Zombie Hashing: Reanimating Tombstones in a Graveyard

YUVARAJ CHESETTI*, Northeastern University, USA
BENWEI SHI*, University of Utah, USA
JEFF M. PHILLIPS, University of Utah, USA
PRASHANT PANDEY, Northeastern University, USA

Linear probing-based hash tables offer high data locality and are considered among the fastest in real-world applications. However, they come with an inherent tradeoff between space efficiency and speed, i.e. when the hash table approaches full capacity, its performance tends to decline considerably due to an effect known as *primary clustering*. As a result they are only used at low load factors.

Tombstones (markers for deleted elements) can help mitigate the effect of primary clustering in linear probing hash tables. However, tombstones require periodic redistribution, which, in turn, requires a complete halt of regular operations. This makes linear probing not suitable in practical applications where periodic halts are unacceptable.

In this paper, we present a solution to forestall primary clustering in linear probing hash tables, ensuring high data locality and consistent performance even at high load factors. Our approach redistributes tombstones within small windows, deamortizing the cost of mitigating primary clustering and eliminating the need for periodic halts. We provide theoretical guarantees that our deamortization method is asymptotically optimal in efficiency and cost. We also design an efficient implementation within dominant linear-probing hash tables and show performance improvements.

We introduce Zombie hashing in two variants: ordered (compact) and unordered (vectorized) linear probing hash tables. Both variants achieve consistent, high throughput and lowest variance in operation latency compared to other state-of-the-art hash tables across numerous churn cycles, while maintaining 95% space efficiency without downtime. Our results show that Zombie hashing overcomes the limitations of linear probing while preserving high data locality.

CCS Concepts: • **Theory of computation → Data structures design and analysis**.

Additional Key Words and Phrases: Dictionary data structure; Hash tables

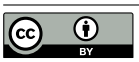
ACM Reference Format:
Yuvaraj Chesetti, Benwei Shi, Jeff M. Phillips, and Prashant Pandey. 2025. Zombie Hashing: Reanimating Tombstones in a Graveyard. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 236 (June 2025), 27 pages. https://doi.org/10.1145/3725424

1 Introduction

Hash tables are one of the fundamental data structures used for managing data in many applications across computer science, including databases [10, 13, 21, 25], storage [6, 12, 18, 27, 45, 58], machine learning [53], high-performance computing [1, 30], computational biology [4, 29], security [22, 52],

*Both authors contributed equally to this research.

Authors' Contact Information: Yuvaraj Chesetti, Northeastern University, Boston, Massachusetts, USA, chesetti.y@northeastern.edu; Benwei Shi, University of Utah, Salt Lake City, Utah, USA, b.shi@utah.edu; Jeff M. Phillips, University of Utah, Salt Lake City, Utah, USA, jeffp@cs.utah.edu; Prashant Pandey, Northeastern University, Boston, Massachusetts, USA, p.pandey@northeastern.edu.



ACM 2836-6573/2025/6-ART236
https://doi.org/10.1145/3725424

and compilers [54]. As a result, high performance hash tables are critical to achieve efficiency and scalability in modern data-intensive applications.

The linear probing hash table [3, 14, 40] is among the most fundamental data structures in computer science. It is one of the fastest hash tables and a go to data structure in every system builder's toolbox. A linear probing hash table is implemented as an array of size n, where each slot either holds an element or remains empty (i.e. a free slot). To insert an element a, the hash function computes $h(a) \in [n]$, and a is placed in the first free slot starting from $h(a)$ and continuing sequentially through $h(a)+1$,$h(a)+2$, and so on (wrapping around using modulo n if needed). To query a, the search scans these slots sequentially, stopping when a is found or when a free slot is encountered.

The key property that makes linear probing appealing is its ***data locality***: each operation only needs to access one or a small number of contiguous cache lines. Even with multiple cache misses, hardware prefetching reduces their cost due to the locality of accesses. Most in-memory production indexes [14, 57] treat the effective cache-line size as 256 bytes even though the hardware cache lines are 64 bytes. Data locality is further critical for building large-scale hash tables for slower storage devices and distributed settings. In these cases, we can perform hash table operations using a single block transfer if contiguous cache lines are accessed. As a result, numerous real-world data management applications employ linear probing for quickly processing and indexing large-scale data [7, 14, 15, 23, 28, 34, 35, 37, 39, 41, 43, 44, 47, 49, 51].

Hash table performance. Hash table performance is defined in terms of *space efficiency* and *speed*. Space efficiency is the ratio of the size of the dataset over the size of the hash table. Speed is further defined in terms of *throughput* and *latency*. High space efficiency is necessary to operate in memory-constrained settings. High throughput enables processing large-scale data quickly. Finally, low and bounded worst-case latency guarantees *consistent* throughput.

In this paper, we focus on designing hash tables that can offer consistent and high throughput while achieving high space efficiency. This has been a longstanding open problem in hash table design.

Target applications. Both the space efficiency and speed of hash tables are crucial for modern data management applications. Various real-world data management applications, such as in-memory caches [32, 46], stream monitoring systems [15, 43], sparse-tensor contractions [26], and genomic assemblers [31], rely on linear probing hash tables to process large-scale data in *memory-constrained* settings. For instance, in-memory caches [32, 46] always maintain hash tables at high space efficiency and must provide consistently low and bounded worst-case latency (i.e. that is no halts) to guarantee service-level agreements. In network stream monitoring systems [15, 43], minimizing worst-case latency is essential to prevent packet loss during data ingestion. In metagenomic assembly [31], even a 5-10% difference in hash table space efficiency can significantly affect the overall accuracy of the assembler.

Challenges in linear probing. Linear probing hash tables suffer from the inherent tradeoff between space efficiency and speed. The performance of operations drops significantly as the hash table becomes full. The reason for these slow insertions is that elements in the hash table have a tendency to *cluster* together into a long contiguous sequence of occupied slots; this is known as ***primary clustering***. Primary clustering is often described as a "rich gets richer" phenomenon, in which the longer a run gets, the more likely it is to accrue additional elements [20]. Quadratic probing [16], which stores elements at a sequence of quadratically growing and non-contiguous slots is commonly suggested as a solution of the primary clustering problem. But the lack of data locality at high load factors again leads to severely degraded performance.

To overcome the effect of primary clustering, probing-based hash tables often employ ***tombstones*** [3, 14]. Tombstones are markers for deleted elements. Tombstones help avoid the immediate compaction that is necessary to rearrange the elements after the deletion. Tombstones do not reduce

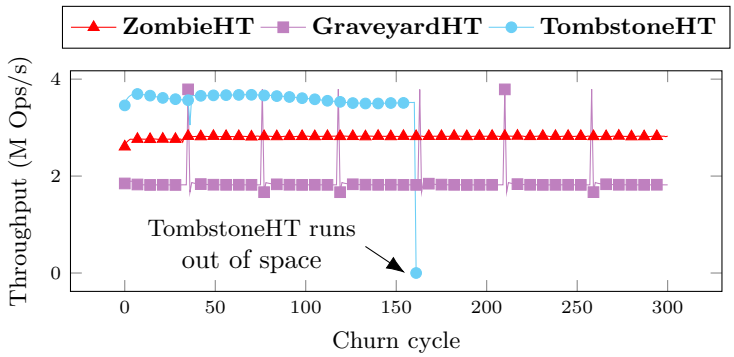


Fig. 1. Throughput of linear probing hash tables with tombstones for multiple churn cycles (updates and queries in equal portions) maintained at 95% load factor. GraveyardHT has lower throughput due to frequent rebuilds (approximately once per churn cycle) and throughput spikes during churn cycles with no rebuilds. The linear probing hash table with tombstones (TombstoneHT) [40] dies after 150 churn cycles due to accumulated tombstones. GraveyardHT [3] periodically stops and redistributes tombstones resulting in halts. Deamortization design in zombie hashing (ZombieHT: this paper) enables consistent performance without periodic halts.

the size of primary clusters, but make insertions and deletions faster. Tombstones interact asymmetrically with queries and insertions: queries skip tombstones during the probe, whereas insertions treat the tombstone as a free slot.

Bender et al. [3] showed that merely using tombstones after deletions leads to their accumulation and performance slowdowns. To avoid this, the table can be regularly rebuilt to clear out tombstones. Graveyard hashing, introduced by Bender et al., periodically redistributes tombstones to ensure low amortized costs for all operations. It achieves better performance than Knuth's analysis [20], with expected operation costs of $O(x)$ when the load factor is $1-1/x$, improving on Knuth's $\Theta(x^2)$ bound for inserts without tombstones.

Graveyard hashing is not practical yet. The paradigm described by Bender et al. [3] (FOCS 21) is exciting and positions linear probing as a viable candidate for high performance hash tables. However, graveyard hashing is not suitable for deployment in practical settings for a variety of reasons. Most notably, it is an ***amortized analysis***, and the algorithm must ***regularly stop*** hash table operations to clean and rearrange all of the tombstones. While the cost of this operation does not dominate the runtime over a long run, it results in complete downtime and dramatically hurts the latency during these clean-up periods, which can be unacceptable for high-performance, time-critical systems [11, 31, 32, 43, 46].

Figure 1 shows the throughput of the state-of-the-art ordered linear probing hash tables for mixed workload containing 50% queries and 50% updates (inserts/deletes). The linear probing hash table employs tombstones (TombstoneHT) deletions. However, it does not stop to reclaim tombstones. As a result, TombstoneHT fails after 150 churn cycles due to tombstone accumulation. This occurs because tombstones introduced by deletions can only be consumed by inserts if the insert falls within the immediate cluster to which the item hashes. Over time, tombstones accumulate, ultimately causing the hash table to run out of empty slots [3]. GraveyardHT on the other hand stops and redistributes tombstones, resulting in periodic halts of operations and high throughput fluctuations.

Why consistent performance is hard? Designing a hash table with consistent performance guarantees is challenging. Local redistribution within a cluster (a sequence of occupied slots) seems promising compared to global redistribution (as in graveyard hashing). However, the cost of local

redistribution grows with the cluster length, which can be as large as $O(x^2)$, causing performance slowdowns. Maintaining two versions of the hash table for redistribution is another option but increases peak space usage significantly. To maintain consistent, high performance, we need a hashing scheme that guarantees $O(x)$ cost per operation, including local redistribution. Our key insight is to rebuild only a small window of the cluster and push excess tombstones to the end of this window. We describe in detail the drawbacks of simple deamortization approaches and elaborate on our approach in Section 3.1, Section 3.2 and Section 4.

This paper. We introduce ***Zombie hashing***, a novel scheme that ***deamortizes*** tombstone redistribution across hash table operations, enabling fast operations without regular halts. Instead of periodic halts, Zombie hashing redistributes tombstones locally in bounded windows during insertions and deletions, maintaining optimal performance. We introduce the notion of *push tombstones* that enable us to perform redistribution inside a bounded window while guaranteeing theoretical bounds. We provide a theoretical analysis showing that deamortizing tombstone redistribution achieves the same optimal asymptotic cost as graveyard hashing (Section 4). We evaluate several deamortization strategies, selecting the most efficient one to minimize cache misses (Section 3).

Moreover, we empirically evaluate Zombie hashing and implement it in two linear probing variants. First, we provide a compact [7] and ordered linear probing hash table (ZombieHT(C)) that uses quotienting and Robin hood hashing (Section 6). Compact hashing enables ZombieHT(C) to achieve very high space efficiency. It guarantees $\Theta(x)$ time complexity for all operations, theoretically proven in Section 4. Unlike graveyard hashing, our $\Theta(x)$ bound does not rely on amortized analysis. Second, we develop a vectorized linear probing hash table (ZombieHT(V)) using the industry-grade AbslHT [14]. The vectorized design enables ZombieHT(V) to achieve similar average throughput and space efficiency as the AbslHT but no periodic downtime and three orders-of-magnitude lower worst-case latency. ZombieHT(V) provides a $\Theta(x^2)$ bound for all operations, similar to other unordered linear probing hash tables, but without requiring periodic pauses to clear tombstones.

