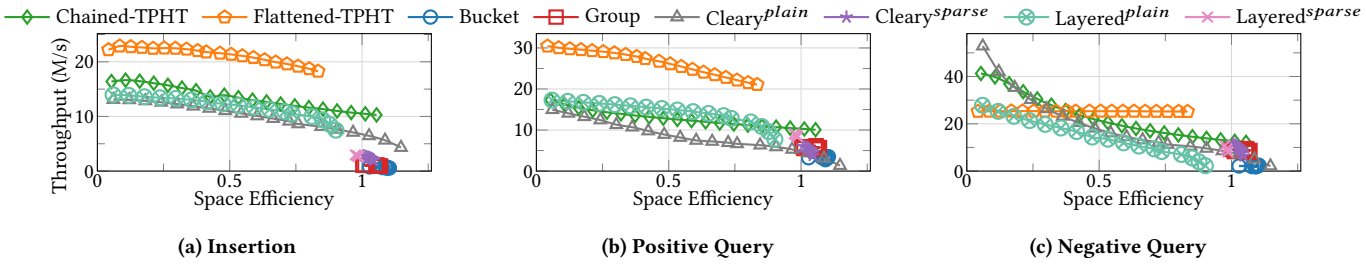


Figure 7: Throughput-space efficiency tradeoff for compact hash tables.

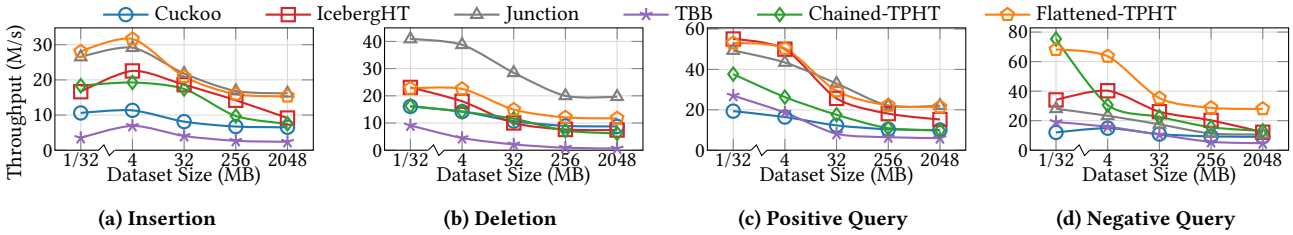


Figure 8: Performance scaling analysis for hash tables with increasing dataset size, single-threaded.

faster). Cleary^{plain} [23] uses bidirectional linear probing and uses an extra displacement field for each slot to recover the key hash. It achieves high space efficiency (114.7%), but still lower throughput than Chained-TPHT (by 62.5%). Using Cleary^{sparse} with its sparse layout does not improve either space efficiency or performance. Layered^{plain} [74] uses a similar linear probing strategy and stores displacement information across multiple levels. However, its space efficiency is 90.1%, which means its natural comparison point is Flattened-TPHT rather than Chained-TPHT, and Flattened-TPHT has 2.44× its throughput. Layered^{sparse} with sparse layout sacrifices 64.1% throughput for an 8.5% gain in space efficiency.

8.4 Scalability

8.4.1 Scaling with dataset size. Figure 8 shows single-threaded throughput across dataset sizes from 2^{11} to 2^{27} on a uniformly distributed microbenchmark. When the dataset fits entirely in L1 cache (2^{11} entries), Flattened-TPHT and Chained-TPHT achieve average throughputs of 43 and 36.9 M ops/s across all operations—1.19× and 1.02× the best baseline, respectively. In particular, Chained-TPHT dominates negative queries at 75.3 M ops/s, as its compact head array keeps lookups L1-resident. As the dataset grows beyond cache capacity, throughput decreases across all systems (except for the cold start points); Flattened-TPHT sustains its lead even at the largest scale, averaging 19.1 M ops/s at 2 GiB—1.11× the best baseline. However, Junction's deletion outperforms all other systems here because a linear probing table treats deletions similarly to positive queries, and since this workload is a single deletion pass, the tombstone mechanism is not stressed.

8.4.2 Scaling with threads. We evaluate scalability with 1–16 threads on a microbenchmark with 64M keys. Most hash tables exhibit nearly linear scaling behavior across all four operation types. Specifically, the scaling ratios for Chained-TPHT and Flattened-TPHT are 0.908 and 0.925, respectively.

