Parallelizing Skip Lists for In-memory Multi-core Database Systems

Zhongle Xie*, Qingchao Cai*, H. V. Jagadish†, Beng Chin Ooi* and Weng-Fai Wong*
* National University of Singapore † University of Michigan
*{zhongle, caiqc, ooibc, wongwf}@comp.nus.edu.sg †jag@umich.edu

***Abstract*—Due to the coarse granularity of data accesses and the heavy use of latches, indices in the B-tree family are not efficient for in-memory databases, especially in the context of today's multi-core architecture. In this paper, we study the parallelizability of skip lists for the parallel and concurrent environment, and present PSL, a Parallel in-memory Skip List that lends itself naturally to the multi-core environment, particularly with non-uniform memory access. For each query, PSL traverses the index in a Breadth-First-Search (BFS) to find the list node with the matching key, and exploits SIMD processing to speed up this process. Furthermore, PSL distributes incoming queries among multiple execution threads disjointly and uniformly to eliminate the use of latches and achieve a high parallelizability. The experimental results show that PSL is comparable to a read-only index, FAST, in terms of read performance, and outperforms ART and Masstree respectively by up to 30% and 5x for a variety of workloads.**

I. Introduction

With exponentially increasing memory sizes and falling prices, it is now frequently possible to accommodate the entire database and its associated indices in memory, thereby completely eliminating the significant overheads of slow disk accesses. Non-volatile memory such as phase-change memory looming on the horizon is destined to push the envelope further. Traditional database indices, e.g., B^+-tree [1], that were mainly optimized for disk accesses, are no longer suitable for in-memory databases since they may suffer from poor cache utilization due to their hierarchical structure, coarse granularity of data access and poor parallelism.

The heavy use of latches during the lookup of B^+-tree n-odes is another factor hindering the adoption of B^+-tree as in-memory database indices. In fact, merely latch inspection can result in a significant penalty due to potential cache flush. Latch modification is even more expensive, as it will invalidate the latch replicas located at the caches of other cores, and force the threads running on these cores to reread the latch for inspection, incurring significant bandwidth cost. According to [2], an in-memory database system today can spend almost one quarter of its time in latching. Hence, removing the need for latches is a must to achieve exceptional parallelism.

The above issues motivate us to reexamine the *skip list* [3] as a possible candidate as the base indexing structure in place of the B^+-tree (or B-tree). Skip list employs a probabilistic model to build multiple linked lists such that each linked list consists of nodes selected according to this model from the list at the next level. Skip lists have been known for several decades now, but are seeing increased popularity in recent years because of their appropriateness for main memory indexing.

In this paper, we present **PSL**, a **P**arallel in-memory **S**kip **L**ist. In PSL, we employ a fine-grained processing strategy to avoid using latches. Queries are organized into batches and each batch is processed simultaneously with multiple threads. Given a query, PSL traverses the index in a breadth-first manner to find the corresponding list node. To handle the case in which two threads find the same list node for some keys, PSL adjusts the query workload among execution threads to ensure that each list node to be modified is accessed by exactly one thread, thereby eliminating the need for latches.

As we implement this idea, there are additional considerations from modern computer architecture that we have to keep in mind to achieve good performance. Memory is typically organized hierarchically with multiple levels of cache. Cache addresses are tied at the bit level to memory addresses, so designing to cache lines is necessary. To that end, we design a novel storage layout for PSL by clustering together the adjacent list nodes of each index layer. This storage layout not only improves cache exploitation, but also enables the use of *Single instruction multiple data* (SIMD) during the lookup of indexed keys.

We conduct an extensive performance study between P-SL and three state-of-the-art indices, namely Masstree [4], ART [5] and FAST [6]. The results show that PSL performs slightly better than the read-only FAST in terms of search query processing, and up to $4-5\times$ better than Masstree and $0.3\times$ better than ART on a variety of query workloads.

II. Related Work

There are two main directions towards enhancing the query performance of B^+-trees in in-memory environments: cache exploitation and latch-freeness. Rao et al. [7] present a cache-sensitive search tree (CSS-tree), where nodes are stored in a contiguous memory area such that the address of each node can be arithmetically computed, eliminating the use of child pointers in each node. Masstree [4] is a trie of B^+-tree to handle keys of arbitrary length with a high query throughput. However, its performance is still restricted by the locks upon which it relies to update records.

Due to the overhead incurred by latches, a large amount of effort has been committed to building latch-free index trees. The Bw-tree [8], which manages its memory layout in a page-oriented manner and is hence well-suited for flash solid state disks, is a representative by using Compare-And-Swap (CAS) instructions to achieve latch-freeness. Sewall et al. propose a latch-free concurrent B^+-Tree, PALM [9], which adopts bulk synchronous parallel (BSP) model to process queries

