simple cache-sentive skip list

Yuchang Ke
aceking.ke@gmail.com

Bill Lv
ideaalloc@gmail.com

Weijie Zhang
zhangweijie@gmail.com

April 27, 2025

Abstract

We present a cache-sensitive skiplist that is very simple and easy to implement. It only requires a few small modifications to the original skiplist to adapt it to modern CPU architectures, resulting in a performance improvement of 1.2 1.6 times, in some system it can reach 2X or 3X. Unlike other optimizations, it does not require specialized instructions such as SIMD or multi-threaded programming. Instead, it takes advantage of modern compiler optimizations, making it more universal. This design can be used not only in in-memory databases or key-value stores, but also in embedded systems.

Keywords: skiplist; cache-sensitive; index structure

1 Introduction

In the field of computer science, ordered data structures are a fundamental tool for organizing and managing data. Efficient execution of search, delete, and insert operations is crucial for a wide range of applications, as it directly impacts overall system performance. To achieve high performance in ordered data structures, many methods have been developed, such as AVL trees and Red-Black trees. However, these structures are relatively complex due to the need for rebalancing. This changed when William Pugh introduced the skip list in 1990[1]. It introduced a randomized data structure that simplified implementation compared to traditional balanced trees.

Thanks to its simplicity, the skip list has been adopted by various systems, including databases such as LevelDB[2], Redis[7], and RocksDB[3], Hbase[4] in-memory caches, and the inverted index in Lucene.

Skip list can be divided into multiple levels, one data level and multiple index levels. Data level is the lowest level(level 0), which stores all the data nodes. And the index levels provide short paths to access the data node at the data level, and all the levels are linked lists. Because of this, these data nodes and index nodes are non-contiguous in memory address. This causes the performance to be degraded by 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.

With the emergence of new technologies and computing architectures, particularly multi-core CPUs and vectorization, novel variants of skip lists have been developed. Such as Cache-Sensitive Skiplist (CSSL)[5], which merges nodes at the index level into a single array, which are named fast lanes, and all nodes in the fast lane are sorted. This design is more cache-friendly. It also uses SIMD instructions, ESL(A High-Performance Skiplist with Express Lane[6]) also merges the nodes in index levels into a single array, and they use version-based lock protocol and lock-free operation log to implement concurrency.

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.

There are 3 key points: 1) At the data level, we use a sorted array with a fixed maximum length. Since the keys are integers, we choose a maximum length of 128 or less. 2) For all index levels, we use only the first element of the array as a pivot for comparison. 3) The last element of the array is considered solely during insertion operations, and only under particular conditions.

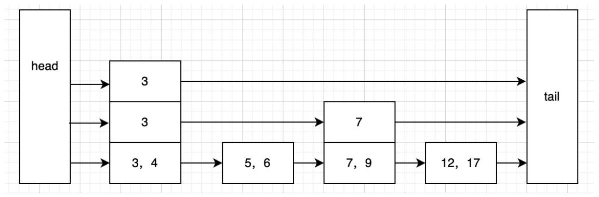

Figure 1: A simple cache-sensitive skip list

Our key contributions include: 1. Introducing a vector, which maintains the algorithm's complexity while improving CPU cache hit rate due to its contiguous storage. 2. Reducing the skiplist's level count and pointer overhead by incorporating a vector. 3. Presenting vector-based insertion, search, and deletion algorithms for data nodes, offering a novel perspective for database index design.

3 Algorithm

This section gives the algorithm for search, insertion, and deletion. The insertion operation inserts new specified data into the data structure, and the search operation finds data and returns it. The deletion operation deletes data.

In SCSSL, at the data level, data is stored in a single array with a fixed maximum size. As the same as the standard skip list, the level of the node is chosen randomly when the node is created. And a level i node has the same number of forward pointers (also i), the maximum of levels is a constant MaxLevel. It also has a header that leads the search of the data structure, and the last nodes of all levels' forward pointers point to NIL.

