A Concurrent Skip List Balanced on Search

Fei Mei[1], Qiang Cao[1(✉)], Fei Wu[1], and Hongyan Li[2]

[1] Wuhan National Laboratory for Optoelectronics,
Key Laboratory of Information Storage System, Ministry of Education,
Huazhong University of Science and Technology, Wuhan, China
{meifei,caoqiang,wufei}@hust.edu.cn
[2] HuBei University of Economics, Wuhan, China

Abstract. We introduce a skip list, T-list, that updates the index on the search process by recording critical positions in the traverse of the index nodes. T-list uses a step counter to decide when and where to build a new index node and guarantees that the index node is generated in critical position unlike the probabilistic skip list, resulting in an efficient index structure. Meanwhile T-list does not enforce strong constraints to the overall structure unlike the deterministic skip list, thus eliminates a lot of maintaining work. Worst case in T-list can be efficiently repaired by a few requests that traverse the most part of list. Building a new index node in T-list only modifies the contents of two adjacent nodes, enabling the algorithm friendly to concurrent accessing. Experimental results show that compared to the skip list used in a popular application - LevelDB, T-list can construct a more efficient and stable index structure and the insertion and search performances are improved by 17.8% and 33.3% respectively. T-list also scales well with the threads number in the multi-core machine.

Keywords: Skip list · Concurrent list · Key value store · Index structure

1 Introduction

Skip list is a structure that is easy to implement and allows fast search and insertion, originally introduced by Pugh et al. [15] as an alternative to the balanced tree. A standard skip list comprises multiple layers of nodes. The bottom layer contains the inserted nodes each with a unique key and the user data (i.e., value). The nodes in a higher layer can be regarded as a subset of the lower layer but only contains the keys and pointers to other nodes and act as indexes. A new node is first inserted into the bottom layer and gets a height in probability, then in each layer within the height a indexing node with the same key is also created. Due to the simplicity of concept and easiness of implementation, skip list has been adopted by many LSM-tree based key-value stores such as BigTable [1], HBase [7], Cassandra [11], LevelDB [4], and MemSQL [17]. In different use scenarios it plays different roles. For example, in LevelDB the new

Y. Dou et al. (Eds.): APPT 2017, LNCS 10561, pp. 117–128, 2017.
DOI: 10.1007/978-3-319-67952-5_11

inserted key-value pairs are stored in a skip list that is a part of the whole user data, while MemSQL uses skip list as a secondary index [12] for its clustered user data.

State-of-the-art implementations of skip lists are categorized into two classes with respective shortcomings. The first one is styled by the implementations based on the original idea that generates height by probability, called probabilistic skip list. Although probability mechanism can expect to obtain the $logN$ search complexity [15], it lacks stability and is not easy for purposely optimizing, such as space locality, because a new node gets its height independently without considering the status of the nearby nodes and is not to be changed in the future once determined. Defective index nodes that degrade the indexing efficiency can also be generated unpredictably. The other class consists of the implementations that enforce a set of predefined rules and constraints to the structure, called deterministic skip list. On each update to a node, the deterministic list forcedly adjusts its whole structure to restore the defined rules. Since the adjusting process leads to many check operations and must maintains information of nearby nodes, the deterministic list is complex to implement and not friendly to concurrent accesses in multi-thread environments.

In this paper we present T-list, a skip list construction algorithm that maintains loose rules on the overall index structure when new node joins, but strengthen it gradually on the processing of search requests. Since an insertion operation always executes a search phase to determine the position for inserting, the index structure can also be built up under 100% insertion workload. When processing operations that may change the structure, T-list only modifies at most two nodes on a layer, making it multi-thread friendly because a write operation only involves locking of two adjacent nodes on a single layer. T-list may generate thin index structure for particular requests sequence. For instance, T-list does generate index for a reversely ordered sequence of keys because the search phase for each insertion needs only on step. However, the index can be built up if some search requests that need more steps are processed. In other words, T-list generates the index on need, which can be a better choice for the memory component of the LSM-tree based key-value stores mentioned above.

The rest of the paper is organized as follows. In Sect. 2 we briefly discuss the related works. The design and implementation are detailed in Sect. 3. Section 4 presents the evaluation results. At last we conclude the work and discuss future plans in Sect. 5.