The compact and vectorized variants of the hash table offer distinct tradeoffs. ZombieHT(C), the compact variant, is highly space-efficient and employs ordered linear probing with $\Theta(x)$ time complexity for all operations, theoretically proven in Section 4. Unlike graveyard hashing, the $\Theta(x)$ bound does not rely on amortized analysis. On the other hand, ZombieHT(V), the vectorized variant, uses unordered linear probing and delivers higher throughput by utilizing vector (SIMD) instructions. The performance gain, however, comes at the cost of reduced space efficiency compared to the compact variant. ZombieHT(V) provides a $\Theta(x^2)$ bound for all operations, similar to other unordered linear probing hash tables, but without requiring periodic pauses to clear tombstones.

We perform extensive empirical evaluation to show that the ZombieHT(C) and ZombieHT(V) achieve high and consistent throughput across hundreds of churn cycles (Section 8). Each churn cycle evaluates the performance of the hash table when it is almost full. This simulates hash table aging that is a common workload in production data management settings [11, 32, 46]. In cache systems with a fixed memory budget, each insert (cache put) is preceded by a delete (cache eviction) once the cache is full, maintaining a roughly fixed (and typically high) load factor. Insertions and deletions are balanced in these settings. When insertions and deletions are unbalanced, ZombieHT also employs primitive tombstones (similar to the graveyard hashing) to balance the number of tombstones and empty slots. Its theoretical guarantees rely on these primitive tombstones and not deletions to break primary clusters, eliminating the need for stable, frequent deletions. While resizing can help mitigate this issue by reducing x, it is often not feasible in memory-constrained applications like caching systems [32, 46] or streaming monitoring [15, 43]. We conclude by empirically analyzing the tuning parameters and optimizations in Zombie hashing, and their impact on the overall performance.

Our results.

(1) We devise a novel hashing scheme, Zombie hashing, which efficiently maintains tombstones in linear probing hash tables; it eliminates the issue of primary clustering at high load factors *and* avoids redistribution downtime.
(2) We theoretically *prove* that Zombie hashing achieves worst case $O(x)$ time for all operations even including the cost of tombstone redistribution.
(3) We empirically show that ZombieHT(C) (an ordered variant) offers the best space efficiency, and among compact, ordered hash tables, provides the highest average throughput, as well as the best consistency (a 4 orders-of-magnitude improvement) measured as worst case latency and latency variance.
(4) Similarly, ZombieHT(V) (an unordered variant) matches state-of-the-art space efficiency among production-level vectorized hash tables. Among these, in common high query/update settings, it achieves the best consistency (3 orders-of-magnitude lower worst latency).

2 Background

Notations. For a linear probing hash table of size n containing t elements (we must have $t \leq n$) from a size u universe, its ***load factor*** is t/n. We typically write this as $t/n = 1 - 1/x$, and the key quantity in the analysis is x. The variable x can be thought of as the average distance between empty slots in the hash table. At a high load factor such as 95% (when x is not a constant), we expect all operations to be $O(x)$. While this load factor cannot be exactly maintained, it is common to keep it approximately fixed (e.g., a load factor of 95% may mean $x \in [18,25]$). As long as it stays within some such bounded range, a so-called ***hovering workload***, it is meaningful to discuss analysis in terms of asymptotic values of x (e.g., $O(x)$ or $\Theta(x^2)$).

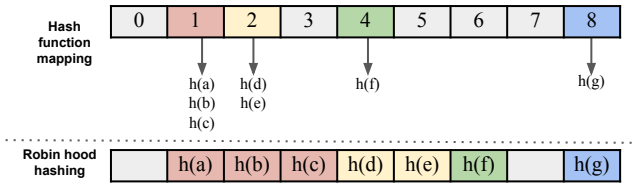
Hash functions. We use a fixed hash function $h : [u] \to [n]$ for the algorithm. For ease of analysis we assume h is drawn uniformly randomly from a fully independent family of hash functions, and all of the randomness in the analysis comes from this random choice. While these conditions on the hash family are somewhat unrealistic in theory, they can be addressed in theory [3] and the practical construction of these is a mostly an orthogonal issue. For example, Murmurhash [55] is a popular hash function that offers enough randomness for hash table theory to hold in practice.

Runs and clusters. For linear probing hash tables, a *run* means an interval of slots where the elements have same hash value. If $h(a)$ is the hash value of the run, we use $run\_start(h(a))$ and $run\_end(h(a))$ for its start and end indices respectively. We define the displacement of the run as $run\_start(h(a)) - h(a)$. We call a run at its home position if its displacement is 0, $h(a) = run\_start(h(a))$. A *cluster* is a contiguous sequence of runs, it starts with a run at its home position, and ends when it reaches an empty slot *or* any other run at its home position. That is, in a cluster, other than the first element, every other element is displaced from its home.

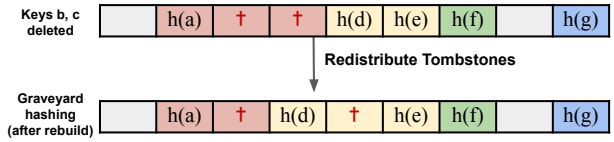
2.1 Linear probing hash table

The linear probing hash table is probably the simplest, most well known, and most well studied open addressing hash table. To insert a key a, it computes the hash value $h(a)$, and put it into the first empty slot, starting from index $h(a)$. To lookup a key a, it scans through the slots starting with $h(a)$ until it finds a or a free slot (indicating a is not in the table). To delete a key a, it first does a lookup, then removes it and shifts the following slots to preceding slots if their hash value is smaller than the position of that preceding slot.

The well known drawback of the linear probing is *primary clustering*. The longer a cluster is, the more likely a key is to be hashed into that cluster, and for that cluster to grow larger. It is an example of "rich gets richer" phenomenon. Although the expected length of a given cluster is $\Theta(x)$ over all



(a) An example showing Robin hood hashing scheme. Items in the hash table are stored in sorted order of their hash value. All items hashing to the same slot are stored contiguously in runs.



(b) An example showing graveyard hashing scheme. The figure demonstrates the state of the hash table before and after a graveyard redistribution operation. Items b, c are deleted and marked with tombstones. After redistribution, the empty slots and tombstones are evenly redistributed.

Fig. 2. Robin hood hashing and graveyard hashing schemes.

clusters, the standard deviation is $\Theta(x^4)$. Since the longer clusters have more keys, for a random key, the average length of its cluster is $\Theta(x^2)$ [20]. So the time complexity of all operations are $\Theta(x^2)$ for a naive linear probing hash table.

2.2 Robin hood hashing

A simple optimization, called *Robin hood* hashing, can be made to the naive linear probing hash table to improve the expected lookup time from $\Theta(x^2)$ to $\Theta(x)$: instead of putting new key at the end of a run, we maintain them in ***sorted order*** [5, 19, 48]. Figure 2a shows the design of the Robin hood hashing scheme.

Robin hood with tombstones. Tombstones are widely used in hash tables to improve the performance of deletions [14]. By using tombstones, one does not have to shift an entire cluster on a deletion. Instead, one simply marks the deleted element with a tombstone [17]. Thus the deletion time now also achieves the same optimal rate as the lookup: $\Theta(x)$. Because these scattered tombstones break up the primary clusters, they can also be utilized by future insertions whose hash value is smaller.

Of course, using tombstones for insertions is not free. Each deletion (except the ones at the end of a cluster) results in a tombstone, but not every insertion consumes a tombstone. Thus the number of tombstones will keep increasing and eventually fill the whole hash table. If an element is inserted far from its home slot, it has no recourse to move closer. Addditionally, lookups need to traverse (or skip) these tombstones to find an element or determine that it is not in the table.

Thus, the practice says that hash tables using tombstones need to rebuild (clean tombstones) after some number R of insertions and deletions. The bigger the R (rebuild window), the slower the lookup time, since accumulated tombstones effectively increase the load factor. The smaller the R, the slower the insertions, as it is less likely to find a tombstone to help speed up the insertion after rebuilds, and we default to the $\Theta(x^2)$ analysis.

Hash table	Insert	Delete	Lookup	Rebuild Age
Linear Probing [20]	$\Theta(x^2)$	$\Theta(x^2)$	$\Theta(x^2)$	∞
RobinHoodHT [5]	$\Theta(x^2)$	$\Theta(x^2)$	$\Theta(x)$	∞
RobinHoodHT +TS [17]	$\tilde{\Theta}(x^{1.5})$	$\Theta(x)$	$\Theta(x)$	$n/\text{polylog}(x)$
GraveyardHT [3]	$\Theta(x)$	$\Theta(x)$	$\Theta(x)$	$n/(4x)$
ZombieHT (This paper)	$\Theta(x)$	$\Theta(x)$	$\Theta(x)$	∞

Table 1. Linear probing variants

2.3 Graveyard hashing

A recent breakthrough by Bender et al. [3] introduced *graveyard hashing* that showed how to effectively maintain tombstones in a linear probing hash table. Their analysis showed that all operations would be efficient in an amortized sense. The key idea is to retain some tombstones on a rebuild and ensure they are evenly distributed. Basically, one wants both empty slots and tombstones; they should roughly be the same ratio, and both evenly spaced. Figure 2b shows the design of the redistribution scheme in graveyard hashing.

For a size n a hash table with a load factor of $1-1/x$, graveyard hashing rebuilds need to be performed after $n/(4x)$ operations (insertion and deletion). During rebuilding, $n/(2x)$ tombstones (primitive) are placed evenly in terms of their hash values. This gives us $\Theta(x)$ time complexity for all operations: insertion, deletion, and lookup. Like many other hash tables that use tombstones, it also must periodically stop and rebuild, the rebuild time is $O(n)$, and so it amortizes to $O(x)$ per insertion or deletion.

Theorem 1 (Graveyard Hashing [3])**.** *For a graveyard hash table with a load factor of $1-1/x$, each insertion/lookup/deletion operation takes expected time $O(x)$, this includes the amortized cost for rebuilds.*

Note that while the expected value of these operations is optimal, the ***variance is extremely high***. If an operation occurs near the start of a rebuild pause, then it takes $O(n)$ time; since it must first wait for the rebuild to finish. **The motivation of this paper is the question, can we deamortize the rebuild operation in graveyard hashing so that the maximum latency of any operation is bounded?**

Table 1 summarizes the theoretical results at high load factors of the linear probing variants we discussed above. Linear Probing and RobinHoodHT do not need rebuilds (rebuild window size $R = \infty$) but they suffer from poor performance at high load factors. RobinHoodHT + Tombstone (TS) and GraveyardHT need rebuilds to maintain their performance at high load factors. The last row, ZombieHT, is our main result and the best possible; it not only matches the GraveyardHT performance $\Theta(x)$ in all operations, but also achieves consistency. Here consistency means that we do not stop regular hash table operations during rebuilds independent of the load factor of the hash table.

3 Deamortizing Graveyard Hashing

In this section, we first discuss potential techniques to efficiently deamortize graveyard hashing. We then present and analyze a simple deamortization technique that we further refine in Section 4.

3.1 Potential deamortization solutions

We preview some potential ways to deamortize graveyard hashing, ones which do not achieve high efficiency at high load factors. However, analyzing these will ultimately lead us to our proposed solution.

The mostly common deamortization technique is to create two versions of the hash table, one that is active, and one that is rebuilding. When the rebuild finishes, it switches to active, and the other

starts its rebuild. However, this requires at least double the amount of space required (maybe even requiring a factor 5 increase since a graveyard triggers a rebuild after $n/(4x)$ operations), and defeats the purpose of studying the high load factor setting.