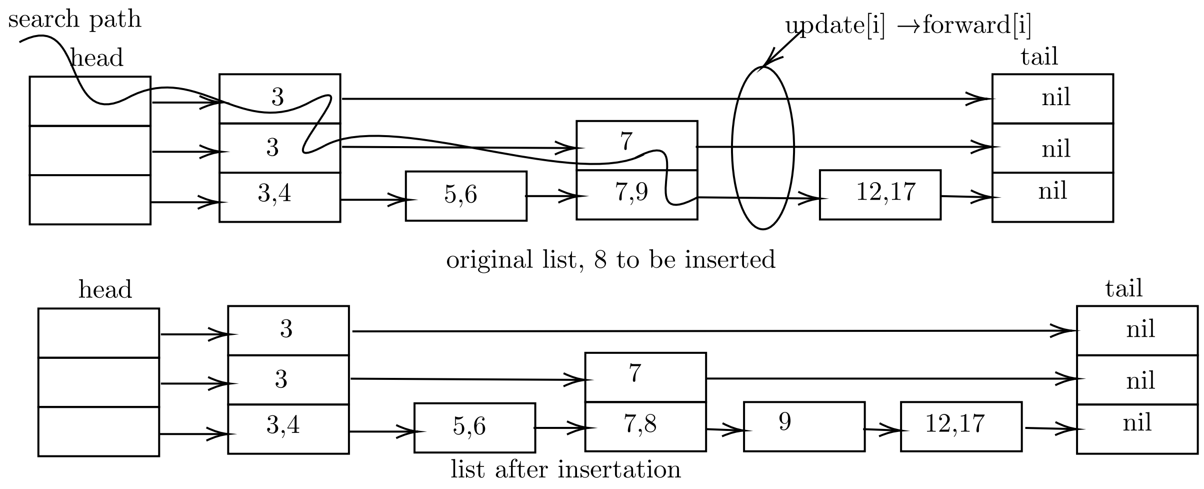

Figure 2: insert 8 into simple cache-sentive skip list(max array size =2)

3.1 Initialization

The initialization operator is elementary, just create a header with MaxLevel forwards array, and set them to Nil. To simplify the algorithm, we set the header's array with a single value, -inf.

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         $current.insert\_into\_array(k)$;
11         $k \leftarrow replace$;
12     if $current \neq header \wedge \neg current.is\_full()$ then
13         $current.insert\_into\_array(k)$;
14     else
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(ines 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

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

```
Input: Key $k$
Output: True if deleted, False otherwise
1  Function delete($k$):
2      $update \leftarrow$ array of size $(max\_level + 1)$;
3      $current \leftarrow header$;
4      for $i \leftarrow level$ down to $0$ do
5          while $current.forward[i] \neq Nil \wedge current.forward[i].array[-1] < k$ do
6              $current \leftarrow current.forward[i]$;
7          $update[i] \leftarrow current$;
8      $current \leftarrow current.forward[0]$;
9      if $current = Nil$ then
10         return $False$;
11     else
12         $res \leftarrow current.delete\_from\_array(k)$;
13         if $\neg res$ then
14             return $False$;
15     if $len(current.array) = 0$ then
16         for $i \leftarrow 0$ to $level$ do
17             if $update[i].forward[i] \neq current$ then
18                 break;
19             $update[i].forward[i] \leftarrow current.forward[i]$;
20         while $level > 0 \wedge header.forward[level] = Nil$ do
21             $level \leftarrow level - 1$;
22         return $True$;
```

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

Here are the overall environments:

Hardware: We chose a MacBook whose CPU is Apple M2, and 16 GB of memory. and another server computer with Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz , 24G ddr4 2133 memory. We use this server computer not only to test the performance, but also to test the CPU cache miss.

Compiler: We complete the algorithm in cpp codes, The test program is compiled by g++ with O3 optimization option.

Benchmark: Because insert data in skip list, will more possibily cause CPU cache missing. so we test insertion with random data. and also give search test. The scale of data elements is from 100 to 300000. We do not present the search performance because the performance gap is negligible.