2 Background and Related Works

Three basic operations are defined on a general list structure, insertion, search and deletion. The insertion and deletion operations always need a search phase to determine where to insert the new node or which node to delete. Based on the basic operations, skip list has been researched for fast searches as well as for favorable concurrent insertions.

Pugh et al. first presented the skip list [15] as an alternative to tree structure for its expected $logN$ search complexity and implemented a lock-based concurrent version [14]. Munro et al. [13] proposed to enforce pre-defined rules on the skip list, to achieve deterministic structure. One of deterministic structure is the 1–2 skip list that adjusts the structure each time after inserting a new node to hold the rules non-violated by recursively inspecting the nearby information until the whole structure became balanced. Another construction method introduced in this paper was the top-down 1-2-3 skip list, which can be regarded as a remedy policy that repairs the structure in the next time and the distance of any two nodes is allowed to be 3 even if it has the equivalent property with 1-2 skip list. T-list is similar to the top-down list in that it moves the index building work to the search phase, and the insertion operation finishes immediately after linking the new node into the bottom layer. However, the top-down list must check the total number of nodes in the gap from which the search process descends, and adjusts the structure to keep the nodes between the gap not exceeding the predefined value (i.e., 3 for the 1-2-3 list). Instread, T-list counts the steps when the search progresses and raises the height of a node when the steps reaches to the predefined value without inspecting all nodes between the gap. Other works are optimizations based on the above concepts and most of them focus on concurrent environment. Herlihy designed the lock-based skip list [10] that is built on the lazy-list [8], which acquires locks for all nodes that need to be modified when inserting or removing nodes. Non-blocking concurrent mechanisms [3,5,9,19] achieves concurrency by using atomic instructions. Skip lists are also used in network environments. Singh et al. presented the algorithms for achieving concurrency in a distributed deterministic 1-2 skip list [18] and a self-stabilizing peer-to-peer network maintenance algorithm is designed by Clouser et al. [2].

Except the above works, skip lists are also adopted by key-value stores as the in-memory component. For example, LevelDB, a popular key-value store, implements a probabilistic skip list (Lev-list) as the in-memory structure [4]. In Lev-list a configurable variable referred to as *branch* (default to 4) controls the general structure. A new node gets a height by the probability related to the value of branch. Such as, the branch value set to 4 means the height will be 2 in probability of $\frac{1}{4}$, and be 3 in probability of $\frac{1}{4^2}$, and so on. This results a list in which a layer expects to have 1/4 nodes of the layer under it, and the list height expects to be $log_4 N$ in which N is the total number of nodes in the bottom layer. Lev-list does not explicitly implement deletion operation but instead marks the node as deleted, known as logical deletion [3,6,16]. The deletion marker is also useful in LevelDB for removing keys that exist on the disk. The lifetime of Lev-list is temporary, as it is periodically transformed and destroyed, and a new empty list is created to receive new requests.

With a study of these researches and the practical implementations, we found that the probabilistic skip list is easy to implement because of no need to maintain rules on the overall structure, but has degraded performance for its non-perfect index structure, while deterministic skip list is the other way around.

The design of T-list aims to resolve the shortcomings that exist in the probabilistic skip list and the deterministic skip list, in order to make a better trade off in practical use case such as LSM-tree environment. Probabilistic skip list builds the index without knowledge of the nearby nodes, leading to defective indexing nodes that degrade the search efficiency. In the contrary, deterministic skip list enforces special rules and constraints on the structure and performs check operations based on the intensity of nearby nodes, leading to heavy maintaining work. T-list decouples the insertion operation to two distinct phases, search and linking. The search phase traverses the list to find a bottom node after which the new node should be linked. In the process of traversing, indexes are built by the traverse steps. The linking phase simply links the new node to the bottom layer.

3 Design and Implementation

3.1 Structure Overview

We start with a figurative description of the structure overview of T-list. First let us assume a sorted link list without indexing structure on it. For each search request, it must traverse the list nodes one by one until it finds a key equal or greater than the requested key. Now we regard each node as a station, and the link between two adjacent nodes as a path. The search request is performed by a traveler who walks along the path station by station to find the target station that contains the requested key. Walking from one station to the next is counted as one step. Each time when he have walked a fixed number of steps (e.g. two) he will want to build a higher station in the higher path that is more convenient. Next time when the traveler accepts a search request he will first walk along the higher path on which he walks faster than along the lower one, until to a station he must go down. The point is that when he traverses on the higher path he as well keeps building more higher stations if he walks the fixed number of steps. Insertion request is served in such a way that after finding the insertion position on the bottom path the new node (station) is simply linked.

