

Treap-Augmented Skip List for Fast Hot-Key Lookup under Skewed Workloads

Seobum Jeong

Computer Science & Engineering
Sungkyunkwan University
Suwon, Republic of Korea
bumi0000@g.skku.edu

Simon S. Woo

Computer Science & Engineering
Sungkyunkwan University
Suwon, Republic of Korea
swoo@g.skku.edu

ABSTRACT

Skip lists are widely deployed in production systems such as Redis and RocksDB, which is a fundamental foundation for big data systems and data sciences. However, skip lists generally treat all keys equally regardless of access frequency under Zipfian workloads—where a small fraction of keys dominates query traffic—this uniformity leads to unnecessary traversal overhead for hot keys. Prior distribution-adaptive approaches modify the internal structure of the skip list, increasing implementation complexity and potentially interfering with range-query performance. We propose the **Treap-Augmented Skip List (TASL)**, a two-tier architecture that places an external treap as a hot-tier cache above an unmodified skip list. TASL is designed to optimize point lookup performance under skewed workloads, while preserving the original range-query behavior by delegating all range operations directly to the skip list. We evaluate TASL against a plain skip list and a standalone plain treap using synthetic Zipfian workloads. Our experiments show that TASL’s relative performance improves as workload skewness increases. Under highly skewed workloads, TASL outperforms the plain skip list, the standalone treap, and the evaluated hybrid structures for point lookups. For range queries, TASL closely matches the performance of the plain skip list, confirming that the hot-tier introduces no additional overhead.

CCS CONCEPTS

• Information systems → Data structures • Theory of Computation → Data structures design and analysis

KEYWORDS

Skip list, treap, hot-tier cache, Zipfian workload, two-tier data structure, eviction policy

ACM Reference format:

Seobum Jeong and Simon S. Woo. 2026. Treap-Augmented Skip List For Fast Hot-Key Lookup under Skewed Workloads: An External Treap Hot Tier for Skewed Access Patterns. In *Proceedings of KDD 2026 Undergraduate Consortium (KDD-UC '26)*. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/1234567890>

1. Introduction

To support big data systems and enable modern AI systems for a wide range of applications, efficient data structure is critical for storing and managing a larger data. In particular, the skip list [1] and its variants are popularly used for a number of commercial and real-world applications. The skip list, introduced by Pugh in 1990, is a probabilistic data structure that supports search, insertion, and deletion in $O(\log n)$ expected time. Its simplicity has led to widespread adoption in systems including Redis, LevelDB, and RocksDB. However, the skip list has a fundamental limitation: node levels are assigned randomly at insertion time and never revised. The structure is entirely oblivious to query frequency.

In practice, real-world query distributions are rarely uniform. Zipfian distributions, which model many cache, database, and key-value workloads, imply that a small fraction of keys receives the vast majority of queries [2]. Under such skewed workloads, frequently accessed hot keys must traverse $O(\log n)$ nodes on every query, accumulating unnecessary overhead.

Existing adaptive approaches address this by modifying the skip list internally. The Splay-list [3] dynamically adjusts node heights based on access frequency, drawing inspiration from the self-adjusting properties of binary search trees [4]. While effective for point lookups, this approach entangles access history with the sorted structure, adding implementation complexity and potentially disrupting range-query behavior.

In this work, we take a different approach. Rather than modifying the skip list, we augment it with an external hot-tier. We propose the Treap-Augmented Skip List (TASL), which maintains an unmodified skip list as the cold tier and a treap [5] as the hot tier (See **Figure 1**). Lookups first probe the treap; on a miss, they fall through to the skip list. Nodes are admitted to the treap after their access count exceeds a threshold, and capacity is controlled by eviction policy.

The key insight is that the two structures are structurally complementary: the skip list excels at range queries via its sorted linked structure, while the treap provides $O(\log k)$ point lookups over a small hot set. Furthermore, by leveraging the cache-friendly

nature of balanced trees [6], TASL can effectively reduce the average memory access latency. By layering them, TASL inherits both strengths without sacrificing either.

We make the following contributions:

1. We propose TASL, the two-tier architecture combining a treap hot-tier with an unmodified skip list cold-tier, and compare it against the plain skip list, the plain treap, and several other hybrid structures.
2. We derive an analytical hit-rate condition predicting when the treap tier provides a net benefit for point lookup, extending classical self-organizing search heuristics [7].
3. We compare three eviction policies and show that random eviction offers the best practical trade-off.