Another option is to divide the hash table address space into a set of T regions. In each region we maintain a graveyard hash table, which each periodically rebuilds. If we ensure each is small (perhaps on the order of $O(x)$), then the rebuild time is no longer problematic. However, while the load factor across the entire table may be $(1-1/x)$, because keys are hashed to regions randomly, it will tend to not be even across regions. Some regions will have a significantly higher load factor, including greater than 1. This is because the hash function is uniformly random, so the guarantee for uniformly distributing keys is only in expectation. Using the balls and bins analysis [33], the partition with maximum load will have $O(\log n/\log\log n)$ keys with high probability. While this could potentially be addressed with a backyard (an extra back-up space), this effectively reduces the overall load factor and leads to very poor locality.

Finally, we consider an attempt to rebuild the hash table *in-place*. Here, for each operation (insert/delete), we do part of the work towards rebuilding the hash table. Two challenges must be addressed for this to work. First, the graveyard hash analysis requires *everything* to be reset after a fixed "age" of $n/(4x)$, so one needs to ensure the necessary invariants hold. Second, ideally one would only spend $O(x)$ time in rebuilding per operation. However, due to primary clustering, the average cost of adjusting an entire run may be $\Theta(x^2)$.

We analyze this in-place rebuild setting in detail in the next section. We show how to handle the first issue (maintaining graveyard invariants). This approach suffers from the second issue but provides the foundation for our method, Zombie hashing, which addresses the second issue.

3.2 Simple deamortized GraveyardHT

Here we analyze a simple method to deamortize graveyard hashing (DGH). This method does not have good locality and induces other overheads that we address in the next section.

This simple approach divides the hash table into R intervals and rebuilds a single interval after each insertion or deletion. Rebuilding an interval means removing existing tombstones in the interval, and putting new ones at an evenly spaced positions called primitive tombstone positions. To clear the other tombstones, we push tombstones until we either reach a primitive tombstone position that does not have a tombstone or the end of the cluster. During the pushing process we shift keys back appropriately to ensure they are not before their home position. This shifting maintains the necessary Robin hood hashing invariants. We can push a set of multiple excess tombstones within one run.

Now we show that with the right settings, DGH has the same performance as graveyard hashing on insertion, query, and deletion, while rebuilding incrementally.

Note that for linear probing hash table with tombstones, the lookup performance is determined by the "true" load factor $1-1/x'$, in which the tombstones are also counted as part of the load. So we first show that x' is $O(x)$ if we rebuild it often enough and do not insert too many primitive tombstones.

Lemma 2. *Consider a DGH hash table running a hovering workload with load factor* $1-1/x$. *Let* $c_p x$ *be the distance between primitive tombstones and* $c_b x$ *be the size of each rebuild intervals. If* $c_b \geq c_p \geq 3$, *then the true load factor* $1-1/x' \leq 1-1/(3x)$.

Proof. There are 2 types of tombstones in DGH: primitive tombstones and those left by deletions. Let T_P and T_D denote the number of primitive tombstones and deletion tombstones, respectively. At any time, $T_P \leq n/(c_p x)$ by its definition. Moreover $T_D \leq R = n/(c_b x)$ since each non-primitive tombstone has age at most R.

$$T_P + T_D \leq \frac{n}{c_p x} + \frac{n}{c_b x} = \frac{n}{x}\frac{2}{H(c_b, c_p)}$$

Where $H:\mathbb{R}^2\to\mathbb{R}$ is the harmonic mean. Let E be the number of empty slots, such that when the load factor is $(1-1/x)$,

$$\frac{n}{x}=E+T_P+T_D=\frac{n}{x'}+T_P+T_D.$$

Solving for

$$\begin{aligned}
x' &= n/\left(\frac{n}{x}-(T_P+T_D)\right)\\
&\le n/\left(\frac{n}{x}-\frac{n}{x}\frac{2}{H(c_b,c_p)}\right)\\
&= x\frac{H(c_b,c_p)}{H(c_b,c_p)-2}\\
&\le x\frac{\min(c_b,c_p)}{\min(c_b,c_p)-2} && H(a,b)\ge\min(a,b)\\
&= x\frac{c_p}{c_p-2} && \text{if } c_b\ge c_p\\
x' &\le 3x && \text{When } c_p=3
\end{aligned}$$

□

Then following the graveyard hashing analysis [3], DGH can do insertion, query, and deletion in time $O(x)$ in expectation:

Lemma 3. *Consider a DGH hash table running a hovering workload with load factor $1-1/x$. It rebuilds a size c_bx interval after each insertion, and the distance between primitive tombstones is c_px. If $c_b=2c_p=6$, then each insertion/query/deletion takes time $O(x)$ in expectation.*

This follows directly from the analysis of graveyard hashing [3]. We outline the main points here. Since, via Lemma 2 the true load of $1-1/O(x)$, then we have $O(x)$ time complexity for lookup and delete. The analysis of graveyard insertion requires two conditions: (C1) the age of primitive tombstones after the insertion index (by its quotient) is half of the total number of primitive tombstones; and (C2) the distance between primitive tombstones is $O(x)$. The first condition can be satisfied by setting $c_b=2c_p$ in DGH, and the second condition is satisfied by setting $c_p=3$. Through a more complicated analysis, deferred to the full version for space, we can use smaller c_b for better performance. We verify that this setting works experimentally in Section 8.

Although the size of each rebuild interval c_bx is $O(x)$, the DGH cannot always finish each rebuild in time $O(x)$. The rebuilding process may not be able to stop at the end of the interval. An insertion of a primitive tombstone has to shift everything after it forwards (to the next interval) until a free slot or another tombstone. Deletion of tombstones has to shift the rest of the cluster backwards if there are more tombstones to be cleaned than primitive tombstones to be inserted. As discussed in Section 2.1, the cluster length is $O(x^2)$ in expectation. Factoring in the rebuild cost of insertion, DGH may require $\Theta(x^2)$ time for insertions.

4 Zombie Hashing

To improve the rebuilding performance of DGH which we introduced in Section 3, we now describe our first working algorithm, zombie hashing (ZH). The hash table implementation is called ZombieHT in later sections.

Description of zombie hashing. Just like the DGH, the zombie hash breaks up the rebuilding process and executes it gradually throughout the rebuild time window. The whole hash table is divided into $n/(c_px)$ rebuild intervals, each with c_px hash values for some constant c_p like 3. These

intervals will be rebuilt from the first one to the last one and start over. After each insert, one of these intervals is rebuilt. The rebuilding of an interval inserts one tombstone at each primitive position (that does not already have one), and pushes all the other tombstones out of the interval. This is the only difference between zombie hashing (ZH) and DGH; specifically, zombie hashing does not push extra tombstones all the way to the end of the cluster, it just leaves them at the beginning of the next rebuild interval. This ensures each rebuild is $O(x)$ time.

The key idea that enables rebuilding intervals is the idea of *pushing tombstones*, which are tombstones that have been pushed out of the rebuild interval, but not yet to the end of the cluster. We let T_F denote their count. We next argue that by leaving these pushing tombstones as pending at the end of an interval rebuild, we do not break any invariants of the Robin hood hash table. These pushing tombstones will eventually either reach an empty slot, a primitive position or the end of the cluster and become free slots. This keeps the tombstones evenly distributed, while pushing extra tombstones forward. Note that this does not require any assumptions about inserts and deletes being evenly distributed in zombie hashing. The deamortization schedule will balance the tombstone distribution by using *primitive* and *pushing* tombstones.

The following observation illustrates the key difference between DGH and ZH.

Fact 4. *Consider two hash tables DGH and ZH on the same input set, using the same hash function, and all of the same parameters. One can convert ZH to DGH by cleaning the T_F pushing tombstones in ZH; or get the status of ZH from DGH by inserting T_F tombstones to DGH at the beginning of the next rebuild interval. Here inserting a tombstone at i means inserting a tombstone with hash value i; if there is already tombstones with hash value i, keep all of them.*

Thus to bound the complexity of ZH's operations, it is important to bound T_F.

Lemma 5. *Consider a ZH with n slots and load factor of $1-1/x$, rebuild size $c_b x$ intervals and use $c_p x$ as the distance between primitive tombstones. If $c_b \geq c_p \geq 3$, then the number of tombstones being pushed forwards T_F is $O(x)$ in expectation.*

Proof. Let C be a cluster in DGH, and $T_D(C)$ be the number of tombstones left by deletions in C. Obviously, the number of pushing tombstones in a cluster C satisfies

$$T_F(C) \leq T_D(C).$$

Let $L(C)$ be the length of C, we have the expected number of non-primitive tombstones in a length ℓ cluster is

$$\mathbb{E}[T_D(C) \mid L(C) = \ell] \leq R\ell \frac{1}{n} = \frac{n}{c_b x} \frac{\ell}{n} = \frac{\ell}{c_b x}.$$

This is because there are at most R deletions that can happen between a rebuild cycle, each hash value in the cluster has $1/n$ probability to be a tombstone left by a deletion, and there are ℓ different hash values in the cluster.

Then we bound the cluster length $L(C)$. The key observation here is that, when DGH encounters a free slot, ZH does so also and has 0 pushing tombstones. That is, the clusters after rebuilding are exactly the same in DGH and ZH. We have shown in Lemma 2 that the load factor of DGH is $1-1/x'$ where $x' = O(x)$ if $c_b \geq c_p \geq 3$. Thus the expected length of a cluster is $O(x'^2) = O(x^2)$.

Put all the above together, we have the expected number of pushing tombstones is

$$\begin{aligned}\mathbb{E}[T_F] &= \mathbb{E}_L[\mathbb{E}[T_F(C) \mid L(C) = \ell]] \leq \mathbb{E}_L[\mathbb{E}[T_D(C) \mid L(C) = \ell]] \\ &= \frac{\mathbb{E}[L]}{c_b x} = \frac{O(x^2)}{c_b x} = O(x).\end{aligned}$$

□

Finally, we can use Fact 4 to show most operations on ZH perform exactly the same in DGH, and the others are off by $T_F = O(x)$; hence all operations have runtime $O(x)$ in ZH.

Theorem 6 (Zombie Hashing)**.** *Consider a Zombie hash table with load factor at most* $1-1/x$*. Let* $c_p x$ *be the distance between its primitive tombstones, and let* $c_b x$ *be the size of the rebuild interval after each insertion. If* $c_b \geq 2c_p \geq 6$*, then each operation (insertion, query, deletion, or rebulding) takes expected time* $O(x)$*.*

Proof. Fact 4 tells us that the state of the hash tables for DGH and ZH are almost the same. And Lemma 3 shows that, if $c_b \geq 2c_p \geq 6$, for the corresponding DGH, the insert, lookup, or delete takes time $O(x)$ in expectation. We define the *active cluster* as the set of slots which are in the same cluster as the pushing tombstones under either ZH or DGH. If an operation indexes outside of the active cluster, then DGH and ZH are the same, and so in ZH the operations also take $O(x)$ time. For an operation that indexes in the active cluster, but after the pushing tombstones, then in ZH these have not been processed in a rebuild yet. However, these intervals must have been rebuilt in the last $R = n/(c_b x)$ operations. Hence, we can invoke the graveyard analysis, as in DGH, to bound the expected runtime of any operation to $O(x)$.

What remains are operations, with home slot (indexed by its quotient) in the active cluster, but before the pushing tombstones. An insert will behave the same in DGH or ZH before the pushing tombstones, and if it hits the pushing tombstones in ZH it can insert and stop. So its runtime in ZH is at most that in DGH, which is $O(x)$. For a delete or lookup, the operation may need to traverse the pushing tombstones in ZH, whereas in DGH, the items afterwards would have been shifted backwards. By the Robin hood invariant, other than shifting backwards, the order of the items does not change. Thus in ZH these operations may take T_F steps longer than in DGH, and Lemma 5 shows $T_F = O(x)$ in expectation. Hence these operations in ZH take $O(x)+O(x) = O(x)$, as desired.