8.5 Resizing

8.5.1 Throughput. Figure 9 analyzes the efficacy of our resizing scheme from Section 7.2. Unlike Section 8.2, each hash table resizes once during the load phase and once during the run phase (when insertion operations are present). Flattened-TPHT achieves 245M ops/s average throughput, fastest among all methods. However, TPHTs' resizing work is charged to insertion operations, the load phase throughput decreases by 76.2% compared to non-resizing tests (Section 8.2). On the contrary, IcebergHT exhibits the smallest performance degradation, with a 35.9% throughput decrease due to its proprietary in-place resizing optimization.

8.5.2 Progressive resizing throughput. Figure 10 shows the throughput on 100 consecutive tumbling windows over multiple resizings. For Flattened-TPHT, throughput drops by 27.4% from immediately after a resize to just before the next resize (after further inserts). Chained-TPHT's throughput tends to be more stable, consistent with evaluation in Section 8.3. IcebergHT's low-performance period is prolonged as dataset grows, which we attribute to its progressive resizing strategy. Moreover, despite the appearance in the figure, staggered resizing in Chained-TPHT affects overall throughput by less than 10%. Meanwhile, with resizing enabled under YCSB Run A, the 99th-percentile insertion latency of Flattened-TPHT and Chained-TPHT is 541 ns and 778 ns, respectively—nearly an order of magnitude below competitors (e.g., Cuckoo at 2.7 μs).

8.5.3 Memory footprint. We initialize the hash tables with 2^{24} capacity and set the double-sizing resizing threshold to 0.7 (except for Cuckoo, which resizes when insertion fails). Then we continuously insert 0.6×2^{27} tuples and show the memory footprint (`VmHWM`) of the hash tables during the whole resizing process in Figure 11.

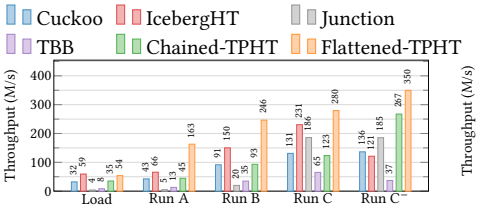


Figure 9: Performance of hash tables on YCSB workloads with 16 threads (resizing enabled).

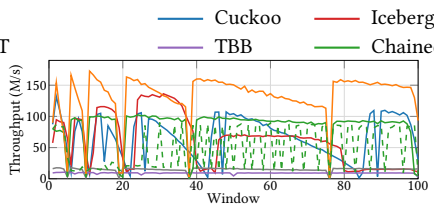


Figure 10: Sliding window insertion throughput across windows. The dashed line indicates staggering enabled.

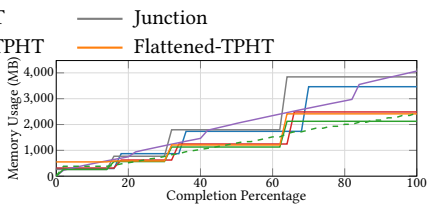


Figure 11: Memory usage during resizing. The dashed line indicates staggering enabled.

Attributed to the staggering technique in Section 7.2, the footprint of Chained-TPHT grows steadily rather than in large steps, unlike the other baselines. On the other hand, although Flattened-TPHT doesn't use staggering, it still has a small memory footprint because of its compact design. Chained-TPHT's worst-case space efficiency with staggering is 46.7%, while IcebergHT with the best worst-case space efficiency among baselines is only 32.6%.

8.6 Dereference Table Capacity

In order to show that the dereference table implementation (Section 3.2) can support high load factors under a sustained workload, we instantiate dereference tables with different bin size and perform sequences of operations. We either perform random insertions until allocation failure or alternating random insertions and deletions (100× the table capacity) at a fixed load factor. The results are shown in Figure 12. As expected, the supported load factor increases with larger bin sizes. When the bin size reaches $2^7 - 1$, the dereference table supports 8-bit tiny pointers at load factors 98.2% and 95% for insertion-only and alternating workloads, respectively.

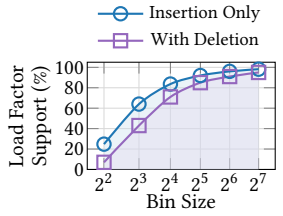


Figure 12: Load factor supported by dereference tables (Section 3.2).

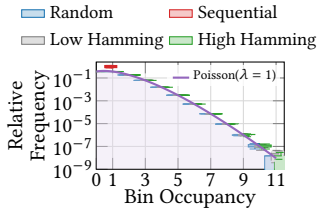


Figure 13: Bin occupancy compared with smoothed Poisson ($\lambda = 1$).