In the remained of the paper, when we say a node or station on the bottom path, they have the same meaning, except that node is used when we refer to a key while stations is used when we refer to traversing. Figure 1 shows a simple T-list example with 8 nodes/stations on the bottom path. Each station on other paths at the same vertical line contains a pointer to the node so the key can be accessed quickly anywhere on the search process.

The fixed number of steps in the above description is defined as *span* of the list, which has the equal role as the *branch* in LevelDB. A *span* of two means that a higher station is to be built every time when the traveler walks two steps. All stations on the same vertical line are transportable. That is, each station contains two pointers to its upper and lower stations respectively. The header stations plays the same role as the others except that it does not contain a key. The last station in each path points to null. The height of the list is determined

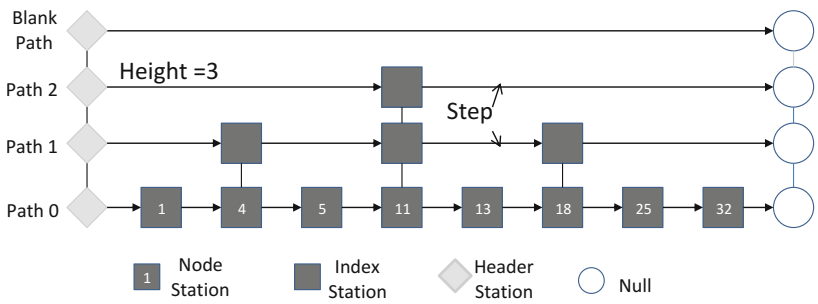


Fig. 1. A T-list example with the span configured to 2. A step on a path usually means two steps on its lower path.

by the highest path. A blank path above the highest path is set with the header station as the last station. The blank path is used for assisting adding station and is not counted for the height.

3.2 Search Procedure

Before introducing other operations, we first give a brief description of how to search a key in T-list. Search operation in T-list has two versions. One is called *PureTravel* that works the same way as in common skip lists. The other is called *BuildTravel* that is the core function in T-list, which plays the role of constructing the indexing structure.

PureTravel begins from the highest path and descends at a station if it is the last station on the path or its next station contains a greater key than the target, until finally it descends to the bottom path. Traveling on the bottom path will report the search result. If no key is found, the last node that has a lesser key than the target will be returned (Fig. 2 targeting the key 26), otherwise the node with the key matching the target is returned. Matching can also be met on other paths above the bottom, in which case the matching node is returned immediately.

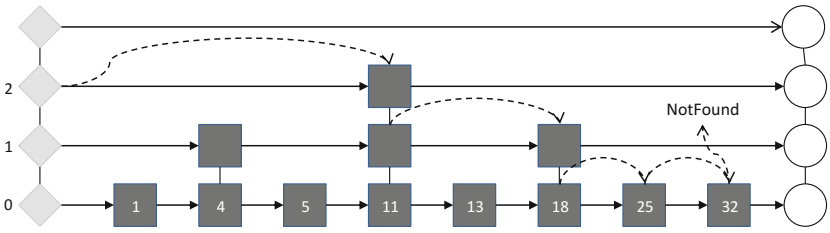


Fig. 2. *PureTravel to Find the Key 26.* The traveler begins at path 2 and ends at path 0 with the node 25, because the next node of node 25 has a greater key than the target.

BuildTravel is based on PureTravel, besides that it maintains two markers when the search traverses on the path and calls *BuildStation* when necessary. One marker is the step counter which indicates how many steps the traveler has walked. Each time the counter reaches the configured *span* value, the BuildStation function is called to build a station on top of the current station (referred to as *base_station*) the traveler suspends on. BuildStaion assures whether the station should be indeed built (check). If the check is passed, a station is built, i.e. adding a higher station on the top of the *base_station*. The other marker is a station pointer that always points to the higher station from which the traveler lastly descends (*pre_higher_station*). At the beginning of the search, the *pre_higher_station* points to the header station on the blank path. The step is reset to 0 each time the descending occurs or after the BuildStation is called. An example is illustrated in Fig. 3, in which we assume the key 33 is searched and the *pre_higher_station* is pointing to the node 18 in path 1 when the traveler walks to 32. At this time, BuildStation is called since the step counter reaches 2.