For the rebuild time, if there are pushing tombstones, the rebuild time is asymptotically the rebuild interval size $c_b x$ plus the number of pushing tombstones T_F; both of them are $O(x)$. Otherwise, the rebuild may need to insert $\lceil \frac{c_b}{c_p} \rceil = O(1)$ primitive tombstones, the time is the same as insertion time, $O(x)$ again. Therefore the rebuild time is $O(x)$ in expectation for ZH. □

5 Optimizations for Zombie Hashing

In this section, we describe an optimization to ZombieHT called ZombieHTDelete. This variant does not introduce new primitive tombstones. Instead, tombstones are only redistributed when introduced as part of a delete. In this optimization, we do not manage a separate rebuild process. Tombstones are redistributed in the neighbourhood of the location they are inserted after performing the deletion. This optimization helps reduce the total number of cache accesses achieves better data locality and overall efficiency.

For this optimization, we need a mechanism to identify if the new tombstone introduced as part of a delete is useful (on redistributing reduces insert cost) or an extra tombstone. To do this, we push this new tombstone to the first primitive tombstone position ahead of it which does not contain a tombstone. If all the primitive tombstone positions between the deleted item and the next empty slot already have tombstones, this new tombstone is cleared and removed.

ZombieHTDelete has the advantage of providing more primitive tombstones, which are the useful tombstones that speed up inserts. ZombieHTDelete only has primitive tombstones, which are bounded by $n/(c_p x)$. To achieve $1-1/(3x)$ true load factor, ZombieHT needs $c_p \geq 3$ (see Section 4), but ZombieHTDelete only needs $c_p \geq 1.5$.

We will now prove that the insert cost in ZombieHTDelete is equivalent to the one in ZombieHT. Consider an insertion of item v that uses a primitive tombstone t which was inserted during one of the previous rebuilding processes in ZombieHT. We aim to show that the cost of insertion of v in ZombieHTDelete is the same as the combined cost of inserting both v and t in ZombieHT. All *tombstome intervals* (the interval between two consecutive tombstones) can be divided into three types per the number of pushing tombstones at the end of their rebuilding process: no tombstone

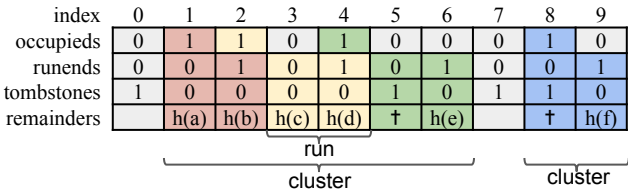


Fig. 3. Ordered linear probing hash table with tombstones. Colors represent different runs. We use the tombstone metadata bit to mark both the tombstone slot and the free slot. Remainders represent the slot in the hash table.

(type 0), exactly 1 tombstone (type 1) and more than one tombstone (type 2). Rebuilding type 0 tombstone intervals using ZombieHTDelete is equivalent to ZombieHT since the cost of clearing or consuming extra tombstones is paid during an insert in ZombieHTDelete. For type 1 tombstone intervals, the rebuilding process in ZombieHTDelete is just a slight movement of a tombstone, so it does not affect performance. For type 2 tombstone intervals, if there are 2 tombstones, the cost to pushing the extra one is the same in ZombieHT and ZombieHTDelete. The rebuilding process in ZombieHT is only more efficient in the case of having more than 2 tombstones as multiple tombstones are pushed together, instead of one by one as in ZombieHTDelete.

6 Ordered variant

The implementations of ZombieHT and other ordered linear probing variants in our evaluation (RobinHoodHT and GraveyardHT) are based on the quotienting [20], a compact hash table design.

6.1 Compact hash table

Quotienting [20] is a technique to compactly store fingerprints in a hash table. It has been used extensively to compactly store small fingerprints in an approximate membership query data structure, such as the quotient filter [40]. The quotient filter stores fingerprints compactly in a hash table using the Robin hood hashing-based linear probing technique. In the context of the quotient filter, the fingerprints are lossy and hence the data structure is approximate. However, if the fingerprints are derived using an invertible hash function [37] then the quotient filter turns into a compact hash table. In this paper, the quotient filter refers to the hash table variant if not specified otherwise.

In quotienting, a hash value $h(a)$, where ($|h(a)| = p$ bits) is divided into two parts, the higher order q bits $h_0(a)$ is called the *quotient*, and the lower order r bits $h_1(a)$ is called the *remainder*, where $q+r=p$ is the total number of bits of the hash value. The quotients are used as the indices in the hash table, and only the remainders are stored in the hash table. So an invertible hash function is needed to recover the key from the hash value and make sure that there is no hash collision. ZombieHT supports 64-bit keys, which is most common data type in hash tables. To support arbitrary-size string keys, we can use indirection as used in existing hash tables [2].

The quotient filter (QF) [40] uses two, size n binary arrays to store the metadata, *occupieds* $[O_i]_{i=1}^{n}$, and *runends* $[R_i]_{i=1}^{n}$. O_i is set if and only if there exists a run whose quotient is i. R_i is set if and only if there exists a run end at i. One can quickly find a run performing rank/select operations on occupieds and runends bit vectors. For example, in Figure 3, if we want to find the run of quotient 4, we first do rank(4) on occupieds to get the number of set bits until 4, which is 3; then do select(3) on runends to get the end of the run, which is 6. To find the start of the run, we first find the end of the previous run. The start of the run is then the maximum of the previous run end plus 1 and the quotient.

Let Z be the information-theoretical lower bound of space to store a certain amount of data in the worst case (max entropy). The data structure used to store the data is *compact* if it uses $O(Z)$

space. For a size n set of elements from universe $[u]$, $Z_{set}(u,n) = \log\binom{u}{n}$, because there are $\binom{u}{n}$ possible different such sets.

The quotient filter (QF) is compact because its space for n keys is $O(nr)$, which is $O(Z_{set}(u,n))$, because

$$nr = n(\log u - \log n) = n\log\frac{u}{n} \leq \log\binom{u}{n}$$

6.2 QF with Tombstones

To introduce tombstones, we extend the QF metadata and add another binary array *tombstones* $[T_i]_{i=1}^{n}$. T_i is set if and only if slot i is available. So we do not just set it for tombstones, but also for empty slots. Thus, we can quickly find an available spot during insertion by a mask and a select operation on the tombstones bit vector.

For example, see Figure 3, to make space to insert g between a and b, we need to find the first available slot after a. With tombstones metadata, we can mask all bits until a and do a select(1) on tombstones to get the first available slot, which is 5. Then we shift everything between a and 5 one slot to the right, and insert g after a.

We further improve the quotient filter (QF) by using block offsets more efficiently. In the original QF design, the remainders and two metadata bits are grouped into blocks of size 64, and each block stores a *block offset* — the number of slots that have overflowed from the previous block. This allows rank/select operations to be limited to the current or nearby blocks, avoiding scans over the entire array. However, at high load factors, this offset can become too large to fit in an 8-bit integer, causing buffer overflows and degrading performance. The original QF handles this by linearly probing subsequent blocks until one with a non-overflowing offset is found.

Instead, we redefine the block offset as the difference between the number of occupied entries and the number of runend markers before the current block — effectively counting the number of *runs* overflowing from the previous block, rather than the number of overflowed slots. This value is significantly smaller and fits within an 8-bit integer in almost all cases, eliminating the overflow issue and improving performance.

6.3 Zombie hashing rebuilds

Now we describe the rebuilds of zombie hashing in detail.

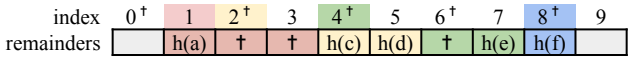
As we mentioned in Section 4, we divide the hash table into size $c_b x$ intervals, and rebuild one interval at a time from the first one to the last one. For the jth rebuild, the rebuild interval is $[s,e)$, where $s = jc_b x$ and $e = jc_b x + c_b x$. The rebuild process first finds the run start using quotient s. If there is no such run, it will be the next existing run. For each quotient $i < e$, do the following,

- if i is a primitive tombstone position (i.e. $i\%(c_p x) = 0$),
  - if there are some pushing tombstones, leave one tombstone at the beginning of the run,
  - otherwise insert a new tombstone at the beginning of the run
- collect all tombstones in the run and push them out of the run. If the next run is empty or has its quotient equal to the index, the pushing tombstones are converted to empty slots.
- find the next existing run.

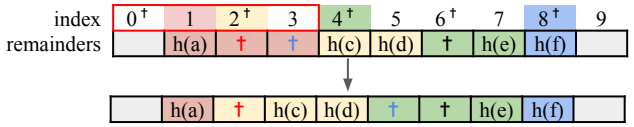
Figure 4 demonstrates a simple example with $c_p x = 2$ and $c_b x = 4$.

7 Unordered variant

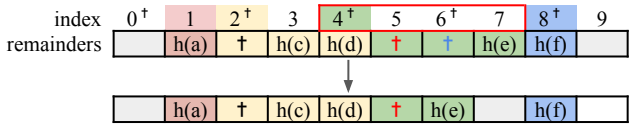
We also implement the deamortized zombie hashing technique in an unordered linear probing hash table called AbslHT [14]. AbslHT is a vectorized quadratic probing hash table from Google [14]. In vectorized design, the hash table is partitioned into blocks and each block is operated on by vector instructions. However, if the blocks are viewed as a consecutive series of slots, the Zombie hashing paradigm can be applied.



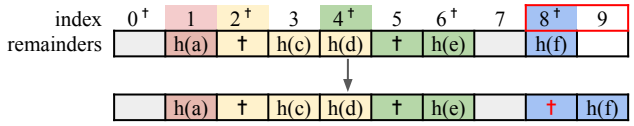
(a) ZombieHT with primitive tombstone slots marked. $c_p x = 2$, every second slot is a primitive tombstone slot.



(b) First rebuild step with rebuild interval [0,3] ($c_b x = 4$). One primitive tombstone (red) is left at the start of run 2, while the extra tombstone (blue) is pushed until it reaches run 4.



(c) Second rebuild step for runs [4,7]. One primitive tombstone (red) is left at start of run 4. Here, the extra push tombstone (blue) is moved to slot 7 and converted to an empty slot.



(d) Third rebuild step for runs [8,9]. As there are no existing tombstones to use, an extra tombstone is introduced for run 8.

Fig. 4. Demonstration of ZombieHT rebuild with $c_b x = 4$ (rebuild window) and $c_p x = 2$ (primitive tombstone distance). We omit metadata bits in these figures and mark slots and corresponding home indexes with the same color. In ZombieHT, the rebuild steps happen in conjunction with inserts and deletes, but here we only show the rebuild steps.

AbslHT uses tombstones to support fast deletes. However, similar to other tombstone-based hash tables, it periodically stops to clear tombstones in order to maintain high throughput. To overcome the periodic down times, we first implement a linear-probing variant in AbslHT while preserving the vectorized execution and original performance. We then implement deamortization using the zombie hashing technique to achieve consistent high throughput with no downtime.

7.1 AbslHT design

The reference AbslHT implementation is an open addressing hash table and uses quadratic probing along with vectorized instructions to quickly probe long chains [50].

Data layout. Items in AbslHT are stored inline continuously in an array. A separate metadata array stores one fingerprint per item in the metadata array. These fingerprints indicate the status of the slot (empty, deleted or filled). The metadata array in conjunction with Streaming SIMD Extension (SSE) [56] instruction set is used to speed up lookup of elements, which is described in more detail below.