8.7 Hash Function Analysis

To evaluate the quality of the hash function family designed in Section 4.1, we examine the relative frequency of bin occupancy. We use random keys, sequential keys, low Hamming weight keys, or high Hamming weight keys, and hash 64 M keys to 64 M bins 100 times. The results are shown in Figure 13.

Compared with the Poisson distribution ($\lambda = 1$), the random, low and high Hamming weight keys all closely approximate the Poisson distribution, as expected. Sequential keys on the other hand are completely evenly distributed, which is also as expected. It demonstrates lack of independence, and the negative association of this dependence (see Section 4.1). Thus though the hash function is not independent, it is only dependent in that keys have better than random spread.

9 RELATED WORK

Practical hash tables. A long line of work targets the latency-space tradeoff with open-addressing variants [19, 22, 43, 45, 63, 64, 89] and hardware-aware implementations using SIMD, FPGAs, GPUs, PMEM, cache-line layouts, and prefetching [2, 17, 26, 29, 38, 39, 42, 44, 49, 53, 54, 60, 62, 70, 86, 88, 93]. These designs still pay pointer overheads or per-entry metadata; TPHT removes both with tiny pointers and quotienting while keeping operations cache-friendly. Hash tables in this style underpin key-value stores [21, 55, 56], caches [30, 65], and joins [57, 82].

Theory-adjacent work. Closer to our techniques, theoretical results on succinct dictionaries and perfect hashing [8, 12, 27, 77], open addressing [13, 31], quotient-based filters [28, 73], balls-into-bins and balanced allocations [52, 58, 76], and history independence [5, 15, 69] approach the information-theoretic optimum but typically rely on multi-level structures considered impractical [16, 33]. Chained-TPHT instantiates these ideas as a simple, practical succinct hash table; Flattened-TPHT carries them into an open-addressing layout tuned for modern hardware.

10 CONCLUSION

We introduced Tiny Pointer Hash Tables, showing that ***tiny pointers*** and ***key quotienting***—two ideas from theory—can be engineered into systems-ready hash tables. Chained-TPHT prioritizes memory use and is, to our knowledge, the first practical succinct hash table. Flattened-TPHT prioritizes latency, placing the common case within a single cache miss and achieving 83.4% space efficiency with up to 89.3% higher throughput than strong baselines. Together, these advances move the latency-space Pareto frontier forward for hash tables. Beyond raw numbers, TPHT supports deletions, online resizing without global pauses, and clean integration with 64-bit keys/values—making the approach directly usable in systems. In future work, we see opportunities to adapt the tiny-pointer/dereference-table machinery to other pointer-heavy data structures and to explore adaptive quotienting and layout choices for different workloads.

ACKNOWLEDGMENTS

Alex Conway and William Kuszmaul were supported by NSF Award CNS-2504470; William Kuszmaul was also supported by Jane Street.

REFERENCES

[1] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. MorphoSys: automatic physical design metamorphosis for distributed database systems. ***Proceedings of the VLDB Endowment*** 13, 13 (2020), 3573–3587. https://doi.org/10.14778/3424573.3424578

[2] Abseil. 2024. Swiss Tables Design Notes. https://abseil.io/about/design/swisstables. Accessed: 2026-05-20.

[3] Ahmed Alquraan, Alex Kogan, Virendra J. Marathe, and Samer Al-Kiswany. 2020. Scalable, near-zero loss disaster recovery for distributed data stores. ***Proceedings of the VLDB Endowment*** 13, 9 (2020), 1429–1442. https://doi.org/10.14778/3397230.3397239

[4] Yuriy Arbitman, Moni Naor, and Gil Segev. 2010. Backyard Cuckoo Hashing: Constant Worst-Case Operations with a Succinct Representation. In ***2010 IEEE 51st Annual Symposium on Foundations of Computer Science (FOCS)***. IEEE, 787–796. https://doi.org/10.1109/FOCS.2010.80

[5] Hagit Attiya, Michael A. Bender, Martín Farach-Colton, Rotem Oshman, and Noa Schiller. 2025. History-Independent Concurrent Hash Tables. arXiv:2503.21016 [cs.DC]

[6] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. 1999. Balanced Allocations. ***SIAM J. Comput.*** 29, 1 (1999), 180–200. https://doi.org/10.1137/S0097539795288490

[7] Nikhil Bansal and William Kuszmaul. 2022. Balanced Allocations: The Heavily Loaded Case with Deletions. arXiv:2205.06558 [cs.DS] https://arxiv.org/abs/2205.06558 Accessed: 2026-05-20.

