A Simple Cache-Sensitive Skip List (SCSSL)

Yuchang Ke
independent researcher
Wuhan, China
aceking.ke@gmail.com

Bill Lv
independent researcher
Wuhan, China
ideaalloc@gmail.com

Weijie Zhang
Hubei Changjiang Yunxin Media Group Co. Ltd
Wuhan, China
zhangweijie@gmail.com

ABSTRACT

We present a cache-sensitive skip list (SCSSL) that is remarkably simple and easy to implement. It requires only minor modifications to the original skip list data structure to adapt it effectively to modern CPU architectures. This adaptation results in a performance improvement typically ranging from 1.2x to 1.6x, and on some systems, can reach 2x to 3x for insertion-heavy workloads. Unlike other optimization techniques, SCSSL does not rely on specialized hardware instructions like SIMD or intricate multi-threaded programming paradigms. Instead, its design inherently benefits from improved data locality due to the use of arrays at the data level, a feature that modern compilers can further leverage. This makes the proposed design broadly applicable, suitable not only for in-memory databases and key-value stores but also for resource-constrained embedded systems. and the github is https://github.com/acekingke/simpleCacheSentiveSkiplist

PVLDB Reference Format:
Yuchang Ke, Bill Lv, and Weijie Zhang. A Simple Cache-Sensitive Skip List (SCSSL). PVLDB, 19(1): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

1 INTRODUCTION

In computer science, ordered data structures are fundamental for organizing and managing data efficiently. The performance of search, insertion, and deletion operations within these structures is crucial for a wide array of applications, as it directly impacts overall system performance. Numerous methods have been developed to achieve high performance in ordered data structures, such as AVL trees and Red-Black trees. However, these balanced tree structures are relatively complex to implement due to the necessity of rebalancing operations. This landscape changed when William Pugh introduced the skip list in 1990 [1]. The skip list, a probabilistic data structure, offered a simpler implementation alternative to traditional balanced trees while providing comparable performance characteristics.

Thanks to its simplicity and efficiency, the skip list has been widely adopted in various systems. Notable examples include databases like LevelDB [2], Redis [7], RocksDB [3] and HBase [4], as well as in-memory caches and the inverted index component of search engines like Lucene.

A standard skip list consists of multiple levels: a data level (level 0) that stores all data nodes in a sorted linked list, and multiple index levels above it. These index levels contain a subset of the nodes from lower levels and act as **express lanes** to expedite traversal to the desired data nodes. However, nodes in a skip list are typically allocated dynamically, leading to non-contiguous memory addresses. This fragmentation can degrade performance on modern CPUs due to frequent cache misses.

Although ESL, CSSL has developed cache sensitive skip lists to fit modern CPU architecture, these algorithms are still complex, and CSSL uses SIMD instructions which are special in Intel. We strongly believe that the value of a skiplist lies in its simplicity. Excessively complex cache-sensitive designs reduce the appeal of skip list variations.

Although cache-sensitive skip list variants like the Cache-Sensitive Skip List (CSSL) [5] and ESL (Express Lane Skip List) [6] have been developed to better align with modern CPU architectures, they often introduce significant complexity. For instance, CSSL utilizes SIMD instructions, which may not be universally available or optimal across all platforms. ESL proposes merging index level nodes into arrays and employs sophisticated version-based locking protocols and lock-free operation logs for concurrency. We believe that the primary appeal of the skip list lies in its inherent simplicity. Overly complex cache-sensitive designs can diminish this advantage.

Our work is motivated by the observation that simple architectural changes can yield substantial performance gains by improving cache locality. We propose a modification that retains the core simplicity of the skip list while significantly enhancing its cache-friendliness.

2 DESIGN OVERVIEW

In this paper, we present a simple and easy-to-implement cache-sensitive skip list. At the data level, it introduces a vector, and the nodes at this level now store arrays instead of single values. Leveraging CPU cache benefits for small arrays to enhance skip list performance. For example, consider a skip list with a vector at the data level; let's suppose the vector's maximum size is 2, as shown in Figure 1.