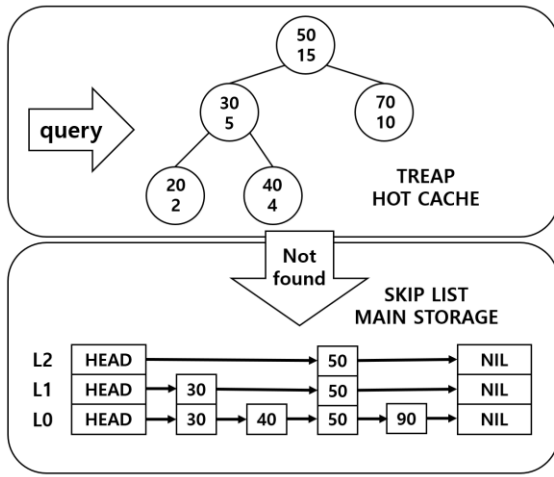


Figure 1: Overview of Treap-Augmented Skip List

2. Related Works

Pugh [1] introduced the skip list as a probabilistic alternative to balanced trees. The Splay-list [3] is the most closely related work, adapting node heights dynamically within the skip list based on access frequency. TASL differs in that it does not modify the skip list at all, preserving its range-query behavior. The Biased Skip List [8] assigns per-key weights to prioritize hot keys, but it requires these weights to be known in advance, which is often impractical in dynamic workloads.

The concept of self-organizing data structures dates back to Rivest’s study on sequential search heuristics [7], which was later extended to trees by Sleator and Tarjan [4]. While these studies focus on reorganizing a single structure, TASL utilizes a decoupled approach. The treap was introduced by Seidel and Aragon [5] as a randomized BST with expected $O(\log n)$ height. Prior work has also explored the cache performance of treap variants [6], suggesting that small, frequently accessed treaps can significantly outperform larger, cache-oblivious structures. To our knowledge, combining a treap as an external hot-tier over a skip list to optimize for Zipfian workloads has not been previously proposed.

3. Our Approach

3.1 Complementary Structures

The skip list and treap share $O(\log n)$ expected complexity for point lookup, yet their strengths are complementary. The skip list maintains a globally sorted structure enabling $O(\log n + k)$ range queries, but treats all point lookups equally. The treap provides $O(\log k)$ point lookups over k hot keys but does not support efficient range queries. TASL exploits this complementarity: the treap handles hot point lookups, and the skip list handles range queries and cold-key lookups.

3.2 TASL Architecture

TASL maintains two structures: an unmodified skip list storing all N keys, and a treap storing at most K keys. On a lookup for key x : (1) the treap is probed first; on a hit, the result is returned. (2) On a miss, the skip list is traversed. If found, the access count is incremented; if the count reaches threshold θ , x is admitted to the treap. If the treap is at capacity K , the eviction policy is applied before admission.

Algorithm 1 TASL Lookup and Admission

Require: key x

State: base skip list S , hot treap T , resident set R , access counts C , hot-tier capacity K , admission threshold θ , eviction policy π

```

1: if  $T.\text{CONTAINS}(x)$  then
2:   return FOUND
3: end if
4: if not  $S.\text{CONTAINS}(x)$  then
5:   return NOTFOUND
6: end if
7:  $C[x] \leftarrow C[x] + 1$ 
8: if  $C[x] \geq \theta$  and  $x \notin R$  then
9:   if  $|R| = K$  then
10:     $v \leftarrow \text{SELECTVICTIM}(R, \pi)$ 
11:     $T.\text{DELETE}(v)$ 
12:     $R \leftarrow R \setminus \{v\}$ 
13:   end if
14:    $T.\text{INSERT}(x)$ 
15:    $R \leftarrow R \cup \{x\}$ 
16: end if
17: return FOUND

```

3.3 Eviction Policies

We implement and compare three eviction policies. Random eviction selects a victim by performing a random walk from the treap root, terminating at any node with probability proportional to the branching structure. This yields $O(\log k)$ eviction cost with no auxiliary data structures, keeping the per-operation overhead minimal. Least Recently Used (LRU) eviction maintains a doubly linked list ordered by recency and a hash map from key to list iterator, enabling $O(1)$ victim selection (the tail of the list). On every treap hit, the corresponding entry is moved to the head of the

list. Least Frequently Used (LFU) eviction maintains per-frequency buckets implemented as doubly linked lists, along with a hash map from key to (frequency, iterator) pair and a min-frequency pointer. This supports $O(1)$ victim selection by evicting the tail of the minimum-frequency bucket, and $O(1)$ frequency update on each treap hit. LFU is theoretically optimal in terms of hit rate but incurs significant constant-factor overhead from maintaining the frequency bookkeeping structures on every access.