Fingerprints. The fingerprints in the metadata array can have 3 states - *empty*, *deleted* (a tombstone marker for deleted items) or *full*. The first bit of the fingerprint is used to differentiate between non-full slots (i.e. empty or deleted) and full slots. For full slots, the last 7 bits of the the fingerprint are filled with the last 7 bits of the hash of the item, which is the fingerprint of the item.

Probe Sequences and groups. The first 57 bits of the hash are used to define a probe sequence for an element. Slots in AbslHT are always probed in *groups*, which are contiguous chunks consisting of k

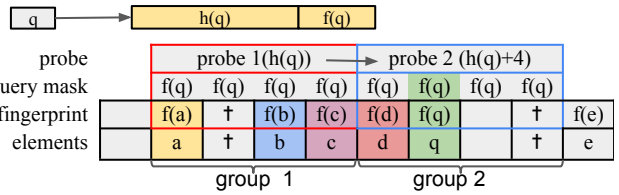


Fig. 5. Unordered linear probing design in AbslHT. Given a query q, it is first hashed to a slot where the probing starts and a 8-byte fingerprint. First, a fingerprint match is performed in the group starting at the hashed slot. If a fingerprint match is found the corresponding slot in the table is compared with the full key. Otherwise, the same process is followed in the next group. The probe stops if the key is found or there is an empty slot in a group.

slots. If a probe sequence for an element is $[p_1,p_2,p_3...]$, then the actual slots checked for in this probe sequence are the slots in the groups $[(p_1,p_1+1,...p_1+(k-1)),(p_2,...p_2+(k-1)),...]$. The reference implementation of AbslHT uses a quadratic probing sequence, where $p_i = (h(a)+k*(i^2+i)/2) \pmod n$. The size k of a group is determined by the width of the vector instruction a machine allows. For a machine that has the 128-byte wide extension, a group width of $k=16$ is used.

Lookup. Using groups along with fingerprints in the metadata array allows AbslHT to quickly probe long probe sequences using vector instructions. The fingerprints in the metadata array are used to quickly find candidate slots in a group that might contain the item. Each fingerprint consists of a control bit (empty or not) and the last 7 hash bits of the item (if the slot is not empty). To find candidate slots, AbslHT constructs a query fingerprint consisting of a set control bit and the last 7 hash bits of the query item. A single vector instruction then finds slots in the group with a fingerprint that matches the query fingerprint. Since the control bit is set, empty slots are ignored when searching for candidate slots. The probe ends if the queried item is found in any of the candidate slots. If the item is not found, the probe ends if the group contains empty slots. Otherwise, the probe continues to the next group in the probe sequence. For a running example, please refer to Figure 5.

Inserts and deletes. Inserts first perform a lookup to check if the item already exists in the table. If the item does not exist, the probe is restarted to find the first empty or deleted slot. Deletes proceed in a similar fashion to lookups. If the item exists, the corresponding fingerprint of the slot is updated with a tombstone to mark the slot as a deleted slot.

Rebuilds. When the actual load factor (the load factor including tombstone slots) goes beyond 87.5%, AbslHT decides between resizing to a new table of twice the capacity or removing all tombstones. The tombstones are cleared if the fraction of tombstones in the occupied slots is greater than 3/32, which is an empirically determined factor. Otherwise, the hash table is resized to double the capacity.

7.2 Deamortized rebuilds in AbslHT

Probing sequence. To enable deamortized rebuilding, we change the probing sequence function used in default AbslHT from $p_i = (h(a)+k*(i^2+i)/2) \pmod n$ to a linear-probing sequence $p_i = (h(a)+k*i) \pmod n$.

Deamortized rebuild schedule. As AbslHT uses unordered linear probing, the rebuild algorithm is slightly different compared to ZombieHT. Instead of rebuilding by pushing tombstones across fixed intervals of size $c_b x$, we rebuild by rehashing all items in a cluster (all items in between two consecutive empty slots) in place. We do this by keeping track of the last slot that was rebuilt l and the target rebuild size t. The target rebuild is incremented by $c_b x$ on every insert, i.e. by inserting j, we should have rebuilt at least until slot $jc_b x$. If $t < l$, the deamortized rebuilding is behind the schedule, triggering a rebuild of the next cluster after t and updating t. In the actual implementation, both t and l must handle wrap

Hash table	Consistency	Space efficiency	Data locality
ZombieHT (This paper)	✓	✓	✓
RobinHoodHT [5]		✓	✓
GraveyardHT [3]		✓	✓
CLHT [9]	✓		
AbslHT [14]			✓
IcebergHT [38]	✓		

Table 2. Hash table performance

around when the end of the hash table is reached. We only trigger rebuilding in a deamortized schedule if the current true load factor (load factor including elements and tombstones) is greater than 87.5%.

Primitive tombstones. Unlike the quotienting implementation of ZombieHT which orders run by their hash value, the ZombieHT version based on AbslHT uses unordered linear probing. While we do use a deamortized rebuild schedule, we do not introduce primitive tombstones. Introducing primitive (or artificial) tombstones in unordered linear probing has a negative effect on the performance. This is in contrast to ordered linear probing, where the lookup can exit early because items are ordered. Intuitively, in ordered linear probing, tombstones and empty slots are almost equivalent when probing for an item — empty slots are essentially tombstones at the end of a run. In unordered linear probing, empty slots are much more valuable than tombstones. This is because empty slots are the only mechanism to end a probe.

8 Experiments

In this section, we evaluate the performance of our Zombie hashing scheme. We evaluate our hash tables on space-efficiency, throughput (number of operations per second) and latency [1] (time taken by individual operations) distribution. Our experiments aim to simulate real-world aging workloads where hash tables are maintained at high load factors and go through multiple *churn cycles*. Specifically, we fill the hash table to 95% load factor and perform multiple cycles of insertions, deletions and query operations. We run aging experiments with varying read-write ratios at high-load factors. Finally, we also evaluate the overhead of rebalancing tombstones when the hash table is not full by comparing the throughput of hash tables when going from empty to full (95% load factor).

There are two major linear probing variants, ordered and unordered. We implement Zombie hashing scheme in both linear probing variants. The ordered variant is also known as Robin hood hashing. We implement the Robin hood hashing variant as a compact hash table **ZombieHT(C)** using quotienting [40] as this one of most commonly used linear probing variant due its high space efficiency [37, 39, 41–43]. We compare ZombieHT(C) against three other ordered-linear probing hashing tables schemes, Robin hood hashing with and without tombstones and graveyard hashing [3]. Our code is available as an open source repository [2]. We implement the unordered variant **ZombieHT(V)** using AbslHT [14], a state-of-the-art vectorized hash table from Google. We compare ZombieHT(V) against three state-of-the-art hash tables, AbslHT [14], CLHT [9] and IcebergHT [38].

Ordered linear probing variants. To ensure a fair comparison of various tombstone rebuild strategies, we implement ZombieHT(C) and all other ordered linear probing variants using the quotienting technique, which is described in Section 6. Therefore, the only difference in performance arises from the choice of the tombstone distribution strategy. All ordered linear-probing variants are implemented as a hash set storing 64 bit keys.

[1]Latency is measured as time to complete 50 operations instead of 1 operation to reduce the noise.
[2]https://github.com/saltsystemslab/ZombieHT

- **RobinHoodHT** does not use tombstones. It performs immediate compaction of elements after deletion.
- **TombstoneHT** is the RobinHoodHT with tombstones on deletes. However, there is no explicit redistribution of deletes.
- **GraveyardHT** use tombstones for deletes and periodically $(n/(4x))$ clears and redistributes tombstones across the entire hash table.
- **ZombieHT(C)** is the deamortized GraveyardHT we introduced in Section 4. In this implementation, we use $c_b = 1.0, c_p = 3.0$, and call redistribution following each insert operation. We choose these parameters based on theoretical and experimental evaluation (Section 8.3).

Unordered linear probing variants. We compare our unordered variants ZombieHT(V) with other state-of-the-art hash tables, including both chaining and open-addressing hash tables. To make the comparison fair, specific modifications were made for individual hash tables which are described below.

- **ZombieHT(V)** is an unordered linear probing variant with an implementation based off of AbslHT. ZombieHT(V) deamoritzes clearing of tombstones. ZombieHT(V), like AbslHT is vectorized to speed up lookups.
- **AbslHT** is a quadratic probing-based hash table and uses tombstones. We disable AbslHT's default behavior of resizing (or rehashing) when the load factor exceeds 87.5% true load factor. Instead, we disable resizing and configure AbslHT to rehash when the true load factor[3] reaches 97.5%.
- **CLHT** is a chaining based hash table. For CLHT, we use the CLHT_LB variant which uses linked list and no resizing. CLHT allocates a bucket size of 3 for each node in the linked list. We setup CLHT with $n/4$ slots to ensure that CLHT operates under comparable space usage as other hash tables.
- **IcebergHT** is a hierarchical hash table with three levels. The first level uses single hashing, second level uses two-choice hashing, and the last level is a chaining hash table. It is specifically designed to be space efficient, stable, and supports low associativity.
- **CuckooHT** is a hash table based on cuckoo hashing[36]. We use libcuckoo[24] as the reference implementation for CuckooHT. Rebuilding is disabled by preallocating the required space.

8.1 Results summary

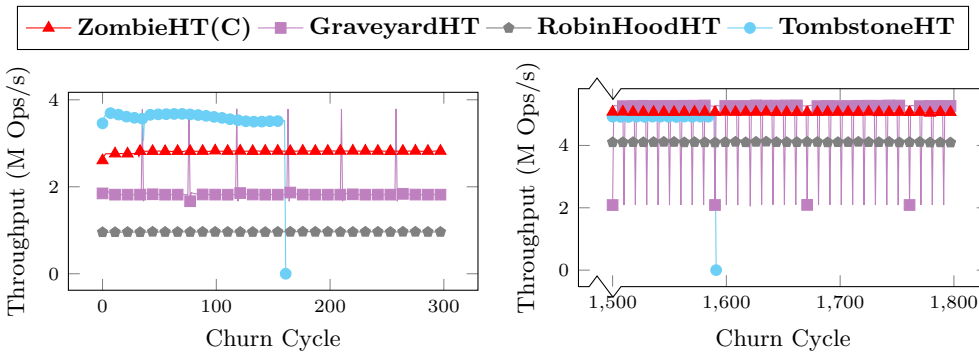
Across various experiments, we find that Zombie hashing variants achieve the most consistent performance (lowest latency variance) compared to other state-of-the-art hash tables. At high load-factors (95%), among ordered linear probing hash tables, ZombieHT(C) achieves the highest throughput (2.81M ops/s); this improves upon other ordered linear-probing hash table schemes which also incur periodic down times such as graveyard hashing (1.82M ops/s) and Robin hood hashing (0.96M ops/s).

For unordered linear probing variants, ZombieHT(V)'s max latency (140 μs) is an order magnitude lower compared to AbslHT's (1.3 sec). The standard-deviation in insert performance is similarly low at 7.31 μs, compared to AbslHT 2965.62 μs. The tradeoff for stable performance is a slightly lower average throughput (-18%) compared to base AbslHT implementation based on quadratic probing.

Hash table performance. Other than the pure throughput of hash table operations, three features dominate the decision of which hash table to use for a particular application. Table 2 lists these three features for the state-of-the-art hash tables.

Consistency means the hash table can consistently provide high throughput at high load factor without any interruption in regular operations. RobinHoodHT does not need rebuilds, but its performance deteriorates significantly (bringing it to almost a halt) at high load factors. Both GraveyardHT and AbslHT rely on rebuilds to maintain their performance. CLHT and IcebergHT do not need rebuilds and thus can offer consistent performance.

[3]The true load factor ratio of the sum of occupied slots and tombstones to the number of slots in the hash table.



(a) 50% updates and 50% queries.

(b) 5% updates and 95% queries.