In this paper, we present a Simple Cache-Sensitive Skip List (SCSSL). The core idea is to modify the nodes at the data level (level 0) to store a small, sorted array of elements

Proceedings of the VLDB Endowment, Vol. 19, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

(e.g., key-value pairs or just keys) instead of a single element. This approach leverages CPU cache locality for these small arrays, thereby enhancing performance. For example, Figure 1 illustrates a conceptual SCSSL where data nodes hold arrays of maximum size 2.

The design of SCSSL hinges on three key principles:

(1) **Array-based Data Nodes:** At the data level (level 0), each node contains a sorted array with a fixed maximum length. For integer keys, a maximum length that allows the node to fit well within CPU cache lines (e.g., up to 128 elements, depending on key/value sizes) is chosen.
(2) **Index Level Pivot:** For navigation through the index levels (levels 1 and above), only the first element (i.e., the smallest key) of the array in a data-level node (or an index node pointing to such an array) is used as the pivot for comparison.
(3) **Insertion Handling:** The last element (i.e., the largest key) of an array in a data node is primarily considered during insertion operations, particularly when an array is full and a new key needs to be accommodated.

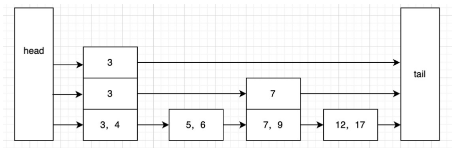

Figure 1: A conceptual view of the Simple Cache-Sensitive Skip List (SCSSL) where data nodes (level 0) store arrays of elements (e.g., max array size = 2 as depicted). Index nodes point to the start of these arrays.

Our key contributions include:

(1) Introducing arrays within data-level nodes, which maintains the asymptotic complexity of skip list operations while significantly improving CPU cache hit rates due to the contiguous storage of multiple elements.
(2) Reducing the effective height of the skip list and pointer overhead, as each data-level node now stores multiple elements, leading to fewer nodes overall for a given dataset size.
(3) Presenting modified insertion, search, and deletion algorithms tailored for these array-based data nodes, offering a practical and simple approach for enhancing skip list performance, particularly in database indexing and similar applications.

3 ALGORITHM

This section details the algorithms for search, insertion, and deletion in SCSSL. These operations are adapted from standard skip list algorithms to accommodate the array-based nodes at the data level.

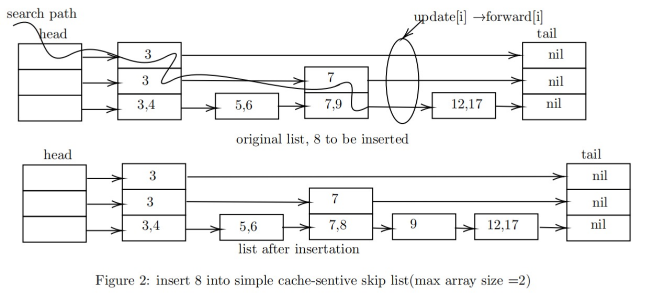

Figure 2: Insertion of key 8 into an SCSSL (max array size = 2). Key 8 displaces 9 from node (7,9). Key 9 is then inserted into a new node.

In SCSSL, each node at the data level (level 0) stores its elements in a sorted array of a fixed maximum size. When a new node is created, its level is chosen randomly, similar to a standard skip list. A node promoted to level k has $k + 1$ forward pointers (indexed 0 to k). The maximum number of levels in the skip list is a predefined constant, `MAX_LEVEL`. The skip list employs a header node to initiate searches; its forward pointers are initially Nil. For simplicity in comparisons, the header node can be conceptually considered to hold a key value of negative infinity or have a special marker. Figure 2 illustrates an example of an insertion operation in SCSSL.

3.1 Initialization

The initialization process is elementary. A header node is created with an array of `MAX_LEVEL+1` forward pointers, all initialized to Nil. To simplify comparisons, the header node's key array (if it were to have one in the same structure) can be considered to contain a single sentinel value of negative infinity, or it can be handled as a special case. In our algorithms, `header.FirstKey()` would effectively be ∞.