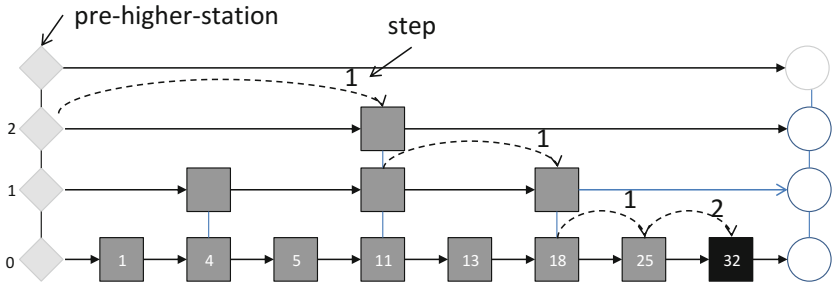


Fig. 3. When searching to the node 32, the step counter reaches 2 (span of the list), so the BuildStation function is called to build station on it and the new station will be linked after pre_higher_station (node 18 in the path 1)

The BuildStation function only needs the *base_station* and *pre_higher_station* to know where to build the new station. It first checks whether the station should be really built. If the station next to the *base_station* has a higher station, the checking would fail and the building operation is canceled, avoiding redundant station in the higher path. In a special case when the check is passed and the higher path is the blank path, meaning that the new built station will make the blank path non-blank, the height of the list will be increased by 1. T-list guarantees the new station is not redundant and has accurate *span* with the *pre_higher_station*. In any case, the step counter is reset to 0 after the BuildStation returns.

3.3 Insertion Procedure

The list is initiated as an empty list that only has the header node pointing to null. This empty list still has a path 0 (the bottom path) and a blank path above it, and its height is regarded as 1. In other words, the path 0 of an empty list has the same structure as the blank path.

An insertion operation is decoupled to two phases: (1) Searching the position on the path 0 where the new key should be placed, and (2) Linking the new node into the location. The location should be between two adjacent nodes that the previous one has the key lesser than the new key and the next one has the key greater than the new key, or the next one is null. The search phase is the main work of the insertion operation and is accomplished by BuildTravel in our implementation, and the linking phase is a simple operation just linking the new node in the found location.

The BuildTravel is called in the search phase of insert operation because potentially higher stations may need to be built as the new joined node increases the number of the bottom stations. Theoretically, the stations built by BuildTravel have no benefits to the calling search process. However, the new node can be taken into consideration in the later building procedures if any request executes BuildTravel across it. There can be a T-list with a thin index structure even if the path 0 has many nodes under special workloads, but this situation is the result that the processed requests need no long travelling. The thin index structure can grow to be strong if a few requests have traveled most part of the list. For example, T-list provides a Perfect function that at most needs $logN$ requests to make a perfect index structure on a bare list that only has one path (path 0) with N nodes. One can execute the insertion process by calling PureTravel, and running the BuildTravel with background threads to build index. However, our experiments showed that running the BuildTravel background is not always prompt to satisfy search efficiency of the insertion.

3.4 Concurrent Operations

We implement a lock-based concurrent mechanism for multiple threads to operate the list without breaking the list structure. As the PureTravel only reads the memory content, we leave the threads free to do such operations. The concurrent mechanism focuses on resolving multiple threads contenting for adding stations to the list.

In summary, the BuildStation operation and the linking operation are two actions that modify the structure of T-list. The BuildStation only adds new stations in non-bottom paths at a time, while the linking only adds a new station (node) to the bottom path. With this property in mind, we design a simple control mechanism that allocates one lock for each path, defined as path lock(PL). Actually, fine grained lock can be applied on T-list to enable multiple threads operating concurrently on a same path. There are totally the max_height (i.e. 40) number of PLs initiated for use, each responsible for a path.