Fig. 6. Overall throughput of ZombieHT(C) compared to other linear probing variants for churn experiments. Note: In Figure 6b we run the experiment for 1800 cycles but only plot the throughput for the last 300 cycles for brevity.

	Throughput (*Mops/sec*)		
HashMap	Load Phase	50% Updates 50% Lookup	5% Updates 95% Lookup
ZombieHT	7.01	2.81	5.06
GraveyardHT	6.21	1.82	4.57
RobinHoodHT	7.58	0.96	4.07
TombstoneHT (★)	7.58	3.59	4.95

Table 3. Average throughput in the load and churn phase (for various read-write ratios). (★) TombstoneHT values are before it runs out of memory and dies.

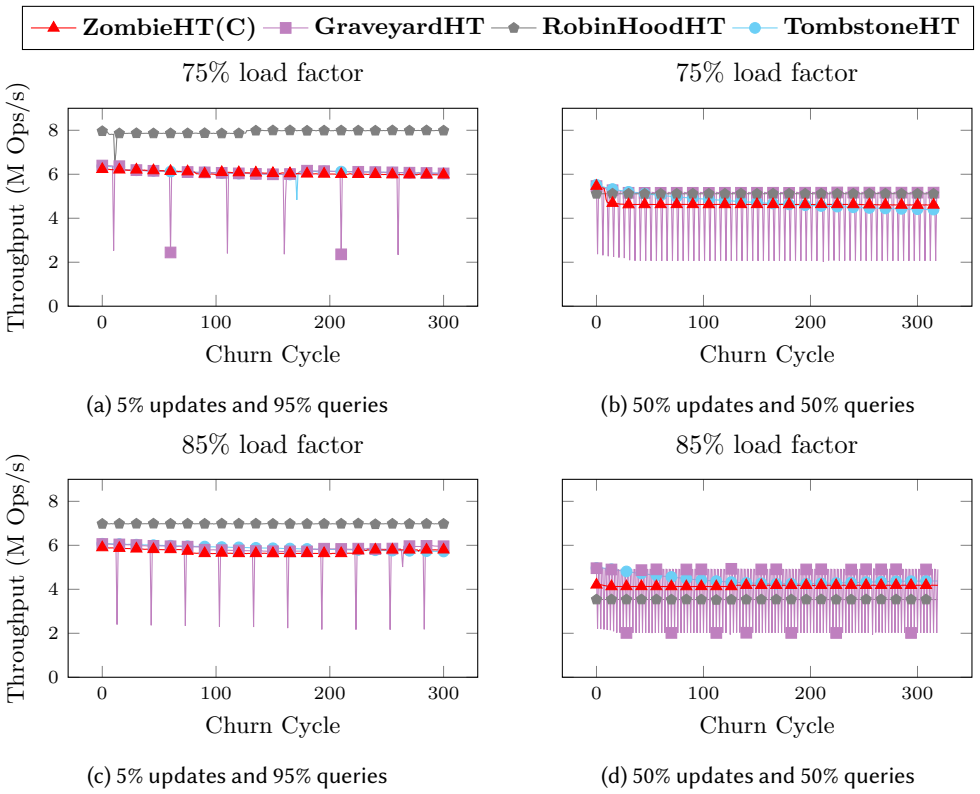
Space efficiency means the hash table is compact, asymptotically matching the information-theoretic low bound (see Section 6.2). All linear probing variants can be implemented using a compact hash table design to achieve near-information-theoretic levels of space efficiency. However, CLHT, AbslHT, and IcebergHT have high overheads and do not achieve high space efficiency.

Data locality means required data for each operation can be loaded within a few contiguous cache lines (or blocks for file based). CLHT is chaining based and has poor data locality. IcebergHT has 3 level hashes and requires a few random cache line accesses. AbslHT also requires multiple random cache line accesses at high load factors.

Only ZombieHT achieves all three features: consistency, space efficiency, and data locality and can be the go-to hash table in many applications including databases, caching, storage, etc.

8.2 Experimental setup

Workloads. Our workloads are based on YCSB [8]. They are divided into two phases: load and run (churn). In the load phase, we initialize all hash tables to 95% load factor. As mentioned in Section 2, we use $1-1/x$ for load factor, so $x=20$ when the load factor is 95%. All the hash tables are setup with 2^{27} slots. Keys are 64 bits and are generated from a uniform-random distribution.



(a) 5% updates and 95% queries

(b) 50% updates and 50% queries

(c) 5% updates and 95% queries

(d) 50% updates and 50% queries

Fig. 7. Overall throughput of ordered linear probing tables when operating 75% and 85% lower load factors.

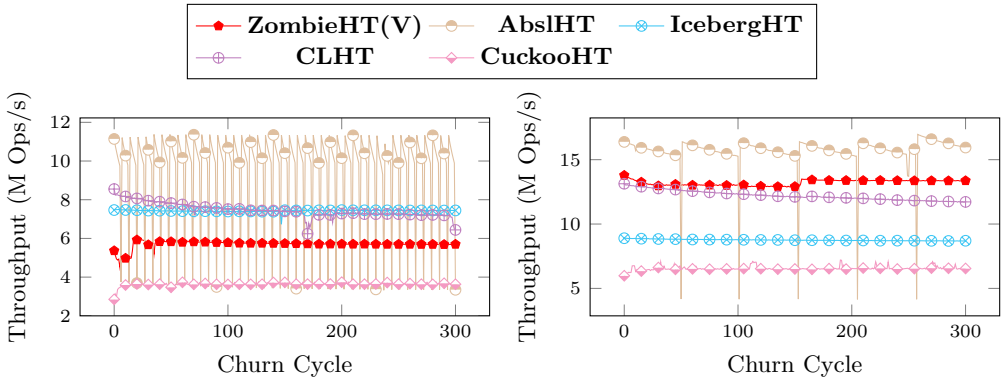
In the churn phase, operations are divided into a series of churn cycles, where each churn cycle performs a total of 5% of n (total slots in the hash table) operations. Each churn cycle first performs a sequence of delete operations, then inserts new keys and finally performs a series of lookup operations. Deletion and lookup keys are picked uniformly at random from keys in the hash table, while insertion keys are picked uniform randomly from the universe – *this models a uniform hash function.* Similar to YCSB [8], we generate two churn workloads by varying the ratio of number of update and lookup operations (50:50 YCSB-A, and 5:95 YCSB-B).

System specification. All the experiments were run on an Intel(R) Xeon(R) Gold 6338 CPU @ 2.00GHz with two NUMA nodes, 32 cores per nodes, and 48MB L3 cache per node. The machine has 1TB of DRAM running Linux kernel 5.4.0-155-generic.

8.3 Ordered linear probing

Figure 6 plots the throughput of various ordered hash tables in our churn experiments for different read-write ratios, while Figure 10 plot the microbenchmarks validating our theoretical results Section 4 and parameter study for ZombieHT(C).

Throughput. Figure 6 and Table 3 show the performance of ZombieHT(C) compared to other ordered linear probing tombstone redistribution strategies on the churn workloads of varying read-write ratios. Among all the linear probing variants, ZombieHT(C) achieves the highest overall throughput (2.81M ops/s on the 50%-50% workload) without periodic drops in throughput.



(a) 50% updates and 50% queries.

(b) 5% updates and 95% queries.

Fig. 8. Performance of ZombieHT(V) and other hash tables under different read-write ratios.

For the 50%-50% read-write workload, RobinHoodHT has the lowest overall throughput (0.96M ops/s) because of primary clustering. TombstoneHT starts out with a higher overall throughput (3.59M ops/s) but eventually runs out of space due to accumulation of tombstones, showing that redistribution of tombstones is necessary. GraveyardHT avoids running out of space with a slightly lowered throughput (1.82M ops/s) compared to TombstoneHT, but suffers from periodic drops in overall throughput with a standard deviation of 9862.98μs.

Figure 7 shows the throughput for the churn workload at 75% and 85% load factors. At lower load factors, primary clustering is less noticeable. For the read-heavy workload (5% updates, 95% lookups), RobinHoodHT achieves the highest throughput due to shorter runs as there are no tombstones and minimal impact from primary clustering on updates. However, at 85% load, update throughput decreases as primary clustering becomes more significant.

Latency Distribution. Table 5 records the latency distribution of individual hash table operations during the churn cycle experiment, averaged across all workloads. For inserts, ZombieHT has a max latency that is roughly 3000× lower than GraveyardHT, and has a much lower standard deviation (7.57μs) compared to GraveyardHT (10049.82μs). The extremely high standard deviation in GraveyardHT shows how rebuilding severely affects tail latency. The max latency in GraveyardHT is not an outlier, but an unavoidable cost that repeatedly shows up by design. The tradeoff is that deamortized redistribution in ZombieHT results in higher median latency (1.6×) than GraveyardHT.

ZombieHT redistributes during inserts only. Its median latency is 1.5× lower than RobinHoodHT on inserts. On deletes, the gains are higher with median latency 16× lower. The latency distribution for TombstoneHT is better than ZombieHT only on inserts, but is not sustainable as it soon runs out of space. RobinHoodHT has the best median lookup latency, but the absolute gains are very minimal (2μs faster for 50 operations).

Load phase throughput. Table 3 also shows the aggregate throughput of various hash tables during the load phase. The load phase fills an empty hash table to 95% load factor. We measure the insert throughput as the load factor of the hash table increases. ZombieHT has similar throughput (7.01M ops/s) compared to RobinHoodHT (7.58M ops/s) and TombstoneHT (7.58M ops/s). During the load phase, the deamortization schedule kicks in only when the true load factor (tombstones and items) exceeds 80%. We determine this cutoff threshold empirically by running the churn test at load factors ranging from 70% to 95%.

	Throughput ($Mops/sec$)		
HashMap	Load Phase	50% Updates 50% Lookup	5% Updates 95% Lookup
ZombieHT(V)	13.41	6.91	13.21
AbslHT	15.67	9.81	15.08
IcebergHT	7.43	6.00	8.75
CLHT	12.68	7.42	12.71
CuckooHT	5.90	3.63	6.54

Table 4. Average throughput in the load and churn phase (for various read-write ratios) of unordered linear probing tables.

8.4 Unordered linear probing

We now compare the unorderd variant with vectorization ZombieHT(V) against four state-of-the-art production hash tables.

Update throughput and latency. Figure 8 and Table 5 show the throughput and latency distribution of hash table operations of ZombieHT(V) and other hash tables. ZombieHT(V) achieves a similar throughput (6.91M ops/s) compared to AbslHT (9.81M ops/s) without suffering from the extreme maximum and standard deviation of latency (140.79μs, 7.31μs) suffered by AbslHT (1199985.97μs, 2695.93μs). CLHT too has a higher max latency (15427.46μs) on inserts, which is incurred when CLHT needs to allocate more buckets for a slot. IcebergHT does not suffer from having high max latency (66.68μs), but has lower space efficiency.

Figure 9 shows the throughput of various hash tables at lower load factors. ZombieHT(V) and AbslHT achieve the highest throughput with excellent space efficiency. On write-heavy workloads, ZombieHT(V) avoids stopping and rebuilding, while AbslHT doesn't rebuild on read-heavy workloads unless the true load factor exceeds 87.5%, which doesn't occur during lower-load churn tests.

Load phase throughput. Table 4 lists the throughput of various hash tables in the load phase as hash tables are filled upto 95% load factor. AbslHT (15.67M ops/s) and ZombieHT(V) (13.41M ops/s), being vectorized hash tables have higher throughput compared to other hash tables. AbslHT uses quadratic probing while ZombieHT(V) uses linear probing. ZombieHT(V) applies a deamortized rebuilding schedule once the load factor exceeds 87.5%.