[8] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. 2009. Hash, Displace, and Compress. In ***Algorithms – ESA 2009 (Lecture Notes in Computer Science, Vol. 5757)***. Springer, 682–693. https://doi.org/10.1007/978-3-642-04128-0_61

[9] Michael A. Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. 2023. Iceberg Hashing: Optimizing Many Hash-Table Criteria at Once. ***J. ACM*** 70, 6, Article 40 (Nov. 2023), 51 pages. https://doi.org/10.1145/3625817

[10] Michael A. Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, and Guido Tagliavini. 2023. Tiny Pointers. In ***Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023***, Nikhil Bansal and Viswanath Nagarajan (Eds.). SIAM, 477–508. https://doi.org/10.1137/1.9781611977554.CH21

[11] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't thrash: how to cache your hash on flash. ***Proc. VLDB Endow.*** 5, 11 (July 2012), 1627–1637. https://doi.org/10.14778/2350229.2350275

[12] Michael A. Bender, Martín Farach-Colton, John Kuszmaul, and William Kuszmaul. 2024. Modern Hashing Made Simple. In ***2024 Symposium on Simplicity in Algorithms (SOSA)***. SIAM, 363–373. https://doi.org/10.1137/1.9781611977936.33

[13] Michael A. Bender, Martín Farach-Colton, John Kuszmaul, William Kuszmaul, and Mingmou Liu. 2022. On the optimal time/space tradeoff for hash tables. In ***Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*** (Rome, Italy) ***(STOC 2022)***. Association for Computing Machinery, New York, NY, USA, 1284–1297. https://doi.org/10.1145/3519935.3519969

[14] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. 2006. Balanced Allocations: The Heavily Loaded Case. ***SIAM J. Comput.*** 35, 6 (June 2006), 1350–1385. https://doi.org/10.1137/S009753970444435X

[15] Guy E. Blelloch and Daniel Golovin. 2007. Strongly History-Independent Hashing with Applications. In ***48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007)***. IEEE, 272–282. https://doi.org/10.1109/FOCS.2007.36

[16] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. 2007. Simple and Space-Efficient Minimal Perfect Hash Functions. In ***Algorithms and Data Structures (WADS 2007) (Lecture Notes in Computer Science, Vol. 4619)***. Springer-Verlag, Berlin, Heidelberg, 139–150. https://doi.org/10.1007/978-3-540-73951-7_13

[17] Maximilian Böther, Lawrence Benson, Ana Klimović, and Tilmann Rabl. 2023. Analyzing Vectorized Hash Tables Across CPU Architectures. ***Proceedings of the VLDB Endowment*** 16, 11 (2023), 2755–2768. https://doi.org/10.14778/3611479.3611485

[18] Mark Braverman and William Kuszmaul. 2024. Tight Analyses of Ordered and Unordered Linear Probing. In ***2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS)***. IEEE, 606–635. https://doi.org/10.1109/FOCS61266.2024.00046

[19] Alex D. Breslow, Dong Ping Zhang, Joseph L. Greathouse, Nuwan Jayasena, and Dean M. Tullsen. 2016. Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing. In ***2016 USENIX Annual Technical Conference (USENIX ATC 16)***. USENIX Association, Denver, CO, 281–294. https://www.usenix.org/conference/atc16/technical-sessions/presentation/breslow

[20] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. 1998. Min-wise independent permutations. In ***Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC '98)***. ACM, 327–336. https://doi.org/10.1145/276698.276781

[21] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In ***Proceedings of the 2018 International Conference on Management of Data*** (Houston, TX, USA) ***(SIGMOD '18)***. Association for Computing Machinery, New York, NY, USA, 275–290. https://doi.org/10.1145/3183713.3196898

[22] Yuvaraj Chesetti, Benwei Shi, Jeff M. Phillips, and Prashant Pandey. 2025. Zombie Hashing: Reanimating Tombstones in Graveyard. ***Proc. ACM Manag. Data*** 3, 3, Article 236 (June 2025), 27 pages. https://doi.org/10.1145/3725424

[23] J. G. Cleary. 1984. Compact Hash Tables Using Bidirectional Linear Probing. ***IEEE Trans. Comput.*** 33, 9 (Sept. 1984), 828–834. https://doi.org/10.1109/TC.1984.1676499

[24] Yann Collet. 2024. xxHash: Extremely fast non-cryptographic hash algorithm. https://xxhash.com/. Accessed: 2026-05-20.

