The Skip2List: A Multi-Layer Skip List for Non-Uniform Data Distributions

Mugilan Ganesan

***Abstract*—There has recently been an increased interest in developing distribution adaptive data structures, which use the frequency distribution of the queried data as a measure by which to optimize their operations.**

In this paper, we present a design for an adaptive data structure called the skip2list. The skip2list uses a novel algorithm to determine optimally positioned guard entries, which are used to improve the worst-case time complexity of a skip list search from $\mathcal{O}(n)$ to $\mathcal{O}(\sqrt{n})$. Existing skip lists can be augmented without altering their underlying implementations.

We first describe the structure of the skip2list, before explaining our guard entry selection algorithm. We then provide experimental results testing the skip2list and the guard entry selection algorithm on various data distributions.

I. Introduction

The past few decades have seen considerable attention directed towards building specialized data structures with optimal time and space complexities. Various new data structures and modifications of older ones have been proposed that aim to provide generic worst-case complexity guarantees for uniformly distributed data. In practice, however, data cannot be expected to be distributed uniformly as the distribution can depend on a number of external factors.

Thus, there has been a growing interest in developing distribution adaptive data structures, which make use of the frequency distribution of the queried data. The method by which they optimize their operations varies from structure to structure. Splay-trees [1], for example, rearrange their nodes to prioritize recently-accessed data. Another example are learned index structures [2]: a new class of data structures that learn the sort order or structure of lookup keys to effectively predict the position or existence of data.

The skip list [3] is a type of probabilistic data structure composed of multiple levels. The bottom-most level is a sorted linked list and the other levels act as "express lanes" for the levels lower than them. Each node in the skip list is assigned a random height, which determines the level that it can first be found at. Through this approach, the skip list achieves an average search time complexity of $\mathcal{O}(\log n)$.

However, the original skip list does not make assumptions about the distribution of its data. As a result, its procedures, node arrangements, and structure can be sub-optimal for distributions that are skewed and show bias. Many approaches [4] [5] [6] [7] have since been proposed, which statically and dynamically adjust the skip list, given knowledge of the data distribution. One standard approach to correcting sub-optimal node arrangements is to ensure that nodes which are more likely to be accessed have a greater height.

An alternative approach is to improve the performance of a regular skip list with an auxiliary data structure. S3 [8] is one such recently proposed data structure. It adopts a two-layer structure. The top layer uses a cache-sensitive structure to search over select guard entries. The bottom layer consists of a semi-ordered skip list to support concurrent insertions and fast lookups.

S3 is of particular interest to us for a few reasons. Despite showing that searching over guard entries can significantly improve search times, S3 is restricted to only selecting guard entries within equal size partitions of the skip list. This approach can be inflexible and limits the data structure's ability to adapt to the data distribution. Furthermore, S3 uses an LSTM neural network to select its guard entries. While neural networks can model distributions well, training a model can be computationally expensive and a model's predictions can be inconsistent.

In this paper, we aim to address these issues by outlining a specification for a new data structure known as the skip2list. An existing skip list can be augmented into a skip2list without modifying its underlying implementation or data. In return, the skip2list improves the worst-case time complexity of a regular skip list from $\mathcal{O}(n)$ to $\mathcal{O}(\sqrt{n})$.

Our goal is to also provide a guard entry selection algorithm for the skip2list which provides strict time and space complexity guarantees, while being efficient, consistent, and conceptually simple to implement. To this end, we propose an optimized local-search algorithm for selecting guard entries. The algorithm can select nodes from any given position within the skip list, which grants it greater flexibility and adaptability.

II. The Skip2List

In this section, we will describe in detail the structure of the skip2list. We will then cover the basic operations of the skip2list and perform a time complexity analysis.

A. The Structure

The skip2list consists of two distinct layers to facilitate searches. The first layer is an auxiliary index for the guard entries, while the second layer is an ordered skip list.