The following steps are executed by a thread in order to add a station.

(i) Decide to add a station. This indicates that the thread has determined to add a station X on path i between two adjacent stations A and B.
(ii) Acquire the lock on path i (PL[i]). If other threads are changing the structure on path i, this thread must wait.

(iii) Reconfirm the real location of the new station if the lock is got. There is possibility that other threads have added stations between A and B, so it is necessary to reconfirm whether it is needed to add this new station and where to add it. The reconfirming is done by looking forward the path i from station A until it meets a station whose key is greater than that of station X. This station is marked as station B′, and the station previous to it is marked as station A′.
(iv) Check if it is really needed to add station between station A′ and B′. If the check is passed, the station would be linked between A′ and B′, otherwise nothing is done.
(v) Release PL[i].

When a thread decides to call BuildStation on a particular path $i(0 \leq i \leq height)$, it first records the station (station A) after which the new one will be linked, and then acquires the lock PL[i]. After path i has been locked, no other threads can change the structure on it, therefor the following operations can be done safely with no disturbances. However, other threads may have added new stations on path i when the PL is acquired, and the real position may turn to be the new station added by other threads. In this case if the thread directly adds its new station, the structure would be broken. To avoid this, a look-forward operation is done to find the real position. After the look-forward, the new station can be safely added, because this operation is done under the lock, other threads can not disturb the operation.

3.5 Other Implementation Issues

In this section we talk about some other implementation issues in T-list. Deletion operation in the prototype of T-list is implemented by logically marking the node as deleted, while all the stations on it are preserved for indexing. There are physical deletion discusses in the top-down skip list [13] and other works [3,10]. In the practical use case as in LevelDB, the deletion operation is a special insertion operation targeting the same key but replaces the value with a deletion marker, which can be regarded as a logical deletion mechanism. We also implement Perfect function to build a perfect indexing structure for the list. This function traverses all the paths from bottom to top by a modified BuildTravel function. The Destroy function is a cleanup procedure after the list life ends. It releases all the resources the list has acquired from the OS.

4 Evaluation

We evaluate T-list with the following purposes:

– Examine the performance of T-list for different workloads.
– Verify the structure property generated by T-list.
– Evaluate the performance scalability for increasing number of the threads.

In the basic evaluation experiments, we compare the results with the skip-list used in LevelDB, which is a single-thread version. To make the comparison fair, we extract the code only related to the skip list structure from LevelDB and make it only serve integral keys. This list will be referred to as Lev-list in the following text. The configuration of *branch* in Lev-list had the same effect as the *span* in T-list, and their values had the equivalent influence on the structure, so we used the word *span* to refer both configurations. We then evaluate the scalability for increasing threads of T-list.

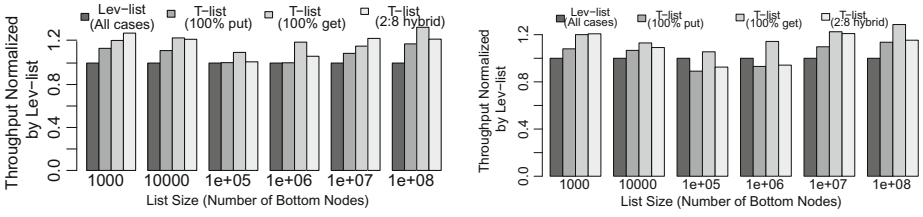
Our evaluation experiments are executed on a machine equipped with Intel Xeon Processor E3-1270 v2 (8M Cache, 3.50 GHz) which supports eight threads and four 8-GB DDR3 memory cards. Each experiment is run 5 times and the average value is computed as the final result.

4.1 Performance

We used three kinds of workloads to evaluate the performances of the two structures.

(1) 100% put. All operations are insertion requests.
(2) 100% get. All operations are search requests.
(3) 2:8 hybrid. For each incoming request, the probability of insertion is 20% and search is 80%.

The put workload fills the list from blank to the given size by random keys with uniform distribution. The get workload searches a million random keys from the list that is filled by the put workload. The hybrid workload fills the list in the same manner as the put workload, except that a lot of search requests are mixed in the process. The size of the list is varied in different number of keys(from 10^3 to 10^8). The *span* is configured with two different values, 2 and 4. Each experiment selects a workload, a list size, and a *span* value to run.