4. Our Analysis

4.1 Point Lookup Condition

The average point lookup latency, $E[T]$, is modeled as the treap search time T_t plus the skip list search time T_s incurred on a treap miss, where p denotes the treap hit rate.

$$E[T] = T_t + (1 - p)T_s \quad (1)$$

For TASL to outperform a plain skip list, its expected lookup latency must be lower than T_s :

$$T_t + (1 - p)T_s < T_s$$

which yields the following hit-rate condition:

$$p > \frac{T_t}{T_s} \quad (2)$$

We next estimate p under a Zipfian workload. Let N denote the total number of keys. The generalized harmonic number is defined as

$$H_{N,\alpha} = \sum_{i=1}^N \frac{1}{i^\alpha} \quad (3)$$

The cumulative access probability of the top K keys is represented by $H_{K,\alpha}/H_{N,\alpha}$.

Let Q denote the total number of queries. Under the Zipf distribution, the effective number of hot keys expected to be accessed at least θ times is approximated as

$$H_{eff} \approx \left(\frac{Q}{\theta H_{N,\alpha}} \right)^{\frac{1}{\alpha}} \quad (4)$$

Assuming that the resident hot-tier keys approximately correspond to the most frequently accessed keys, the treap hit rate is estimated as

$$p \approx \frac{H_{\min(K, H_{eff}), \alpha}}{H_{N,\alpha}} \quad (5)$$

4.2 Memory Consumption Analysis

In this section, we analyze the practical memory overhead of TASL by comparing the node structures across the two tiers. The primary spatial advantage of TASL is that it localizes the structural augmentation to a small, fixed-capacity hot tier, rather than modifying the entire cold-tier structure. At the same time, the practical implementation also requires auxiliary metadata for admission and residency tracking, which must be considered separately from the hot-tier nodes themselves.

4.2.1 Structural Metadata Distribution

The baseline memory footprint is dominated by the skip list, which contains N nodes. In our implementation, each skip-list node S_{sl} maintains a fixed array of pointers to support $O(\log N)$ search. TASL does not introduce additional structural pointers into the N cold-tier nodes. Instead, the main structural augmentation is confined to the K nodes of the treap-based hot tier, where each node S_{tr} stores a priority value and two child pointers for balancing.

In addition to this structural hot tier, the current implementation maintains lightweight global metadata for admission control and residency tracking, such as access counters and resident-key bookkeeping. Therefore, the practical memory overhead is not determined solely by the treap nodes.

4.2.2 Linear Marginal Cost with Respect to K

From a structural point of view, the additional hot-tier cost scales linearly with the capacity K . Since K is a tunable parameter that can be guided by the H_{eff} analysis to capture the head of a Zipfian distribution, the treap size remains significantly smaller than the total dataset ($K \ll N$). Structurally, S_{tr} is more compact than S_{sl} , because skip-list nodes require a high density of forward pointers, whereas treap nodes maintain only two child pointers and a priority field.

Accordingly, the asymptotic structural overhead of TASL is dominated by the $O(K)$ hot tier, although the implementation-level memory usage also includes auxiliary bookkeeping beyond the treap itself.

4.2.3 Practical Overhead Ratio

The relative overhead can be expressed as $\Delta M \approx (K \times S_{tr}) / (N \times S_{sl})$. To validate this estimate empirically, we measured the private memory footprint of TASL against a plain skip list under our experimental workload. Under steady-state operation with a fully resident hot tier, TASL incurs a 6.8% increase in total memory — including not only the treap structure itself but also the admission-control and residency-tracking metadata required in practice. This controlled growth of the "hot" metadata relative to the "cold" data ensures that TASL provides substantial lookup latency reductions with a modest impact on the overall system memory budget.