8.5 Space efficiency

Space efficiency is the ratio of data size over hash table size. Here the data size is defined as the information-theoretical lower bound, which is $\log\binom{u}{t}$ bits, where $u = 2^{64}$ is the universe size, t is n times load factor 0.95, and $n = 2^{27}$. Table 6 shows the size of the hash table at 95% load factor. As a compact quotienting-based hash table, ZombieHT uses the lowest space (1.63 GB) among all the other hash tables with a space efficiency of 92.12%. Note that, ZombieHT is implemented as a hash set, the space reported above is space ZombieHT would need as a hash table with 64 bit values. CLHT has low space efficiency and does not achieve a higher space efficiency than 35.66%. CLHT allocates buckets which are cache aligned to be performant, but all buckets are not always filled. AbslHT, ZombieHT(V) and IcebergHT are not quotienting-based, and they store the entire key in the slots. CuckooHT is based on the open-addressing scheme has similar space efficiency, but it is not as performant as ZombieHT(V).

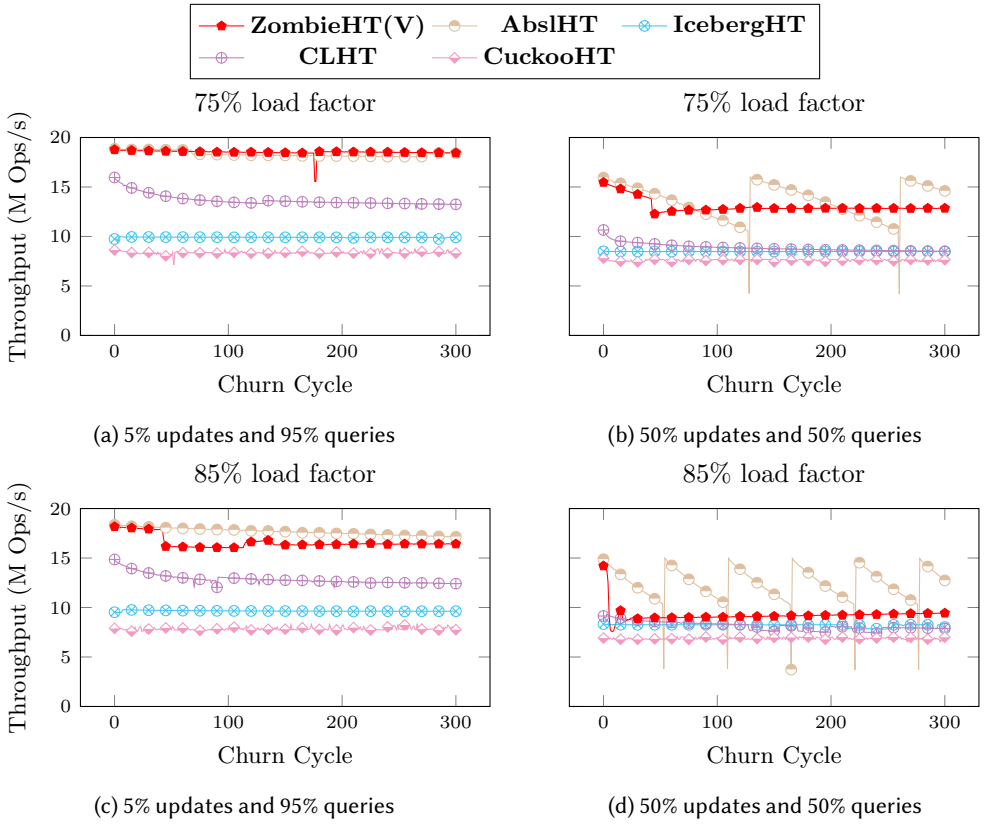


Fig. 9. Throughput of ZombieHT(V) and other hash tables tables when operating 75% and 85% lower load factors.

8.6 Microbenchmarks

Figure 10 plots various internal statistics that were captured to validate the theoretical results we show in Section 4. Specifically, we compare the optimizations and configuration parameters to choose the most practical version of ZombieHT(C). The microbenchmarks run the read heavy workload.

ZombieHT(C) parameters. In Section 4, to simplify the analysis, all the theoretical bounds are derived with a relative large rebuild interval size $c_b x$ with $c_b = 6$. In practice, a smaller c_b is feasible although the theoretical analysis is cumbersome. The rebuild interval for ZombieHT can be as small as polylog(x) as we know that a large rebuild window of $n/\text{polylog}(x)$ provides better performance than a small rebuild window $n/O(x)$ [3] for Robin hood hashing with tombstones. We run experiments with different c_b and the smaller c_b gives the best update performance while not hurting the lookups significantly. *Thus, we choose $c_b = 1.0$ in ZombieHT(C) as the rebuild window size in as our default value in our evaluations.*

The primitive tombstone space parameter c_p can be tuned according to the workload. A bigger c_p results in better read performance as there as less tombstones at the cost of higher inserts and conversely write heavy workloads should prefer a smaller c_p. Unless otherwise stated, *we use ZombieHT(C) with $c_p = 3.0$ in the experiments.*

Insert (μs)									
Percentile	**ZombieHT(C)**	RobinHoodHT	TombstoneHT ⋆	GraveyardHT	**ZombieHT(V)**	AbslHT	CLHT	IcebergHT	CuckooHT
Min	30.33	24.76	16.19	14.59	7.31	5.08	5.40	7.02	16.84
50%	42.43	63.01	25.92	25.00	24.04	8.73	8.03	9.32	36.34
99.99%	188.18	134.68	53.67	52.18	82.12	16.06	8869.38	14.88	66.35
Max	349.90	1700.60	INF	2173487.96	140.79	1308525.04	15427.46	66.68	102.36
Std dev.	7.57	12.42	4.38	9862.98	7.31	2965.62	199.92	0.65	6.15
Deletes (μs)									
Percentile	ZombieHT(C)	RobinHoodHT	TombstoneHT ⋆	GraveyardHT	ZombieHT(V)	AbslHT	CLHT	IcebergHT	CuckooHT
Min	7.36	36.37	8.24	7.27	2.96	2.92	3.19	6.19	4.36
50%	9.39	133.75	10.60	9.34	4.33	4.22	5.27	8.28	5.59
99.99%	17.57	324.94	19.03	17.76	9.84	9.45	13.11	15.65	13.24
Max	68.04	1848.03	59.41	58.47	71.22	36.29	271.30	56.87	49.35
Std dev.	0.81	34.44	0.88	0.79	0.61	0.58	1.03	0.72	0.56
Lookup (μs)									
Percentile	ZombieHT(C)	RobinHoodHT	TombstoneHT ⋆	GraveyardHT	ZombieHT(V)	AbslHT	CLHT	IcebergHT	CuckooHT
Min	6.66	5.28	7.25	6.54	2.23	2.16	2.65	4.06	4.39
50%	8.52	7.00	9.45	8.42	3.36	3.27	4.24	5.68	6.83
99.99%	13.43	13.39	19.87	13.91	7.61	7.63	9.53	10.66	12.42
Max	74.74	1622.36	1652.95	70.25	70.03	73.84	272.79	72.58	72.30
Std dev.	0.56	1.13	1.24	0.55	0.41	0.38	0.70	0.49	0.67

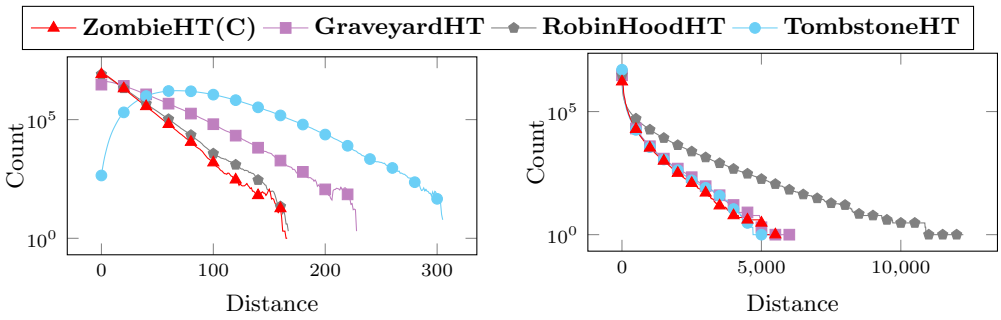
Table 5. Latency distribution of various hash table operations collected over 100 churn cycles. Latency is measured as the time taken to complete 50 operations. TombstoneHT ⋆ did not finish the churn experiment (max insert latency is infinity).

Hash table	Size	Space efficiency
ZombieHT(C)	1.63 GB	92.12%
CuckooHT	2.00 GB	74.96%
ZombieHT(V)	2.13 GB	70.55%
AbslHT	2.13 GB	70.55%
IcebergHT	2.39 GB	62.71%
CLHT	4.20 GB	35.66%

Table 6. Space efficiency of various hash tables at 95% load factor. Hash tables are configured to have $N = 2^{27}$ slots.

ZombieHT(C) vs ZombieHTDelete. Figure 11 also compares the overall performance on the churn workload of two ZombieHT variants (the default uses $c_b = 1.0$, while the variant uses $c_b = 3.0$) and ZombieHTDelete optimization introduced in Section 5. The ZombieHTDelete achieves similar throughput to ZombieHT(C), however without that the theoretical guarantees that ZombieHT(C) provides.

Home Slot and tombstone distance distribution. Figure 10a and Figure 10b plot the distribution of the distances of slot items to their (1) home slot and (2) nearest available (tombstone or free slot) after 100 churn cycles respectively. Note the log-scale on the y-axis. Higher home slot distance corresponds to worse lookup performance, which explains why ordered linear probing variants with tombstones have a slightly higher median lookup latency. Notably, because TombstoneHT does not manage its tombstones, elements drift further and further away from their home slot with more operations are performed and eventually runs out of space. A high available slot distance implies that the more items must be shifted to do an insert (and deletes in RobinHoodHT) operation. From Figure 10b, it can be seen that cluster sizes in RobinHoodHT can be as high as 12000 because of not having the anti-clustering properties of tombstones. ZombieHT(C) achieves roughly the same latency distribution as GraveyardHT, but with the above discussed advantages of lower variance and improved throughput.



(a) Distance from home slot (b) Distance to available slot

Fig. 10. Hash table microbenchmarks. Higher distance between home slot to next nearest available slot key corresponds to worse insert performance.

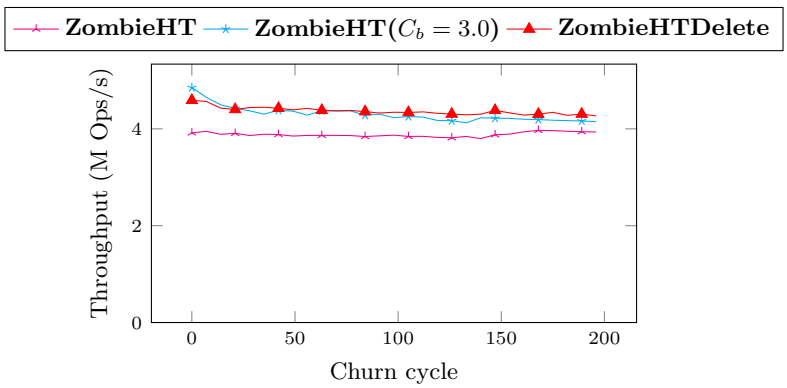


Fig. 11. Evaluating rebuild interval size(C_b) and tombstone space distance (C_p) parameters for ZombieHT(C), and comparing against ZombieHTDelete.

9 Discussion & Conclusion

In this paper, we develop a new hashing scheme (Zombie hashing) that achieves consistent high throughput, high space efficiency, and high data locality for real-world workloads for hundreds for churn cycles. We design a novel deamortization scheme and prove theoretical bounds that each operation always operates with low, bounded cost, $\Theta(x)$, where $1-1/x$ is the load factor, resulting in consistent performance. ZombieHT is an ideal candidate for applications that require strong consistent performance guarantees and are constrained on space. Thus ZombieHT is suited to accelerate large-scale data processing (OLTP) which benefit from high data locality.