[25] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In ***Proceedings of the 1st ACM Symposium on Cloud Computing*** (Indianapolis, Indiana, USA) ***(SoCC '10)***. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[26] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In ***Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*** (Istanbul, Turkey) ***(ASPLOS '15)***. Association for Computing Machinery, New York, NY, USA, 631–644. https://doi.org/10.1145/2694344.2694359

[27] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. 1994. Dynamic Perfect Hashing: Upper and Lower Bounds. ***SIAM J. Comput.*** 23, 4 (1994), 738–761. https://doi.org/10.1137/S0097539791194094

[28] Tomer Even, Guy Even, and Adam Morrison. 2022. Prefix filter: practically and theoretically better than bloom. ***Proc. VLDB Endow.*** 15, 7 (March 2022), 1311–1323. https://doi.org/10.14778/3523210.3523211

[29] Facebook. 2023. F14 Hash Tables: A High-Performance C++ Hash Table Family in Folly. GitHub repository, https://github.com/facebook/folly/blob/main/folly/container/F14.md. Accessed: 2026-05-20.

[30] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In ***10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)***. USENIX Association, Lombard, IL, 371–384. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan

[31] Martin Farach-Colton, Andrew Krapivin, and William Kuszmaul. 2025. Optimal Bounds for Open Addressing Without Reordering. arXiv:2501.02305 [cs.DS] https://arxiv.org/abs/2501.02305 Accessed: 2026-05-20.

[32] Philippe Flajolet, Patricio Poblete, and Alfredo Viola. 1998. On the analysis of linear probing hashing. ***Algorithmica*** 22, 4 (1998), 490–515. https://doi.org/10.1007/PL00009236

[33] Michael L. Fredman, János Komlós, and Endre Szemerédi. 1984. Storing a Sparse Table with O(1) Worst Case Access Time. ***J. ACM*** 31, 3 (June 1984), 538–544. https://doi.org/10.1145/828.1884

[34] Rémi Géraud, Marius Lombard-Platet, and David Naccache. 2019. Quotient hash tables: efficiently detecting duplicates in streaming data. In ***Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*** (Limassol, Cyprus) ***(SAC '19)***. Association for Computing Machinery, New York, NY, USA, 582–589. https://doi.org/10.1145/3297280.3297335

[35] Manu Goyal, Bin Fan, Xiaozhou Li, David G. Andersen, and Michael Kaminsky. 2013. libcuckoo. https://github.com/efficient/libcuckoo Accessed: 2025-10-21.

[36] Leo J. Guibas and Endre Szemerédi. 1976. The analysis of double hashing (Extended Abstract). In ***Proceedings of the Eighth Annual ACM Symposium on Theory of Computing (STOC '76)***. ACM, 187–191. https://doi.org/10.1145/800113.803647

[37] Steef Hegeman, Daan Wöltgens, Anton Wijs, and Alfons Laarman. 2024. Compact Parallel Hash Tables on the GPU. In ***Euro-Par 2024: Parallel Processing: 30th European Conference on Parallel and Distributed Processing, Madrid, Spain, August 26-30, 2024, Proceedings, Part II*** (Madrid, Spain). Springer-Verlag, Berlin, Heidelberg, 226–241. https://doi.org/10.1007/978-3-031-69766-1_16

[38] Daokun Hu, Zhiwen Chen, Wenkui Che, Jianhua Sun, and Hao Chen. 2022. Halo: A Hybrid PMem-DRAM Persistent Hash Index with Fast Recovery. In ***Proceedings of the 2022 International Conference on Management of Data*** (Philadelphia, PA, USA) ***(SIGMOD '22)***. Association for Computing Machinery, New York, NY, USA, 1049–1063. https://doi.org/10.1145/3514221.3517884

[39] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. 2021. Persistent memory hash indexes: an experimental evaluation. ***Proc. VLDB Endow.*** 14, 5 (Jan. 2021), 785–798. https://doi.org/10.14778/3446095.3446101

[40] Yu Hua, Hong Jiang, and Dan Feng. 2014. FAST: Near Real-Time Searchable Data Analytics for the Cloud. In ***SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis***. IEEE, 754–765. https://doi.org/10.1109/SC.2014.67

[41] Intel Corporation. 2021. Intel oneAPI Threading Building Blocks (oneTBB). https://www.intel.com/content/www/us/en/docs/onetbb/developer-guide-api-reference/2021-9/concurrent-hash-map.html. Accessed: 2025-10-21.