5. Experimental Results

5.1 Setup

We implement several structures in C++: plain skip list (baseline), plain treap ($N = 1,000,000$ keys, standalone), and TASL (treap hot-tier + skip list cold-tier). All structures store $N = 1,000,000$ keys. Point lookup experiments use $Q = 1,000,000$ queries drawn from Zipf(α) with $\alpha \in \{0.5, 0.6, \dots, 1.5\}$. TASL parameters: admission threshold $\theta = 3$, treap capacity $K = 30,000$ (3%), random eviction. All experiments run 5 times; we report the mean latency. Hot-key latency is measured over the top 250 most frequent keys.

5.2 Point Lookup

Figure 2 illustrates the average point lookup latency across different skewness levels. In the low-skewness region ($\alpha = 0.5$ to 0.7), where data access is relatively uniform, SkipList + Hash and SkipList + SortedArray show competitive performance. This is because the overhead of hash computation or binary search is offset by direct access to the hot cache.

However, as the workload becomes more highly skewed ($\alpha \geq 0.8$), the proposed SkipList + Treap (TASL) begins to significantly outperform all other structures, while deterministic structures like RBTREE and Hash suffer from severe performance degradation.

In extreme-skewness region ($\alpha \geq 1.3$), TASL is the only evaluated hybrid structure that outperforms both the plain skip list and the standalone treap.

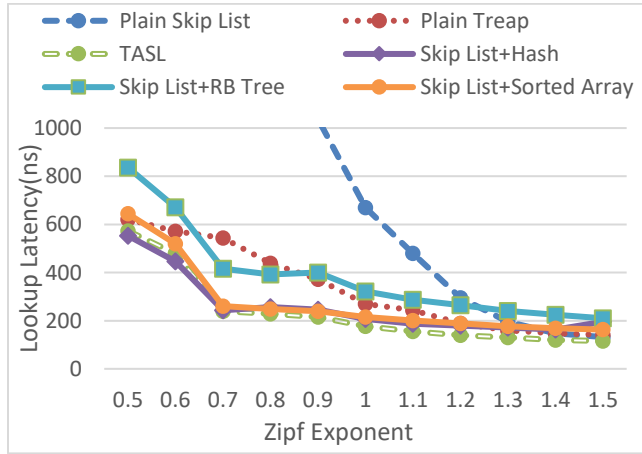


Figure 2: Average Point Lookup Latency across Zipf Exponent.

Figure 3 illustrates the hot key point lookup latency across different skewness levels. In the low-skewness region ($\alpha = 0.5$ to 0.6), where data access is relatively uniform, plain treap and SkipList + Hash show competitive performance.

As the workload becomes more highly skewed ($\alpha \geq 0.7$), the proposed SkipList + Treap (TASL) begins to significantly outperform all other structures. For $\alpha \geq 0.8$, all other hybrid structures suffer from performance inversion, where the cache maintenance overhead outweighs the retrieval benefits. TASL is the only hybrid structure that converts data skewness into performance gains as shown in Figure 3.

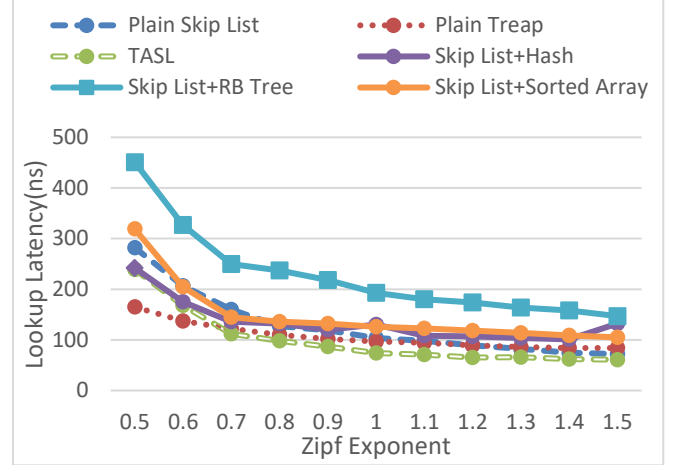


Figure 3: Hot-Key Point Lookup Latency across Zipf Exponent.

The hot p99 latency further distinguishes TASL from its competitors. While other hybrid structures encounter a 'latency floor'—failing to drop below 200–300ns even under extreme skewness due to fixed algorithmic overheads—TASL exhibits a continuous downward trend.

As α increases from 0.7 to 1.5 , TASL's hot p99 latency steadily decreases, even much better than plain skip list and plain treap. This suggests that TASL effectively exploits highly skewed access patterns: repeated queries increasingly target a small set of resident keys, potentially benefiting from repeated search paths and cache reuse.