3.2 Search

Algorithm 1: Search Key in simple cache-sentive skip list

```
Input: Key $k$
Output: Node containing key or Nil
1 Function search($k$):
2     $current \leftarrow header$;
3     for $i \leftarrow level$ down to 0 do
4         while $current.forward[i] \neq Nil \wedge current.forward[i].Key() \leq k$ do
5             $current \leftarrow current.forward[i]$;
6     return $current.search\_in\_array(k)$;
```

As the algorithm shows in Algorithm 1, from line 3 to line 5, it's very similar to a standard skip list, but in line 4,current.forward[i].Key() returns the least value(at the first position) of the array in the `current.forward[i]`. Out of the while loop, the current's forward Key() must be just greater than key, and if the key exists in the data structure, it must be in the current node array, so just search in the array of the current node in line 6.

3.3 Insertion

Algorithm 2: Insert Key into simple cache-sentive skip list

```
   Input: Key $k$
   Output: Nil
1  Function insert($k$):
2      $update \leftarrow$ array of size $(max\_level + 1)$;
3      $current \leftarrow header$;
4      for $i \leftarrow$ level down to 0 do
5          while $current.forward[i] \neq Nil \wedge current.forward[i].Key() \leq k$ do
6              $current \leftarrow current.forward[i]$;
7          $update[i] \leftarrow current$;
8      if $current \neq header \wedge current.is\_full() \wedge current.array[-1] > k$ then
9          $replace \leftarrow current.array.pop()$;
10         $current.insert\_into\_array(k)$;
           // CASE 1
11         $k \leftarrow replace$;
12     if $current \neq header \wedge \neg current.is\_full()$ then
           // CASE 2
13         $current.insert\_into\_array(k)$;
14     else
           // CASE 3
15         $current \leftarrow current.forward[0]$;
16         if $current \neq Nil \wedge current \neq header \wedge \neg current.is\_full()$ then
17             $current.insert\_into\_array(k)$;
18             return
19         $new\_level \leftarrow random\_level()$;
20         if $new\_level > level$ then
21             for $i \leftarrow level + 1$down to$new\_level$ do
22                 $update[i] \leftarrow header$;
23             $level \leftarrow new\_level$;
24         $new\_node \leftarrow Node([k], new\_level)$;
25         for $i \leftarrow 0$ to $new\_level$ do
26             $new\_node.forward[i] \leftarrow update[i].forward[i]$;
27             $update[i].forward[i] \leftarrow new\_node$;
```

In Algorithm 2, from line 2 to line 7, it's the same as a standard skip list. However, from line 8 to line 18, It should consider that level 0 nodes have a array. So there are several constraints which cannot be ignored:

- If the current node is a header, a new node should be created to insert the key. We don't want to insert data in the header; otherwise, it will be trouble for the deletion operation. So, in line 8-18, we exclude the header.
  There are some cases for insertion:
- CASE 1: current is not header, but current array size reach the maximum size(`current.is_full()` function to check it) and key value is between the least element (first position, Key() return it) and the largest element(last position, `current.array[-1]`). So in this case, current node's array is full, we make a small strick to deal with it, we pop the last position element, so the array is not full, and we insert the key in current node array(line 9,10) and it must be full again.then go on the next steps(for example, we replace 9 with 8 , and then insert 9 in another node, see Figure 2).
- CASE 2: current is not header, current node's array is not full, just insert the key in the array(line 12, 13). Then the insertion operation is finished.
- CASE 3: The key is lower than current.forward's the least value of its array(lines 4-7 has insurance for this), and the key is greater than the largest element of current node's array, So this key should be inserted into the current's forward node. We just insert it when the node is not full (from line 15-18), and we do not pop last element like line 9-11, for just to be simple.
- Other CASES: we should create a new node and insert the key in its array , and modify the forward pointers(from line 19 to 27)

3.4 Deletion

The deletion algorithm is shown in Algorithm 3, which should be noted that in line 5, the condition checks whether the current forward array's largest element is less than k, so out of the while loop, `current.forward[0]` is the first node whose largest element is equal to or greater than k, which means that if the key exists, it must be in the current node's `forward[0]` array. Line 8 to 14, delete the key in the selected array, and Line 15 to 21 checks whether the array is empty, if it is empty, we should delete the node from the data structure.