[42] Zsolt István, Gustavo Alonso, Michaela Blott, and Kees Vissers. 2015. A Hash Table for Line-Rate Data Processing. ***ACM Trans. Reconfigurable Technol. Syst.*** 8, 2, Article 13 (March 2015), 15 pages. https://doi.org/10.1145/2629582

[43] Aarati Kakaraparthy, Jignesh M. Patel, Brian P. Kroth, and Kwanghyun Park. 2022. VIP hashing: adapting to skew in popularity of data on the fly. ***Proc. VLDB Endow.*** 15, 10 (June 2022), 1978–1990. https://doi.org/10.14778/3547305.3547306

[44] Antonios Katsarakis, Vasilis Gavrielatos, and Nikos Ntarmos. 2024. DLHT: A Non-blocking Resizable Hashtable with Fast Deletes and Memory-Awareness. In ***Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24)***. ACM, 186–199. https://doi.org/10.1145/3625549.3658682

[45] Robert Kelly, Barak A. Pearlmutter, and Phil Maguire. 2018. Concurrent Robin Hood Hashing. arXiv:1809.04339 [cs.DC] https://arxiv.org/abs/1809.04339 Accessed: 2026-05-20.

[46] Donald E. Knuth. 1998. ***The Art of Computer Programming, Volume 3: Sorting and Searching*** (2nd ed.). Addison-Wesley Professional, Reading, MA.

[47] Dominik Köppl. 2019. ***Separate Chaining Meets Compact Hashing***. IPSJ SIG Technical Report 2019-AL-173, No. 5. Information Processing Society of Japan. 1–11 pages. https://ipsj.ixsq.nii.ac.jp/api/records/195522 Accessed: 2026-05-20.

[48] Dominik Köppl, Simon J. Puglisi, and Rajeev Raman. 2022. Fast and Simple Compact Hashing via Bucketing. ***Algorithmica*** 84, 9 (Sept. 2022), 2735–2766. https://doi.org/10.1007/s00453-022-00996-y

[49] Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders. 2023. PaCHash: Packed and Compressed Hash Tables. In ***2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)***. SIAM, 162–175. https://doi.org/10.1137/1.9781611977561.ch14

[50] William Kuszmaul and Zoe Xi. 2024. Towards an Analysis of Quadratic Probing. In ***51st International Colloquium on Automata, Languages, and Programming (ICALP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 297)***. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 103:1–103:19. https://doi.org/10.4230/LIPIcs.ICALP.2024.103

[51] Shir Landau-Feibish, Zaoxing Liu, and Jennifer Rexford. 2025. Compact Data Structures for Network Telemetry. ***ACM Comput. Surv.*** 57, 8, Article 191 (March 2025), 31 pages. https://doi.org/10.1145/3716819

[52] Christoph Lenzen, Merav Parter, and Eylon Yogev. 2019. Parallel Balanced Allocations: The Heavily Loaded Case. In ***The 31st ACM Symposium on Parallelism in Algorithms and Architectures*** (Phoenix, AZ, USA) ***(SPAA '19)***. Association for Computing Machinery, New York, NY, USA, 313–322. https://doi.org/10.1145/3323165.3323203

[53] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In ***Proceedings of the 26th Symposium on Operating Systems Principles*** (Shanghai, China) ***(SOSP '17)***. Association for Computing Machinery, New York, NY, USA, 137–152. https://doi.org/10.1145/3132747.3132756

[54] Zexuan Li and Kaixin Huang. 2024. A read-efficient and write-optimized hash table for Intel Optane DC Persistent Memory. ***Future Generation Computer Systems*** 161 (2024), 49–65. https://doi.org/10.1016/j.future.2024.06.028

[55] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: a memory-efficient, high-performance key-value store. In ***Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*** (Cascais, Portugal) ***(SOSP '11)***. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2043556.2043558

[56] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: a holistic approach to fast in-memory key-value storage. In ***Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*** (Seattle, WA) ***(NSDI'14)***. USENIX Association, USA, 429–444.