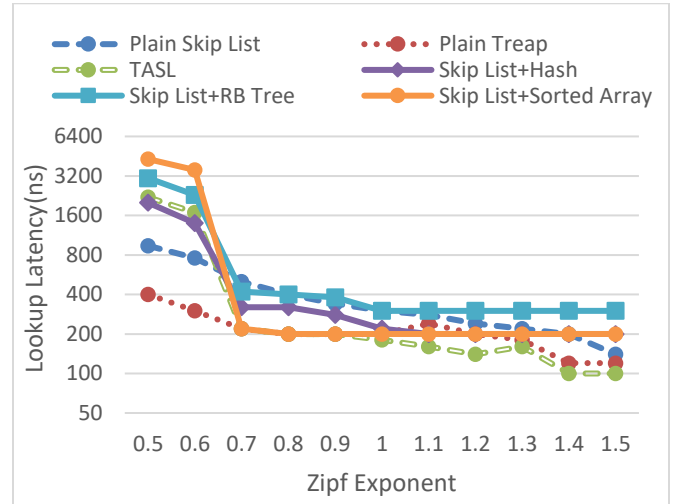


Figure 4: p99 Latency for Hot Keys across Zipf Exponent.

5.2.1 Parameters

The number of hot keys: HOT_KEYS controls how many of the most frequently accessed keys are tracked for hot-key latency measurement. In our setup, HOT_KEYS = 250, meaning we measure separate latency for the top 250 keys by query frequency.

A larger HOT_KEYS value widens the measured hot set, capturing more of the Zipfian head. However, as HOT_KEYS grows, the average frequency of included keys decreases, reducing the treap hit rate for that subset and increasing measured hot-key latency. The default value of 250 was chosen to represent the tightly concentrated head of the Zipfian distribution at $\alpha \geq 0.7$.

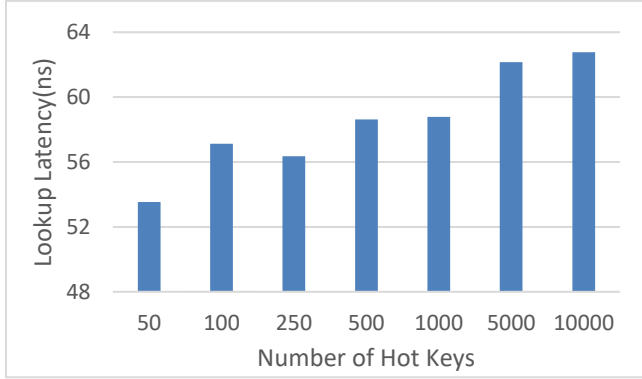


Figure 5: Point Lookup Average Latency across Number of Hot Keys.

TREAP_ADMIT_THRESH: TREAP_ADMIT_THRESH (θ) is the minimum access count a key must accumulate in the skip list before being promoted to the treap. A low threshold admits keys aggressively, filling the treap quickly but with keys that may not be consistently hot, increasing eviction pressure and noise. A high threshold ensures only genuinely frequent keys are admitted but delays promotion, causing hot keys to incur skip list traversal cost longer than necessary. We use $\theta=3$ as the default, which balances responsiveness to emerging hot keys against premature admission of transient spikes.

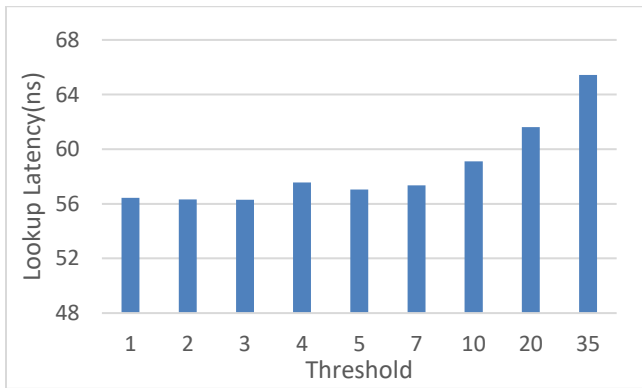


Figure 6: Point Lookup Average Latency across the Threshold.