4 EVALUATION

4.1 Experimental Environment

The experiments were conducted in the following environments:

- **Hardware 1 (MacBook):** Apple M2 CPU, 16 GB RAM.
- **Hardware 2 (Server):** Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz, 24 GB DDR4 2133 MHz RAM. This server was used not only for performance testing but also specifically for analyzing CPU cache miss rates using appropriate profiling tools (e.g., `perf`).

Algorithm 3: Delete key from simple cache-sentive skip list

```
Input: Key $k$
Output: True if deleted, False otherwise
Function delete($k$):
    $update \leftarrow$ array of size $(max\_level + 1)$;
    $current \leftarrow header$;
    for $i \leftarrow level$ down to 0 do
        while $current.forward[i] \neq Nil \wedge current.forward[i].array[-1] < k$ do
            $current \leftarrow current.forward[i]$;
        $update[i] \leftarrow current$;
    $current \leftarrow current.forward[0]$;
    if $current = Nil$ then
        return $False$;
    else
        $res \leftarrow current.delete\_from\_array(k)$;
        if $\neg res$ then
            return $False$;
    if $len(current.array) = 0$ then
        for $i \leftarrow$ $0$ to $level$ do
            if $update[i].forward[i] \neq current$ then
                break;
            $update[i].forward[i] \leftarrow current.forward[i]$;
        while $level > 0 \wedge header.forward[level] = Nil$ do
            $level \leftarrow level - 1$;
    return $True$;
```

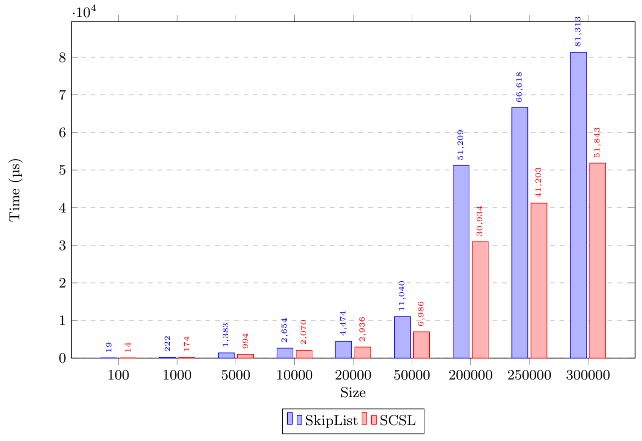

Figure 3: Insertion Time Comparison Between SkipList and SCSL at Different Scales(smaller is better) in Macos

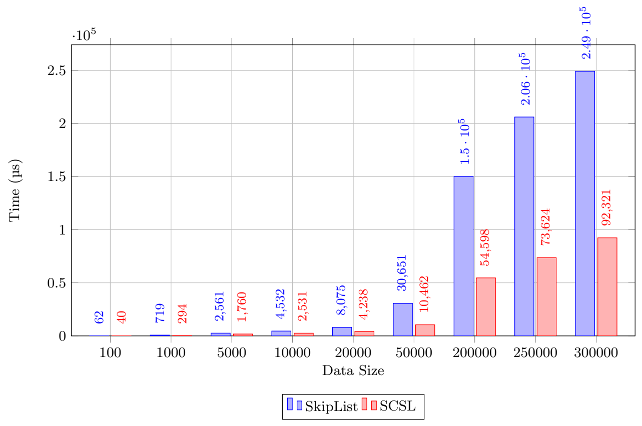

Figure 4: Insertion Time Comparison Between SkipList and SCSL at Different Scales(smaller is better) in Server