The keys stored in the first layer are the original keys of the guard entries and their corresponding values are the pointers to the guard entries in the skip list. A number of different data structures can be used for the first layer, but they should support the predecessor operation with a worst-case time complexity of preferably $\mathcal{O}(\log n)$ as this is the average search time complexity of a regular skip list. We have

opted for a simple binary search tree as the first layer and will later base our complexity analysis on it.

The second layer consists of a regular singly-linked skip list with ordered nodes. The guard entry approach can be extended to skip lists based on doubly-linked lists, but our intention was to maintain a minimal space complexity. A key advantage of the skip2list is that it treats its second layer as a black box. Any existing skip list can be modified into a skip2list without having its internal implementation changed. The first layer is merely an add-on, but one that significantly improves search times.

When a skip2list is first created, the guard entry selection algorithm is executed based on previously collected data about the data distribution. However, data will be updated, inserted, and deleted over time and the data distribution will change. We assume that the data distribution changes smoothly, which is generally true in practice. Therefore, the selection algorithm can be rerun after a set number of operations or a set time interval has passed. For small changes in the distribution, the algorithm will only alter the positions of the existing guard entries, which is computationally-efficient and allows the skip2list to adapt to changes continuously. For larger changes, the guard entries can be generated from scratch and the algorithm can be rerun from the start.

A naive assumption to improve the performance of the skip2list would be to increase the number of guard entries. There is, however, a duality between the first and second layers that must be considered. While increasing the number of guard entries would reduce the search time in the second layer, it increases the memory and time required to search the first layer, rendering it inefficient. Similarly, a scarcity of guard entries would result in an overburdened second layer nearly equivalent to a regular skip list. In this paper, we have considered the number of guard entries $m = \sqrt{n}$. This minimizes the worst-case time complexity of a skip2list search with evenly spaced guard entries as we will see in the complexity analysis.

B. Basic Operations

When a key is searched for in the skip2list, the guard entry with the largest key less than or equal to the requested key is looked up from the first layer. If the key belongs to a guard entry, then the guard entry itself can be returned directly. Once the closest guard entry has been identified, a regular skip list search for the requested key can be performed from it. This is because skip lists are implemented with linked lists, which do not significantly differentiate between head nodes and regular nodes. Thus, the guard entry can serve as the head of a truncated skip list, which significantly narrows down the possible positions of the requested node.

The insertion operation is similar to that of a regular ordered skip list. The initial search operation to find the expected position of the key to be inserted is replaced with that of the skip2list search. The creation of the node and the linking process proceeds as per usual.

Deletion is a slightly more complex operation. If the key to be deleted is not a guard entry, then the operation is performed similarly to a regular skip list. If a guard entry is to be deleted, however, then a new guard entry must be selected to take its place. The deleted node must be replaced in the first layer by the new node. Selection of the new guard entry can vary per implementation. One method is to set the new guard entry to be the node directly before the old one. Another method is to randomly select a node from the second layer.

C. Complexity Analysis

We will now compare the search time complexity of a skip2list to a regular skip list. The insertion and deletion operations depend primarily on the complexity of searching. A generic search for a key in a non-deterministic skip list of n elements has an average time complexity of $\mathcal{O}(\log n)$ and a worst-case complexity of $\mathcal{O}(n)$.

Suppose that a skip2list has been constructed with m guard entries that are evenly spaced apart. In a BST, the worst-case time complexity of the predecessor operation is $\mathcal{O}(n)$. Thus, the initial search for the guard entries in the first layer is $\mathcal{O}(m)$. As the guard entries are evenly spaced, there are $\frac{n}{m}$ nodes between the selected guard entry and the following one. The requested node must lie within this range. Hence, the worst-case time for a skip list search in the second layer is $\mathcal{O}(\frac{n}{m})$.