TREAP_MAX_SIZE: TREAP_MAX_SIZE (K) is the maximum number of keys the treap hot-tier can hold. A larger K increases the treap hit rate by accommodating a broader hot set, but

raises per-lookup cost as the treap grows deeper ($O(\log K)$). A smaller K keeps lookups fast but evicts keys prematurely, reducing hit rate. We set $K = 30,000$ (3% of $N = 1,000,000$), which is large enough to cover the Zipfian head at all tested α values while keeping treap depth well below that of the full skip list.

When Zipf exponent increases, the K needed for convergence decreases. This indicates that in a more skewed dataset, we can converge with less K . This suggests that this structure can be more effective in skewed datasets.

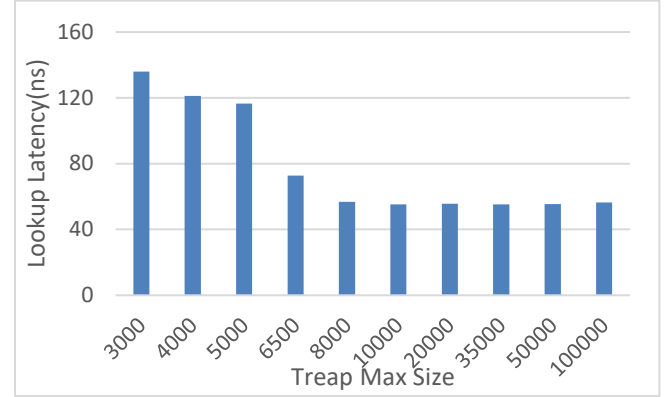


Figure 7: Point Lookup Average Latency across Maximum Size of Treap.

5.3 Eviction Policy Comparison

Figure 8 compares average point lookup latency across the three eviction policies over $\alpha \in [0.5, 1.5]$. Random eviction consistently achieves the lowest latency across all tested α values, ranging from 777 ns at $\alpha = 0.5$ down to 58.89 ns at $\alpha = 1.4$. LRU eviction is substantially slower than random at every α , ranging from 1270.66 ns at $\alpha = 0.5$ to 95.78 ns at $\alpha = 1.5$ —roughly 1.6 $\times$ to 2.0 $\times$ slower than random across the mid-range. LFU eviction performs similarly to LRU but is consistently slower still, ranging from 1263.89 ns at $\alpha = 0.5$ to 138.78 ns at $\alpha = 1.5$. The gap between random and both LRU/LFU is largest at low α (≤ 0.7), where the hot set is diverse and eviction events are frequent, amplifying the overhead of the auxiliary bookkeeping structures. At high α (≥ 1.3) the gap narrows as most queries hit a small number of keys and eviction becomes rare, yet random eviction remains the fastest throughout. Despite LFU being theoretically optimal for hit rate, its per-access frequency update overhead raises the effective treap lookup cost T , increasing the required hit-rate threshold from our analysis. LRU shares the same structural overhead as LFU and similarly underperforms random in practice. Random eviction therefore provides the best practical trade-off: $O(\log k)$ eviction cost, no per-access bookkeeping, and the lowest observed latency across all workload skews.

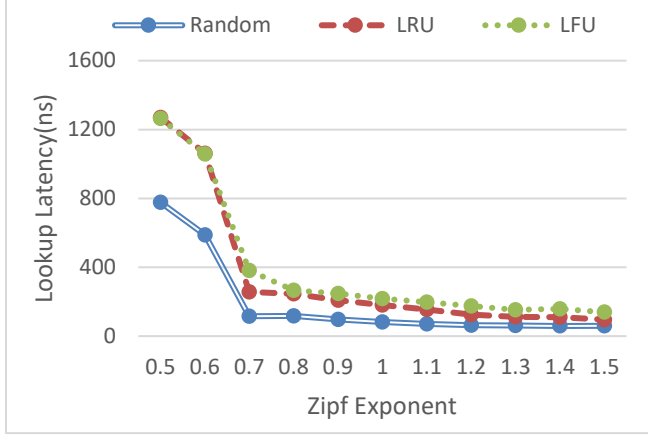


Figure 8: Average Latency by Eviction Policy across Zipf Exponent.

5.4 Insert/Update/Delete

We additionally measure insert, delete, and update latency under a randomized workload over $N = 1,000,000$ keys. Existing and unused keys are selected uniformly at random, and an update consists of a random deletion followed by an insertion.