Other modern hash tables, such as IcebergHT achieve consistent high throughput and can be the appropriate choice in many applications; however, they do not achieve high space efficiency and may not be the right choice in applications where memory is scarce. Furthermore, IcebergHT gives up on data locality and hash ordering which is a critical feature in many data analysis systems to quickly compute hash table joins and merges.

Acknowledgments

This research is funded in part by NSF grant OAC 2339521 and III 2311954.

References

[1] Rizwan A. Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F. DeMara, Chen-Yong Cher, and Pradip Bose. 2015. Understanding the propagation of transient errors in HPC applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, Austin Texas, 1–12. doi:10.1145/2807591.2807670

[2] Michael A. Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, and Guido Tagliavini. 2023. Iceberg Hashing: Optimizing Many Hash-Table Criteria at Once. *J. ACM* 70, 6 (2023), 40:1–40:51. doi:10.1145/3625817

[3] Michael A. Bender, Bradley C. Kuszmaul, and William Kuszmaul. 2021. Linear Probing Revisited: Tombstones Mark the Demise of Primary Clustering. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*. IEEE, 1171–1182. doi:10.1109/FOCS52979.2021.00115

[4] Jose Nelson Perez Castillo, Miguel Gutierrez, and Nelson Enrique Vera Parra. 2016. Computational Performance Assessment of k-mer Counting Algorithms. *J. Comput. Biol.* 23, 4 (2016), 248–255. doi:10.1089/CMB.2015.0199

[5] Pedro Celis, Per-Åke Larson, and J. Ian Munro. 1985. Robin Hood Hashing (Preliminary Report). In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*. IEEE Computer Society, 281–288. doi:10.1109/SFCS.1985.48

[6] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. 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, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 275–290. doi:10.1145/3183713.3196898

[7] John G. Cleary. 1984. Compact Hash Tables Using Bidirectional Linear Probing. *IEEE Trans. Computers* 33, 9 (1984), 828–834. doi:10.1109/TC.1984.1676499

[8] 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. doi:10.1145/1807128.1807152

[9] 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, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, 631–644. doi:10.1145/2694344.2694359

[10] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, Beatrice Yormark (Ed.). ACM Press, 1–8. doi:10.1145/602259.602261

[11] DynamoDB 2024. https://aws.amazon.com/dynamodb/. Accessed: 2024-10-06.

[12] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, Nick Feamster and Jeffrey C. Mogul (Eds.). USENIX Association, 371–384. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan

[13] H. Garcia-Molina and K. Salem. 1992. Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering* 4, 6 (Dec. 1992), 509–516. doi:10.1109/69.180602 Conference Name: IEEE Transactions on Knowledge and Data Engineering.

[14] Google. 2024. Abseil / Abseil Containers. https://abseil.io/docs/cpp/guides/container, Last accessed 2024-10-06.

[15] Philipp M. Grulich, Sebastian Breß, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2487–2503. doi:10.1145/3318464.3389739

[16] F. Robert A. Hopgood and James H. Davenport. 1972. The quadratic hash method when the table size is a power of 2. *Comput. J.* 15, 4 (1972), 314–315. doi:10.1093/COMJNL/15.4.314

[17] Rosa M. Jiménez and Conrado Martínez. 2018. On Deletions in Open Addressing Hashing. In *2018 Proceedings of the Meeting on Analytic Algorithmics and Combinatorics (ANALCO)*. Society for Industrial and Applied Mathematics, 23–31. doi:10.1137/1.9781611975062.3

[18] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. 191–205. https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet

[19] Don Knuth. 1963. Notes On "Open" Addressing.

[20] Donald Ervin Knuth. 1997. *The art of computer programming*. Vol. 3. Pearson Education.

[21] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the walkers: accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. Association for Computing Machinery, New York, NY, USA, 468–479. doi:10.1145/2540708.2540748

[22] Piyush Kumar, Mobashshirur Rahman, Suyel Namasudra, and Nageswara Rao Moparthi. 2023. Enhancing Security of Medical Images Using Deep Learning, Chaotic Map, and Hash Table. *Mobile Networks and Applications* (Sept. 2023). doi:10.1007/s11036-023-02158-y

[23] Dean De Leo and Peter A. Boncz. 2021. Teseo and the Analysis of Structural Dynamic Graphs. *Proc. VLDB Endow.* 14, 6 (2021), 1053–1066. doi:10.14778/3447689.3447708

[24] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic improvements for fast concurrent Cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 27, 14 pages. doi:10.1145/2592798.2592820

[25] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 21–35. doi:10.1145/3035918.3064015

[26] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. 2021. Sparta: high-performance, element-wise sparse tensor contraction on heterogeneous memory. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, Jaejin Lee and Erez Petrank (Eds.). ACM, 318–333. doi:10.1145/3437801.3441581

[27] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Transactions on Storage* 13, 1 (March 2017), 5:1–5:28. doi:10.1145/3033273

[28] Tobias Maier, Peter Sanders, and Roman Dementiev. 2019. Concurrent Hash Tables: Fast and General(?)! *ACM Trans. Parallel Comput.* 5, 4 (2019), 16:1–16:32. doi:10.1145/3309206

[29] Guillaume Marçais and Carl Kingsford. 2011. A fast, lock-free approach for efficient parallel counting of occurrences of *k*-mers. *Bioinform.* 27, 6 (2011), 764–770. doi:10.1093/BIOINFORMATICS/BTR011

[30] Gabriel Mateescu, Wolfgang Gentzsch, and Calvin J. Ribbens. 2011. Hybrid Computing—Where HPC meets grid and Cloud Computing. *Future Generation Computer Systems* 27, 5 (May 2011), 440–453. doi:10.1016/j.future.2010.11.003

[31] Hunter McCoy, Steven A. Hofmeyr, Katherine A. Yelick, and Prashant Pandey. 2023. Singleton Sieving: Overcoming the Memory/Speed Trade-Off in Exascale κ-mer Analysis. In *SIAM Conference on Applied and Computational Discrete Algorithms, ACDA 2023, Seattle, WA, USA, May 31 - June 2, 2023*, Jonathan W. Berry, David B. Shmoys, Lenore Cowen, and Uwe Naumann (Eds.). SIAM, 213–224. doi:10.1137/1.9781611977714.19

[32] Memcached 2024. Memcached. https://memcached.org/. Accessed: 2024-10-06.

[33] Michael Mitzenmacher and Eli Upfal. 2017. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press.

[34] 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, EuroSys 2023, Rome, Italy, May 8-12, 2023*, Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan (Eds.). ACM, 817–834. doi:10.1145/3552326.3587457

[35] Anna Pagh, Rasmus Pagh, and Milan Ruzic. 2009. Linear Probing with Constant Independence. *SIAM J. Comput.* 39, 3 (2009), 1107–1120. doi:10.1137/070702278

[36] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo Hashing. *J. Algorithms* 51, 2 (May 2004), 122–144. doi:10.1016/j.jalgor.2003.12.002

[37] Prashant Pandey, Fatemeh Almodaresi, Michael A Bender, Michael Ferdman, Rob Johnson, and Rob Patro. 2018. Mantis: A fast, small, and exact large-scale sequence-search index. *Cell systems* 7, 2 (2018), 201–207.

[38] 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. *Proceedings of the ACM on Management of Data* 1, 1 (May 2023), 47:1–47:26. doi:10.1145/3588727

[39] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. 2017. deBGR: an efficient and near-exact representation of the weighted de Bruijn graph. *Bioinformatics* 33, 14 (2017), i133–i141. doi:10.1093/BIOINFORMATICS/BTX261

[40] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. 2017. A General-Purpose Counting Filter: Making Every Bit Count. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, Chicago Illinois USA, 775–787. doi:10.1145/3035918.3035963

[41] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. 2018. Squeakr: an exact and approximate k-mer counting system. *Bioinform.* 34, 4 (2018), 568–575. doi:10.1093/BIOINFORMATICS/BTX636

[42] Prashant Pandey, Yinjie Gao, and Carl Kingsford. 2021. VariantStore: An Index for Large-Scale Genomic Variant Search. 22, 1 (2021), 231. doi:10.1186/s13059-021-02442-8

[43] Prashant Pandey, Shikha Singh, Michael A. Bender, Jonathan W. Berry, Martin Farach-Colton, Rob Johnson, Thomas M. Kroeger, and Cynthia A. Phillips. 2020. Timely Reporting of Heavy Hitters using External Memory. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA],*

June 14-19, 2020, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1431–1446. doi:10.1145/3318464.3380598

[44] Johannes Pietrzyk, Annett Ungethüm, Dirk Habich, and Wolfgang Lehner. 2019. Fighting the Duplicates in Hashing: Conflict Detection-aware Vectorization of Linear Probing. P-289 (2019), 35–53. doi:10.18420/BTW2019-04

[45] Sylvia Ratnasamy, Brad Karp, Li Yin, Fang Yu, Deborah Estrin, Ramesh Govindan, and Scott Shenker. 2002. GHT: a geographic hash table for data-centric storage. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications (WSNA '02)*. Association for Computing Machinery, New York, NY, USA, 78–87. doi:10.1145/570738.570750

[46] Redis 2024. Redis. https://redis.io/. Accessed: 2024-10-06.

[47] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. *Proc. VLDB Endow.* 9, 3 (2015), 96–107. doi:10.14778/2850583.2850585

[48] G. Schay and W. G. Spruth. 1962. Analysis of a file addressing method. *Commun. ACM* 5, 8 (Aug. 1962), 459–462. doi:10.1145/368637.368827

[49] Malte Skarupke. 2017. I Wrote The Fastest Hashtable. https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/. Accessed: 2025-01-22.

[50] Swiss Table Notes 2024. abseil / Swiss Tables Design Notes. https://abseil.io/about/design/swisstables. Accessed: 2024-10-06.

[51] Sebastian Sylvan. 2013. Robin Hood Hashing should be your default Hash Table implementation. https://www.sebastiansylvan.com/post/robin-hood-hashing-should-be-your-default-hash-table-implementation/. Accessed: 2025-01-22.

[52] Hui Tian, Yuxiang Chen, Chin-Chen Chang, Hong Jiang, Yongfeng Huang, Yonghong Chen, and Jin Liu. 2017. Dynamic-Hash-Table Based Public Auditing for Secure Cloud Storage. *IEEE Transactions on Services Computing* 10, 5 (Sept. 2017), 701–714. doi:10.1109/TSC.2015.2512589 Conference Name: IEEE Transactions on Services Computing.

[53] Sujatha R. Upadhyaya. 2013. Parallel approaches to machine learning—A comprehensive survey. *J. Parallel and Distrib. Comput.* 73, 3 (March 2013), 284–292. doi:10.1016/j.jpdc.2012.11.001

[54] Aho A. V. 1986. Compilers Principles. *Techniques, and Tools* (1986). https://cir.nii.ac.jp/crid/1570572699852468736 Publisher: Addison-Wesley publishing company.

[55] Wikipedia. 2024. MurmurHash. https://en.wikipedia.org/wiki/MurmurHash. Accessed: 2024-10-06.

[56] Wikipedia. 2024. Streaming SIMD Extensions. https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions Accessed 2024-10-06.

[57] Helen Xu, Amanda Li, Brian Wheatman, Manoj Marneni, and Prashant Pandey. 2023. BP-Tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B-Trees. *Proc. VLDB Endow.* 16, 11 (aug 2023), 2976–2989. doi:10.14778/3611479.3611502

[58] 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 (2016), 301–312. doi:10.14778/3025111.3025113

Received October 2024; revised January 2025; accepted February 2025