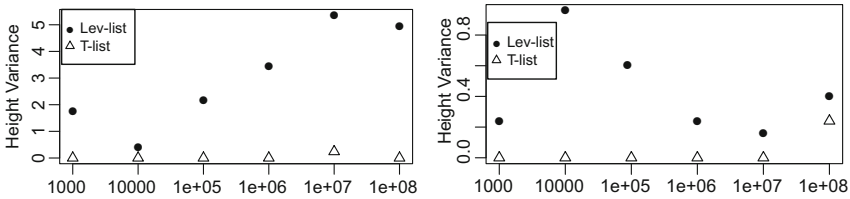


(a) *span*=2 (b) *span*=4

Fig. 4. Normalized performance with different sized lists. For 100% put, the list is inserted with random keys from blank to the size. For 100% get, the list is first constructed by 100% put to the size, and then search 1 million random keys. For 20% put 80% get, every operation is determined by this ratio and 5 times the size operations are processed.

Fig. 4a and b give the results of all the experiments normalized by Lev-list grouped by the list size, with span configured to 2 and 4 respectively. We can

see that when *span* = 2, T-list has better performance in all cases. When *span* is configured to 4, T-list also performs better except when the list size is 10^5 and 10^6. As T-list builds index stations on the search phase when inserting a node and generates faster paths for later operations, its advantage may not show up when the list size is small under the put workload. When the list size increases, T-list can build a stable index structure and a single insert operation benefits well from it. Lev-list uses probability mechanism to build index nodes that can have varied list height in a same experiment. Figure 5 demonstrates the height variances that are calculated from the put workload experiments. Although Lev-list has the expected height for a given list size, it intends to generates a more higher structure than the expected value, which makes the search operations traverses more paths to the bottom. This can be reflected from the result of the get workload, in which T-list always performs better than Lev-list in any cases no matter the *span* value. As the get workload works on a static list, the experiment results can reflect that T-list generates a more perfect and efficient index structures.



(a) *span*=2 (b) *span*=4

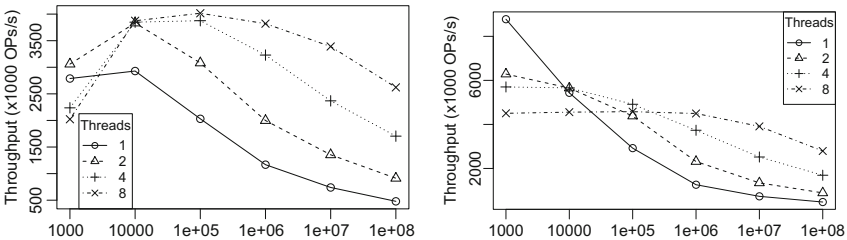
Fig. 5. Height variance of constructing different sized lists. The result comes from the 100% put workload. Every experiment is repeated five times and the heights are recorded for computing the variance.

4.2 Multi-thread

We use the put and get workloads introduced above to evaluate the concurrency of T-list. The threads number is varied in 1, 2, 4 and 8. The *span* is configured to 2. For the put workload, each experiment creates a number of threads by the configuration and all of them perform random insertions to the list until the list reaches the defined size. For the get workload, firstly one thread is used to fill a list with a defined number of random keys, then a number of threads by the configuration are created to do random searches on the list until totally 10 million requests are processed.

Figure 6a and b show the results of the put and get workloads with different threads running on varied sized lists. The figures show that, while multi-threads is more efficient for the large sized list, it degrades the performance when the list size is small. This is comprehensible since there are overheads of the threads management work, which emerge to be significantly when the overheads on normal operations are small.

Specifically for the put workload, a lock is shared in all threads for adding a station on a same path. In small sized list the threads are more likely to contend for locks because only few paths can be operated at the same time. With the list size increasing, the overheads for contending locks are distributed as the search route becomes long. As search operations do not need to acquire locks, theoretically they do not suffer the contention overheads that are significant in the small list. However, the scalability of multi-threads for the get workload also is achieved when the size increases. This can be resulted from the high proportion of scheduling overheads in concurrently accessing small portion of data.