The overall worst-case time complexity for a skip2list search is $\mathcal{O}(m + \frac{n}{m})$. Minimizing this expression reveals that $m = \sqrt{n}$ results in an optimal time complexity bound of $\mathcal{O}(\sqrt{n})$. It is critical to note that this is a *worst-case guarantee* for any percentile, whereas the original skip list search is worst-case $\mathcal{O}(n)$. A skip2list search is on average $\mathcal{O}(\log m)$ for the first layer and $\mathcal{O}(\log \frac{n}{m})$ for the second layer. This results in an average time complexity of $\mathcal{O}(\log n)$. Furthermore, this analysis does not consider non-uniform data distributions. When the guard entry selection algorithm is executed to prioritize frequently requested keys, the average search time complexity can improve to $\Omega(\log m)$ as the most requested keys will be as close to the guard entries as possible to mitigate the slower linear search of the second layer.

III. The Selection Algorithm

We will now provide a mathematical derivation of a cost function to optimize. Afterwards, we will explain the process and justification of our local search algorithm to select the guard entries. As the basic selection algorithm is relatively inefficient, the subsequent section will provide an optimized version that can be practically implemented.

A. Guard Entry Routing Cost

Consider a skip2list, where each node is indexed from 0 to n and the index of a node is equivalent to its position along the skip list. Assume that the skip2list has m guard entries. The position/index of the ith guard entry within the skip2list is denoted as x_i. We will also refer to the ith guard entry itself as x_i for convenience; x_0 and x_{m+1} are equal to 0 and n respectively as they are the head and tail node of the skip2list.

We can define the cost of reaching the jth node from the ith guard entry as:

$$C_{ij} = Q_j \, (j - x_i) \tag{1}$$

Q_j denotes the total frequency of the queries for the jth node. The cost depends on 1) the number of queries for the jth node and 2) the distance between the guard entry and the jth node. Thus, the total cost C of reaching all nodes from their respective guard entries in the skip2list is given by:

$$C = \sum_{i=0}^{m} \sum_{j=x_i}^{x_{i+1}-1} C_{ij} \tag{2}$$

After simplifying the above expression, it is found that minimizing C is equal to maximizing

$$V = \sum_{i=0}^{m} V_i \tag{3}$$

where

$$V_i = x_i \sum_{j=x_i}^{x_{i+1}-1} Q_j \tag{4}$$

This is a discrete equation in m variables (the position of each guard entry x_i), so it is inefficient to optimize analytically for large values of m.

B. Determining the Optimal Guard Entry Shift

The position of a given guard entry x_i has a single degree of freedom which corresponds to shifting the position of the guard entry to the left or to the right. As our goal is to maximize V, we would like to shift x_i in the direction that results in a greater increase of V (if at all). This change can be quantified as ΔV. If V^* denotes the new value of V after a change:

$$\Delta V = V^* - V \tag{5}$$

While shifting the position of x_i may appear to have an effect on all the guard entries, it can be recognized that shifting the position of x_i only affects the value of V_{i-1} and V_i. This justifies the use of a local search algorithm as the effects of shifts are locally contained. It also greatly simplifies the computation of ΔV as only V_{i-1} and V_i need to be evaluated. Let us first consider ΔV_{left}: the change resulting from setting $x_i \leftarrow x_i - 1$.

$$\Delta V_{left} = \Delta V_i + \Delta V_{i-1} = (V_i^* + V_{i-1}^*) - (V_i + V_{i-1}) \tag{6}$$

We can explicitly write $V_i^* + V_{i-1}^*$ and $V_i + V_{i-1}$ as

$$V_i^* + V_{i-1}^* = (x_i - 1) \sum_{j=x_i-1}^{x_{i+1}-1} Q_j + (x_{i-1}) \sum_{j=x_{i-1}}^{x_i-2} Q_j \tag{7}$$

$$V_i + V_{i-1} = x_i \sum_{j=x_i}^{x_{i+1}-1} Q_j + (x_{i-1}) \sum_{j=x_{i-1}}^{x_i-1} Q_j \tag{8}$$