- **Compiler:** The algorithms were implemented in C++. The test programs were compiled using g++ (version specific to environment, e.g., 9.0 or higher) with the `-O3` optimization flag.
- **Benchmark Configuration:** Insertion operations in a skip list are prone to CPU cache misses due to dynamic memory allocations and pointer chasing. Therefore, our benchmarks primarily focus on insertion performance using randomly generated integer keys. The number of elements ranged from 100 to 300,000. Search performance was also evaluated. However, we do not present detailed search results as the performance difference between SCSSL and the standard skip list was found to be negligible for typical search operations. This is likely because the dominant factor in search is the traversal of levels, which has a similar path length, and the final array scan in SCSSL is very fast for small arrays. Both the standard skip list and SCSSL were configured with a maximum level (MAX_LEVEL) of 16. For SCSSL, the maximum array size within a data node was set to a value empirically found to be effective (e.g., 64 or 128, chosen to fit well within L1 cache lines).

4.2 Results and Discussion

On the MacBook (Apple M2 processor, Figure 3 SCSSL demonstrates a speedup of 1.28x-1.39x for small datasets (¡1,000 elements) during insertion. For medium-scale datasets (¿20,000 elements), the speedup increases to a range of 1.55x-1.65x. This trend indicates that the cache-friendly benefits of SCSSL become more pronounced as the dataset size grows, leading to more memory accesses.

On the server system (Intel i7-6700, Figure 4, SCSSL achieves even more significant speedups for insertions, reaching up to nearly 3x compared to the standard skip list for larger datasets (e.g., 300,000 elements). This enhanced improvement on the server platform might be attributed to

Test Case	Cache References	Cache Misses	Cache Misses Rate
`SCSL`	51,567,218	459,636	0.89%
`Standard Skip List`	142,249,678	47,251,323	33.22%

Table 1: Cache Performance Comparison Between SCSL and standard Skip list in server computer

differences in cache architecture, memory subsystem performance, or the relative cost of cache misses compared to the M2 architecture.

Table 1 presents the cache performance data for inserting 300,000 random elements on the server. The standard skip list exhibits a high L1 data cache miss rate of 33.22%. In contrast, SCSSL drastically reduces this rate to a mere 0.89%. This substantial reduction in cache misses directly corroborates our hypothesis that embedding arrays within data nodes effectively improves data locality and cache utilization, which translates into the observed speedups. The significantly lower number of cache references for SCSSL also suggests more efficient data access patterns.

5 CONCLUSIONS

We have presented SCSSL, a novel cache-sensitive skip list variant that introduces a simple yet highly effective modification: replacing single elements in data-level nodes with small, sorted arrays. This architectural change allows SCSSL to enhance CPU cache performance by improving data locality, while crucially maintaining the algorithmic simplicity that is a hallmark of skip lists.

Experimental results demonstrate significant performance improvements for insertion operations, particularly on systems where cache performance is a critical factor. The observed speedups of 1.2x-3x, coupled with a dramatic reduction in cache miss rates (e.g., from 33.22% to 0.89% in our server tests), validate the efficacy of this approach. SCSSL offers a practical and easily implementable method to boost skip list performance in modern computing environments without resorting to complex algorithmic changes or specialized hardware features. Its simplicity makes it an attractive option for a wide range of applications, from embedded systems to large-scale in-memory data stores.

Future work could explore optimal array sizes for different hardware platforms and key/value types, as well as investigate the application of SCSSL principles in concurrent skip list implementations.

REFERENCES

[1] Skip Lists: A Probabilistic Alternative to Balanced Trees
[2] Google. LevelDB. Available online: https://github.com/google/leveldb (accessed on 20 July 2023).
[3] Facebook. RocksDB. Available online: http://rocksdb.org/ (accessed on 20 July 2023).
[4] Apache. Welcome to Apache HBase™. Available online: https://hbase.apache.org/ (accessed on 20 July 2023).
[5] Cache-Sensitive Skip List: Efficient Range Queries on modern CPUs - informatik.hu-berlin.de.
[6] ESL: A High-Performance Skiplist with Express Lane https://www.mdpi.com/2076-3417/13/17/9925
[7] Redis, Accessed: 2024-01-09.