[57] Ming-Ling Lo and Chinya V. Ravishankar. 1996. Spatial hash-joins. In ***Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*** (Montreal, Quebec, Canada) ***(SIGMOD '96)***. Association for Computing Machinery, New York, NY, USA, 247–258. https://doi.org/10.1145/233269.233337

[58] Dimitrios Los, Thomas Sauerwald, and John Sylvester. 2022. Balanced Allocations: Caching and Packing, Twinning and Thinning. In ***Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)***. SIAM, 1847–1874. https://doi.org/10.1137/1.9781611977073.74

[59] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: scalable hashing on persistent memory. ***Proc. VLDB Endow.*** 13, 8 (April 2020), 1147–1161. https://doi.org/10.14778/3389133.3389134

[60] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2021. Scaling Dynamic Hash Tables on Real Persistent Memory. ***SIGMOD Rec.*** 50, 1 (June 2021), 87–94. https://doi.org/10.1145/3471485.3471506

[61] Michael Luby and Charles Rackoff. 1988. How to Construct Pseudorandom Permutations from Pseudorandom Functions. ***SIAM J. Comput.*** 17, 2 (1988), 373–386. https://doi.org/10.1137/0217022

[62] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects. In ***Proceedings of the 2022 International Conference on Management of Data*** (Philadelphia, PA, USA) ***(SIGMOD '22)***. Association for Computing Machinery, New York, NY, USA, 1017–1032. https://doi.org/10.1145/3514221.3517911

[63] Tobias Maier, Peter Sanders, and Roman Dementiev. 2019. Concurrent Hash Tables: Fast and General(?)! ***ACM Trans. Parallel Comput.*** 5, 4, Article 16 (Feb. 2019), 32 pages. https://doi.org/10.1145/3309206

[64] Tobias Maier, Peter Sanders, and Stefan Walzer. 2019. Dynamic Space Efficient Hashing. ***Algorithmica*** 81, 8 (Aug. 2019), 3162–3185. https://doi.org/10.1007/s00453-019-00572-x

[65] Memcached Project. 2009. Memcached. https://memcached.org/. Accessed: 2026-05-20.

[66] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. 1996. ***Handbook of Applied Cryptography*** (1st ed.). CRC Press, Boca Raton, FL.

[67] Michael Mitzenmacher and Eli Upfal. 2005. ***Probability and Computing: Randomized Algorithms and Probabilistic Analysis***. Cambridge University Press, Cambridge, UK.

[68] André Müller, Christian Hundt, Andreas Hildebrandt, Thomas Hankeln, and Bertil Schmidt. 2017. MetaCache: context-aware classification of metagenomic reads using minhashing. ***Bioinformatics*** 33, 23 (2017), 3740–3748. https://doi.org/10.1093/bioinformatics/btx520

[69] Moni Naor and Vanessa Teague. 2001. Anti-persistence: history independent data structures. In ***Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing (STOC '01)***. ACM, 492–501. https://doi.org/10.1145/380752.380844

[70] Vikram Narayanan, David Detweiler, Tianjiao Huang, and Anton Burtsev. 2023. DRAMHiT: A Hash Table Architected for the Speed of DRAM. In ***Proceedings of the Eighteenth European Conference on Computer Systems*** (Rome, Italy) ***(EuroSys '23)***. Association for Computing Machinery, New York, NY, USA, 817–834. https://doi.org/10.1145/3552326.3587457

[71] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. ***Journal of Algorithms*** 51, 2 (2004), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002

[72] Prashant Pandey, Michael A. Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, Guido Tagliavini, and Rob Johnson. 2023. IcebergHT: High Performance Hash Tables Through Stability and Low Associativity. ***Proc. ACM Manag. Data*** 1, 1, Article 47 (May 2023), 26 pages. https://doi.org/10.1145/3588727

[73] Prashant Pandey, Alex Conway, Joe Durie, Michael A. Bender, Martin Farach-Colton, and Rob Johnson. 2021. Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design. In ***Proceedings of the 2021 International Conference on Management of Data*** (Virtual Event, China) ***(SIGMOD '21)***. Association for Computing Machinery, New York, NY, USA, 1386–1399. https://doi.org/10.1145/3448016.3452841

[74] Andreas Poyias, Simon J. Puglisi, and Rajeev Raman. 2017. m-Bonsai: a Practical Compact Dynamic Trie. arXiv:1704.05682 [cs.DS] https://arxiv.org/abs/1704.05682 Accessed: 2026-05-20.

[75] Jeff Preshing. 2024. Junction: Concurrent data structures in C++. GitHub repository, https://github.com/preshing/junction. Accessed: 2026-05-20.