After substituting equations 7 and 8 into equation 6, we find that

$$\Delta V_{left} = (x_i - x_{i-1} - 1) Q_{x_i - 1} - \sum_{j=x_i}^{x_{i+1}-1} Q_j \tag{9}$$

Now that we have obtained an equation for computing ΔV_{left}, we can find an equation for ΔV_{right}. We will first define ΔV_{right} in a similar manner as we did for ΔV_{left}.

$$\Delta V_{right} = \Delta V_i + \Delta V_{i-1} = (V_i^* + V_{i-1}^*) - (V_i + V_{i-1}) \tag{10}$$

A rightward shift corresponds to $x_i \leftarrow x_i + 1$. We can now write out $V_i^* + V_{i-1}^*$ in this case as

$$V_i^* + V_{i-1}^* = (x_i + 1) \sum_{j=x_i+1}^{x_{i+1}-1} Q_j + (x_{i-1}) \sum_{j=x_{i-1}}^{x_i} Q_j \tag{11}$$

As $V_i + V_{i-1}$ is the same as equation 8, we can evaluate ΔV_{right} to be equal to

$$\Delta V_{right} = (x_{i-1} - x_i - 1) Q_{x_i} + \sum_{j=x_i}^{x_{i+1}-1} Q_j \tag{12}$$

C. The Selection Algorithm

Putting equations 12 and 9 together allows us to build a straightforward procedure for computing the optimal shift of x_i. This is outlined in algorithm 1. Note: $A[i]$ denotes the ith index of an array A.

Algorithm 1: The Guard Entry Selection Algorithm

```
x is set to the initial guard entry positions
for i ← 1 to k do
    for j ← 1 to m do
        Compute ΔV_left and ΔV_right
        if ΔV_left > ΔV_right then
            x[j] ← x[j] − 1
        else
            x[j] ← x[j] + 1
        end
    end
end
```

The inner loop repeats the shift procedure for each guard entry. However, x_0 and x_{m+1} are ignored and kept stationary, since they are defined to be the head and tail node of the skip list respectively. As an iteration of the inner loop only moves a guard entry by 1 position, an outer loop is required. The number of outer loop iterations is set as k. This is merely a constant that can be increased to achieve an arbitrary level of fine-tuning and optimization.

However, we have not yet mentioned the initial states of the guard entries before the algorithm starts. The initial state of a local optimization algorithm can significantly influence its final result. Zhang et al. have shown mathematically that evenly spaced guard entries are optimal for a uniform distribution of data [8]. While data isn't uniformly distributed

in practice, they have found this assumption to be effective regardless. Hence, the initial positions of the guard entries in our algorithm are evenly spaced as well.

A caveat of a local optimization algorithm is that it may fail to reach the global optimum and halt at an inferior local optimum. However, global optimization approaches to this problem, such as through simulated annealing, are more computationally expensive, while offering limited gains in optimization. We have found that micro-optimization of the guard entry positions is not worth the computational power and time in practice and our local search algorithm provides close approximate solutions to that of a global optimization approach.

D. Implementation

While the previous section describes the general working principle of the algorithm, it is relatively inefficient. The inner-loop requires continuous recomputation of the computationally expensive summation Q_{sum} for every computation of ΔV_{left} and ΔV_{right}.

$$Q_{sum} = \sum_{j=x_i}^{x_{i+1}-1} Q_j \tag{13}$$

This can be avoided by caching the values of Q_{sum} in an array and therefore reducing the computation of Q_{sum} to a constant time operation. We will designate this array as T. The partial summations that constitute Q_{sum} are computed beforehand and stored in T. This optimization and a complete implementation has been provided as algorithm 2.