(a) Multi-thread Put (b) Multi-thread Get

Fig. 6. Threads scalability in different sized lists, the span configured to 2. The threads number are varied from 1 to 8. For the put workload, different number of threads are created and they concurrently insert random keys to the list until it reaches the size. For the get workload, a list is first created to the determined size by one thread with random keys, and then different number of threads are created to do concurrent search operations in this list (totally a million random keys are processed).

5 Conclusion

In this paper we introduce and implement a skip list construction algorithm, called T-list, that employs a special search procedure to build indexes according to the traversing steps on the search progress. Building-on-search makes the index construction work distributed on the search phases so as the heavy operations on the new nodes are relieved. Besides, T-list maintains loose constraint rules to make the index structure self-adjustable according to the workload patterns for insert-intensive workloads. On the other hand, concurrent operations can benefit from T-list as each update to the structure only needs to lock two nodes on a single path. The evaluations on the prototype show that T-list achieves better performance than the skip list used in LevelDB. For multi-core environments it also performs well in the scalability with the increasing number of threads.

Nevertheless, more potential properties can be exploited from T-list. As the real-world workloads are varied, a more intelligent algorithm that can fit varied environments is worthwhile to be studied. We plan to improve the algorithm in the future by leveraging its adjustable characteristic to make it aware of and intelligent to the complex and varied use cases.

References

1. Chang, F., et al.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. (TOCS) **26**(2), 4 (2008)
2. Clouser, T., Nesterenko, M., Scheideler, C.: Tiara: a self-stabilizing deterministic skip list. In: Kulkarni, S., Schiper, A. (eds.) SSS 2008. LNCS, vol. 5340, pp. 124–140. Springer, Heidelberg (2008). doi:10.1007/978-3-540-89335-6_12
3. Crain, T., Gramoli, V., Raynal, M.: No hot spot non-blocking skip list. In: IEEE 33rd International Conference on Distributed Computing Systems (ICDCS), pp. 196–205. IEEE (2013)
4. Dean, J., Ghemawat, S.: LevelDB. In: Retrieved, vol. 1, p. 12 (2012)
5. Fraser, K.: Practical lock-freedom. Ph.D. thesis. University of Cambridge (2004)
6. Gao, F.: A concurrent skip list implementation with RTM and HLE (2014)
7. George, L.: HBase: the definitive guide. In: O'Reilly Media, Inc. (2011)
8. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N., Shavit, N.: A lazy concurrent list-based set algorithm. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 3–16. Springer, Heidelberg (2006). doi:10.1007/11795490_3
9. Henson, V.: A Brief history of UNIX file systems (2004)
10. Herlihy, M., Lev, Y., Luchangco, V., Shavit, N.: A simple optimistic skiplist algorithm. In: Prencipe, G., Zaks, S. (eds.) SIROCCO 2007. LNCS, vol. 4474, pp. 124–138. Springer, Heidelberg (2007). doi:10.1007/978-3-540-72951-8_11
11. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Syst. Rev. **44**(2), 35–40 (2010)
12. MemSQL. MemSQL Documentation. http://docs.memsql.com/4.1/concepts/indexes/
13. Munro, J.I., Papadakis, T., Sedgewick, R.: Deterministic skip lists. In: Proceedings of the Third Annual ACM-SIAM Symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, pp. 367–375 (1992)
14. Pugh, W.: Concurrent maintenance of skip lists (1990)
15. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM **33**(6), 668–676 (1990)
16. Sémon, P., et al.: Lazy skip-lists: an algorithm for fast hybridization-expansion quantum Monte Carlo. Phys. Rev. B **90**(7), 075149 (2014)
17. Shamgunov, N.: The MemSQL in-memory database system. In: IMDM@ VLDB (2014)
18. Singh, R., Chakraborty, S., Karmakar, S.: Concurrent deterministic 1–2 skip list in distributed message passing systems. Int. J. Parallel Emergent Distrib. Syst. **30**(2), 135–174 (2015)
19. Timnat, S., Braginsky, A., Kogan, A., Petrank, E.: Wait-free linked-lists. In: Baldoni, R., Flocchini, P., Binoy, R. (eds.) OPODIS 2012. LNCS, vol. 7702, pp. 330–344. Springer, Heidelberg (2012). doi:10.1007/978-3-642-35476-2_23