[76] Martin Raab and Angelika Steger. 1998. "Balls into Bins" — A Simple and Tight Analysis. In ***Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM '98) (RANDOM '98)***. Springer-Verlag, Berlin, Heidelberg, 159–170. https://doi.org/10.1007/3-540-49543-6_13

[77] Rajeev Raman and Satti Srinivasa Rao. 2003. Succinct Dynamic Dictionaries and Trees. In ***Automata, Languages and Programming (ICALP 2003) (Lecture Notes in Computer Science, Vol. 2719)***. Springer, 357–368. https://doi.org/10.1007/3-540-45061-0_30

[78] Christian Rodriguez. 2025. TinyPointers. https://github.com/rodrigch18/TinyPointers. GitHub repository, accessed 2026-02-16.

[79] Ori Shalev and Nir Shavit. 2006. Split-ordered lists: Lock-free extensible hash tables. ***J. ACM*** 53, 3 (May 2006), 379–405. https://doi.org/10.1145/1147954.1147958

[80] Alex C. Snoeren, Craig Partridge, Luis A. Sanchez, Christine E. Jones, Fabrice Tchakountio, Stephen T. Kent, and W. Timothy Strayer. 2001. Hash-based IP traceback. ***ACM SIGCOMM Computer Communication Review*** 31, 4 (2001), 3–14. https://doi.org/10.1145/964723.383060

[81] Kunal Talwar and Udi Wieder. 2013. Balanced Allocations: A Simple Proof for the Heavily Loaded Case. ***CoRR*** abs/1310.5367 (2013). arXiv:1310.5367 http://arxiv.org/abs/1310.5367

[82] Xilin Tang, Feng Zhang, Shuhao Zhang, Yani Liu, Bingsheng He, and Xiaoyong Du. 2024. Enabling Adaptive Sampling for Intra-Window Join: Simultaneously Optimizing Quantity and Quality. ***Proc. ACM Manag. Data*** 2, 4, Article 198 (Sept. 2024), 31 pages. https://doi.org/10.1145/3677134

[83] The Linux Kernel Developers. 2025. Seqlock Implementation in the Linux Kernel (v6.11). https://elixir.bootlin.com/linux/v6.11/source/include/linux/seqlock.h. Accessed: 2025-10-29.

[84] The Linux Kernel Developers. 2025. Sequence Locks (seqlock). https://www.kernel.org/doc/Documentation/locking/seqlock.txt. Accessed: 2025-10-29.

[85] Andrew Todd, Huan Truong, Justin Deters, John Long, Gavin Conant, and Michela Becchi. 2016. Parallel Gene Upstream Comparison via Multi-Level Hash Tables on GPU. In ***2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)***. IEEE, 1049–1058. https://doi.org/10.1109/ICPADS.2016.0139

[86] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. Plush: a write-optimized persistent log-structured hash-table. ***Proc. VLDB Endow.*** 15, 11 (July 2022), 2895–2907. https://doi.org/10.14778/3551793.3551839

[87] David Wajc. 2017. Negative Association: Definition, Properties, and Applications. Manuscript, Carnegie Mellon University. https://www.davidwajc.com/papers/NA.pdf

[88] Chao Wang, Junliang Hu, Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang. 2023. SEPH: Scalable, Efficient, and Predictable Hashing on Persistent Memory. In ***17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)***. USENIX Association, Boston, MA, 479–495. https://www.usenix.org/conference/osdi23/presentation/wang-chao

[89] Junchang Wang, Dunwei Liu, Xiong Fu, Fu Xiao, and Chen Tian. 2022. DHASH: Dynamic Hash Tables With Non-Blocking Regular Operations. ***IEEE Transactions on Parallel and Distributed Systems*** 33, 12 (2022), 3274–3290. https://doi.org/10.1109/TPDS.2022.3151499

[90] Ziqi Wang. 2016. index-microbench. https://github.com/wangziqi2016/index-microbench Accessed: 2025-10-21.

[91] Derrick E. Wood, Jennifer Lu, and Ben Langmead. 2019. Improved metagenomic analysis with Kraken 2. ***Genome Biology*** 20, 1 (2019), 257. https://doi.org/10.1186/s13059-019-1891-0

[92] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. 2016. Bluecache: a scalable distributed flash-based key-value store. ***Proc. VLDB Endow.*** 10, 4 (Nov. 2016), 301–312. https://doi.org/10.14778/3025111.3025113

[93] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-KV: a case for GPUs to maximize the throughput of in-memory key-value stores. ***Proc. VLDB Endow.*** 8, 11 (July 2015), 1226–1237. https://doi.org/10.14778/2809974.2809984