Algorithm 2: An Optimized Implementation

```
x is set to the initial guard entry positions
T = φ
for i ← 1 to m do
    T[i] ← sum of Q_k for k ← x_{k-1} to x_k
end
for i ← 1 to k do
    for j ← 1 to m do
        Q_sum ← T[j]
        ΔV_left ← (x[j]−1−x[j−1])Q[x[j]−1]−Q_sum
        ΔV_right ← (x[j−1]−1−x[j])Q[x[j]]+Q_sum
        if ΔV_left > ΔV_right then
            T[j−1] ← T[j−1] − Q[x[j]−1]
            T[j] ← T[j] + Q[x[j]−1]
            x[j] ← x[j] − 1
        else
            T[j−1] ← T[j−1] + Q[x[j]]
            T[j] ← T[j] − Q[x[j]]
            x[j] ← x[j] + 1
        end
    end
end
```

This is the final algorithm that was implemented for experimental testing. It has a time complexity of $\mathcal{O}(mk)$. If we set $k = n$ and $m = \sqrt{n}$, we find that it has a time complexity of $\mathcal{O}\left(n^{3/2}\right)$.

IV. Experimental Results

We will now present the results of our experimental tests on the skip2list and its guard entry selection algorithm.

A. Environment and Methodology

These tests were run on a quad-core Intel(R) Core(TM) i7-5557U CPU @ 3.10GHz with 16GB of RAM. The skip2list and base skip list were both implemented in C and compiled with Clang 11 [9]. The experiment was repeated 200 times for each comparison. The values obtained are the average query throughput in Millions of Queries per Second. The heights of skip list nodes were randomly determined at run-time with $p = 0.5$ to obtain an average expected performance comparison between different algorithms independent of the actual run-time heights.

B. Parameters

The number of skip list nodes n was taken to be 1×10^5. The number of guard entries m was taken as $\sqrt{n}$ or approximately 316, since this minimizes the theoretical worst-case time complexity of a lookup as mentioned earlier.

The overall number of queries was set to 2×10^5. The number of queries per node was determined by the probability density function of each distribution, with the indices of the nodes corresponding to the output of a random variable.

The data distributions considered are a uniform distribution and a normal distribution of $\sigma = 2 \times 10^4$.

C. The Skip2list vs The Skip list

The first test was a comparison of the lookup query throughput of a skip2list with that of a regular skip list. After a skip list was generated and tested, it was directly converted into a skip2list. As a skip2list preserves the structure of its constituent skip list. this ensures that the comparison between the skip list and skip2list would be independent of the role played by the randomly determined heights of the nodes. The results are shown in Fig. 1.

For a uniform distribution, the query throughput of a skip2list is greater by a factor of 6.79. Similar results were also obtained for a normal distribution. These results show that the skip2list generally has faster lookup speeds compared to a normal skip list for simple distributions, despite only requiring an extra space complexity of $\mathcal{O}(m)$.

This search time improvement can be attributed to the work performed by the first layer. A regular skip list is biases towards nodes closer to the head node as it performs a sequential search. In contrast, the binary search used to lookup guard entries in the first layer of the skip2list has an expected time complexity of $\mathcal{O}(\log m)$ regardless of the data percentile.

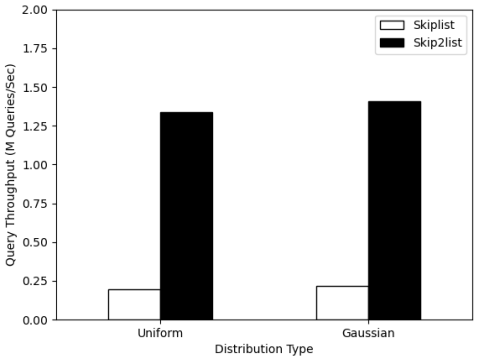

Fig. 1. Comparison of query throughput between a skip2list and a skip list for different data distributions