4.2 Performace

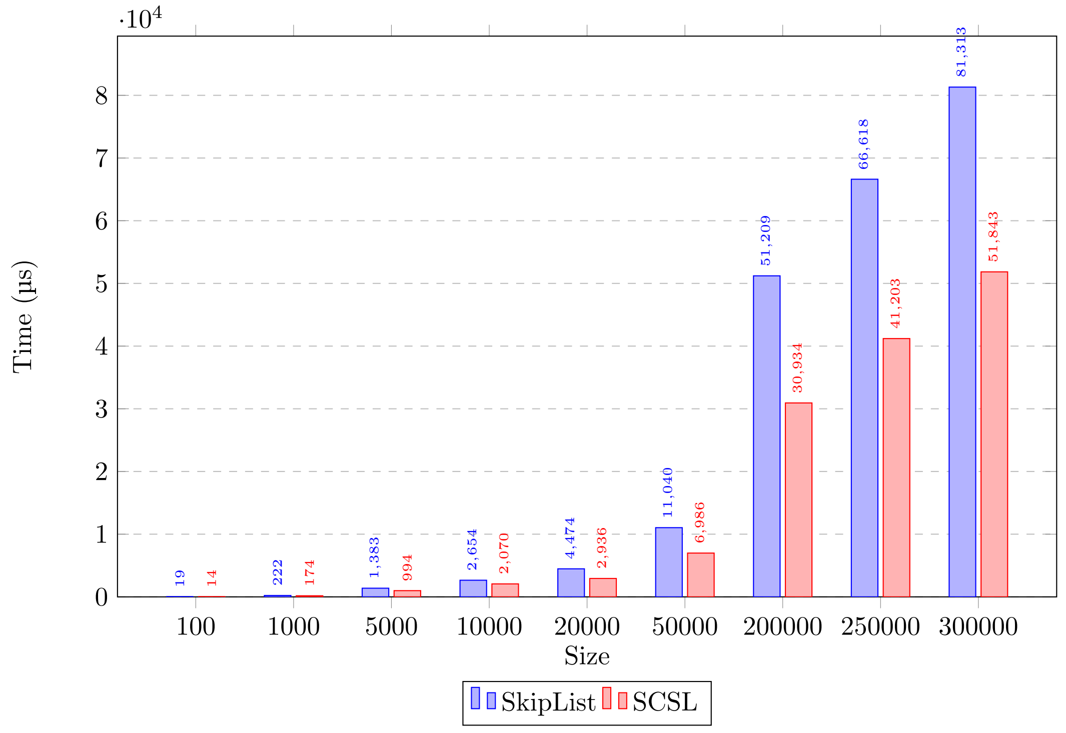

Figure 3: Insertion Time Comparison Between SkipList and SCSL at Different Scales(smaller is better) in Macos

We evaluated the standard skip list and SCSL with a maximum level of 16. Figure 3 presents the insertion performance comparison between the standard skip list and simple cache skip list in a Macbook. It shows that under small scale of data (¡ 1000) simple cache skip list can speed up to 1.28 1.39 times, and under medium scale (over 20000) , it can speed up to 1.55 1.65 times, as the scale increasing, the more benefit for performance. Figure 4 also shows that, in server computer, the simple cache skip list can speed up even 3 times.

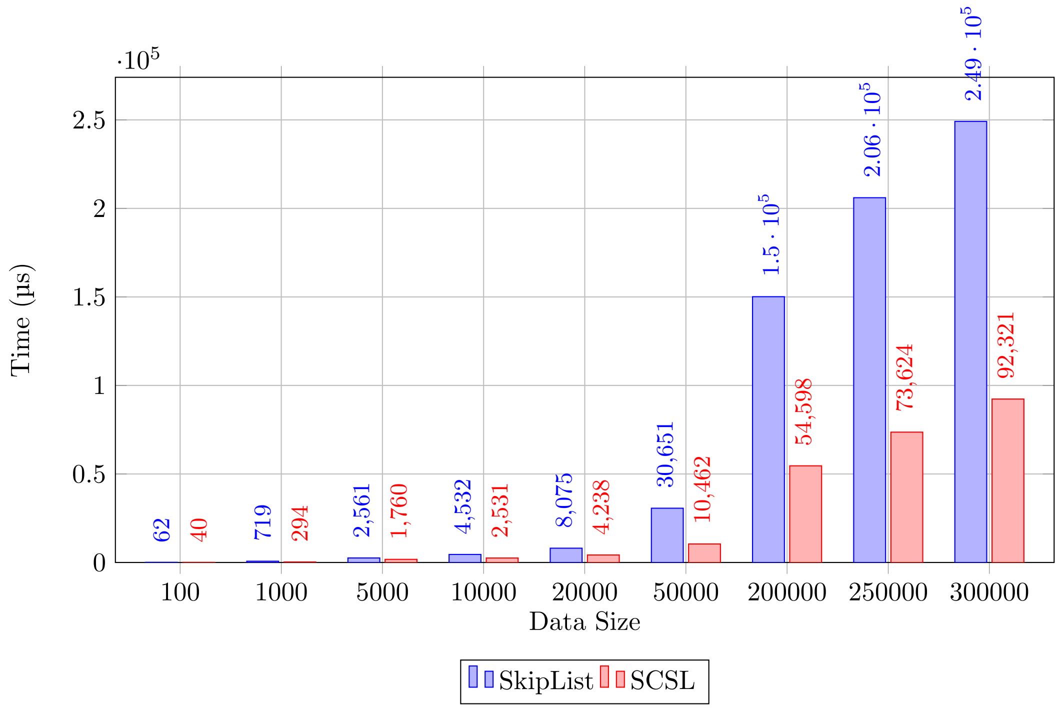

Figure 4: Insertion Time Comparison Between SkipList and SCSL at Different Scales(smaller is better) in Server

Test Case	Cache References	Cache Misses	Cache Miss 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

As shown in the table above(table 1), the standard skip list in the test achieves a 33.22% cache-miss rate, whereas SCSL has only a 0.89% cache-miss rate. It also proves that adding arrays to data nodes can work very well.

5 Conclusions

We present a new method to implement a cache-sensitive skip list, SCSL, which only changes a single element in the data-level nodes to an array. SCSL effectively leverages CPU caches by using data stored in continuous memory, and the algorithm is simple. Additionally, the performance results are also significant.

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.