Implementation	Insert (ns)	Delete (ns)	Update (ns)
Plain Skip List	4195.59	3953.96	8293.49
SkipList + Treap	4436.94	4789.20	8602.63

TASL preserves near-baseline insert and update performance, with overheads of 5.8% and 3.7%, respectively. Delete latency increases by 21.1%, reflecting the additional hot-tier and residency bookkeeping required during deletion. Thus, TASL’s write overhead is modest for insertion and update but more pronounced for deletion.

5.5 Discussion

The results reveal distinct behavioral trends for each hybrid structure as workload skewness increases. SkipList + Hash and SkipList + SortedArray remain competitive at low α , but their performance plateaus or declines relative to TASL as skewness increases. SkipList + RBTree also remains slower than TASL across the evaluated high-skew workloads. These results indicate that asymptotic lookup complexity alone does not determine hot-tier performance; practical performance also depends on the combined costs of lookup, admission, eviction, and metadata maintenance.

TASL distinguishes itself by maintaining a robust, positive correlation between skewness and efficiency. Unlike the alternative hot-tier implementations, the treap-based tier shows lower observed lookup overhead as accesses become concentrated on a small resident key set, allowing the hot-key mean latency to improve where other structures fail. This structural advantage is most evident in the hot p99 trends; while competing hybrids hit a

fixed latency floor due to their management overhead, TASL is the only architecture that breaks this barrier, showing a continuous downward trend in tail latency as workload skewness increases. These trends confirm that TASL’s probabilistic approach effectively converts increasing workload skew into consistent lookup performance gains.

6. Limitations and Future Work

While TASL demonstrates significant performance gains in our experiments, we identify two primary limitations: first, the current implementation is designed for single-threaded execution, lacking the lock-free treap operations or fine-grained concurrency control mechanisms required for production environments like Redis or RocksDB to ensure data consistency under high contention; and second, our evaluation relies on synthetic Zipfian workloads which, despite testing a wide range of skewness, may not fully capture the complex, time-varying access patterns found in real-world production key-value stores, necessitating further validation using industry-scale traces to confirm its adaptability.

Future work includes concurrent TASL with lock-free treap operations, adaptive admission thresholds that respond to workload shifts, and evaluation on real-world traces from production key-value stores. An additional direction is leveraging the treap hot-tier to accelerate short range queries. Since the treap maintains the hot key set in a BST structure, it may be possible to exploit this for short-range queries that fall entirely within the hot set, potentially reducing reliance on the skip list for such cases. However, the precise conditions under which this yields a net benefit—and how to integrate it without disrupting the skip list’s range-query semantics—remain open questions for future investigation.

7. Conclusion

We presented TASL, a two-tier data structure that augments an unmodified skip list with an external treap hot-tier to optimize performance under skewed workloads. By exploiting the complementary strengths of both structures—the skip list’s efficiency in range queries and the treap’s fast point lookups for hot sets—TASL achieves superior latency compared to both standalone structures and other hybrid baselines. Our results demonstrate that TASL effectively converts high data skewness into consistent performance gains while maintaining the original range-query behavior. Furthermore, we showed that simple random eviction provides the best practical balance between low management overhead and high lookup efficiency. Ultimately, TASL offers a robust and adaptable architecture for modern key-value stores facing highly concentrated access patterns.

REFERENCES

- [1] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.
- [2] George K. Zipf. 1949. *Human Behavior and the Principle of Least Effort*. Addison-Wesley.
- [3] Vitaly Aksenov, Dan Alistarh, Alexandra Drozdova, and Amirkeivan Mohtashami. 2020. The Splay-List: A Distribution-Adaptive Concurrent Skip-List. In *34th International Symposium on Distributed Computing (DISC 2020)*.
- [4] Daniel D. Sleator and Robert E. Tarjan. 1985. Self-adjusting binary search trees. *J. ACM* 32, 3 (1985), 652–686.
- [5] Raimund Seidel and Cecilia R. Aragon. 1996. Randomized search trees. *Algorithmica* 16, 4–5 (1996), 464–497.
- [6] Josh MacDonald and Ben Zhao. 1999. T-treap and Cache Performance of Indexing Data Structures. Technical Report, UC Berkeley.
- [7] Ronald L. Rivest. 1976. On self-organizing sequential search heuristics. *Commun. ACM* 19, 2 (1976), 63–67.
- [8] Amitabha Bagchi, Adam L. Buchsbaum, and Michael T. Goodrich. 2005. Biased skip lists. *Algorithmica* 42, 1 (2005), 31–48.