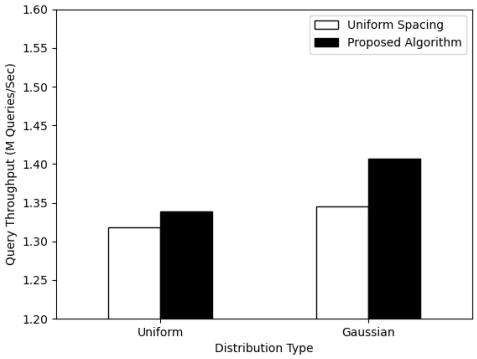

Fig. 2. Comparison of query throughput between different guard entry selection mechanisms

D. The Guard Entry Selection Mechanism

We will now show how the query throughput of a skip2list differs depending on the use of different guard entry selection mechanisms. The method of selecting evenly spaced guard entries is compared to our algorithm. The results are shown in Fig. 2.

The query throughput of our algorithm is similar to that of the evenly spaced guard entries in the case of a uniform distribution. This is to be expected as evenly spaced positioning results in the optimal placement of the guard entries for a uniform distribution [8].

For a normal distribution, our algorithm performs better as it is able to adapt to the data distribution. The difference in query throughput can be increased further, however, by taking into account the heights of the nodes in the skip2list. The cost of traversing down the levels of a skip list is non-negligible in practice, which means that the placements of the guard entries can be further optimized.

V. Future Application

One possible application of this data structure is in storing time series data. As the dates must be stored in order and the frequency distribution of the queries can be concentrated around certain dates, this provides a suitable situation to use a skip2list. A skip2list can greatly improve lookup speeds by adapting to certain date ranges, while supporting efficient extensibility of the data set for newer data points.

VI. Conclusion

By adapting to the frequency distribution of the data stored within them, conventional data structures can optimize their operations to competitive levels. The skip2list is a distribution-adaptive skip list that existing skip lists can easily be modified into. It guarantees faster and more consistent lookup operations of worst-case time complexity $\mathcal{O}(\sqrt{n})$ without compromising on memory requirements.

The guard entry selection mechanism proposed is optimized and effective. It outperforms other methods, such as an evenly spaced selection of guard entries. Compared to the LSTM used by S3, it also provides stricter time and space complexity guarantees and is simpler to implement. Furthermore, it avoids the computationally expensive model retraining required when data is altered by continuously modifying the guard entry positions in increments.

VII. Acknowledgements

This research was carried out at the Centre for Fundamental Research and Creative Education (CFRCE), Bangalore, India. I would like to sincerely thank Dr. B S Ramachandra for his guidance on this paper. I would also like to thank Mr. Sameer Kolar and Mr. Rahul K Balaji for their advice.

References

[1] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[2] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 489–504, New York, NY, USA, 2018. Association for Computing Machinery.

[3] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[4] V. Aksenov, Dan Alistarh, Alexandra Drozdova, and Amirkeivan Mohtashami. The splay-list: A distribution-adaptive concurrent skip-list. In *DISC*, 2020.

[5] Funda Ergün, Süleyman Cenk Sahinalp, Jonathan Sharp, and Rakesh K. Sinha. Biased skip lists for highly skewed access patterns. In *Revised Papers from the Third International Workshop on Algorithm Engineering and Experimentation*, ALENEX '01, page 216–230, Berlin, Heidelberg, 2001. Springer-Verlag.

[6] Prosenjit Bose, Karim Douïeb, and Stefan Langerman. Dynamic optimality for skip lists and b-trees. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '08, page 1106–1114, USA, 2008. Society for Industrial and Applied Mathematics.

[7] Jonathan J. Pittard and Alan L. Tharp. Simplified self-adapting skip lists. In *Proceedings of the 11th International Conference on Intelligent Data Engineering and Automated Learning*, IDEAL'10, page 126–136, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] Jingtian Zhang, Sai Wu, Zeyuan Tan, Gang Chen, Zhushi Cheng, Wei Cao, Yusong Gao, and Xiaojie Feng. S3: A scalable in-memory skip-list index for key-value store. *Proc. VLDB Endow.*, 12(12):2183–2194, 2019.

[9] Clang. https://clang.llvm.org/.