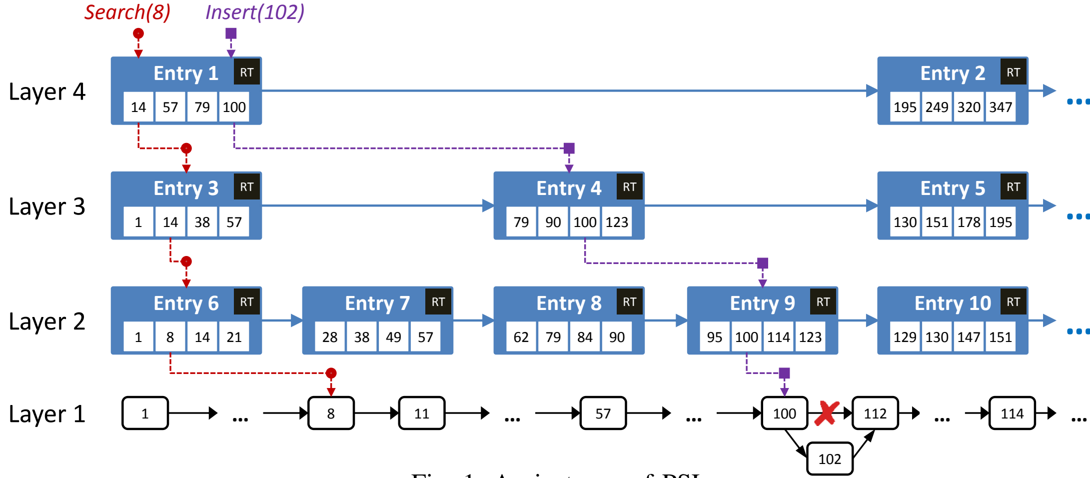


Fig. 1: An instance of PSL

in batches. FAST [6] uses a similar searching method, and achieves twice the query throughput of PALM at a cost of not being able to make updates to the index tree.

Compared with B^+-tree, a skip list has approximately the same average search performance, but requires much less effort to implement. In particular, even a latch-free implementation, which is notoriously difficult for B$^+$ trees, can be easily achieved for skip lists by using CAS instructions [10]. Abraham et al. combine skip lists and B-trees for efficient query processing [11]. Skip lists are more parallelizable than B^+-tree because of the fine-grained data access and relaxed structure hierarchy. However, naïve linked list based implementation has poor cache utilization due to the nature of linked lists.

III. INDEX DESCRIPTION

In a traditional skip list, since nodes are dynamically allocated, they do not reside within a contiguous memory area. Non-contiguous storage of nodes causes cache misses during key search and renders impossible SIMD processing, which requires the operands to be stored within a contiguous memory area. We shall elaborate on how PSL overcomes these two limitations and meanwhile achieves latch-free query processing.

A. Structure

Like a traditional skip list, PSL also consists of multiple layers of sorted linked lists. The bottommost layer is a linked list of data nodes, whose definition is given in Definition 1. The remaining (upper) layers are called *index layers*, each of which is composed of the keys randomly selected with a fixed probability from those contained in the linked list of the next lower layer.

Definition 1: A data node α is a triplet (κ, p, Γ) where κ is a key, p is the pointer to the value associated with κ, and Γ is the height of κ, representing the number of linked lists where key κ is present. We say a key $\kappa \in S$ if there exists a data node α in the bottom layer of the index, S, such that $\alpha.\kappa = \kappa$.

Compared to a B^+-tree, the update operation in a skip list is simplified and parallelizable in that nodes can be directly inserted into or removed from linked lists without re-balancing. PSL reserves this parallelizability by organizing its bottom layer as a single linked list, and moves a step further towards hardware consciousness. Each index layer of PSL consists of a linked list of *entries*, each of which contains several keys that can be loaded into a single SIMD register. Figure 1 gives an instance of PSL where each of the three index layers is a linked list of 4-key entries. The "RT" in the figure represents routing table, which contains the address of the next entry to access for each possible comparison result between the lookup key and the current entry,

Given an initial dataset, the index layers of PSL can be constructed in a bottom-up manner. We only need to scan the bottom layer once to fill the high-level entries as well as the associated routing table, This construction process is $O(n)$ where n is the number of records at the bottom layer, and typically takes less than 0.2s for 16M records when running on a 2.0 GHz CPU core. Further, the construction process can be parallelized, and hence can be sped up using more resources.

B. Queries and Algorithms

PSL, like most indexing structures, supports three types of queries, namely search, insert and delete. The update query is considered as insert here. An abstraction of the queries is given in Definition 2.

Definition 2: A query, denoted by q, is a triplet $(t, \kappa, [p])$ where t and κ are the type and key of q, respectively, and if t is insert, p provides the pointer to the new value associated with key κ.

We now define the query set Q in Definition 3. There are two points worth mentioning in this definition. First, the queries in a query set are in non-decreasing order of the query key k, and the reason for doing so will be elaborated in Section III-B4. Second, a query set Q only contains point queries, and we will show how such a query set can be constructed and leveraged to answer range queries in Section III-B5.

Definition 3: A query set Q is given by $Q = \{q_i | 1 \leq i \leq N\}$ where N is the number of queries in Q, q_i is a query defined in Definition 2, and $q_i.\kappa \leq q_j.\kappa$ *iff* $i < j$. For a query q, we define the corresponding interception, I_q, as the data node with the largest key among those in $\{\alpha | \alpha.\Gamma > 1, \alpha.\kappa \leq q.\kappa\}$.

Algorithm 1: Query processing

```
Input  : S, PSL index
         Q, query set
         t_1, ..., t_{N_T}, N_T threads
Output: R, Result Set
1  R = ∅;
2  for i = 1 → N_T do
3      Q_i = partition(Q, N_T);
4  /* traverse the index layers to get interceptions */
5  foreach Thread t_i do
6      Π_i = traverse(Q_i, S);
7  waitTillAllDone();
8  /* redistribute query workload */
9  foreach Thread t_i do
10     redistribute(Π_i, Q_i, t_i);
11 /* query execution */
12 foreach Thread t_i do
13     R_i = execute(Π_i, Q_i);
14 waitTillAllDone();
15 return ∪R_i;
```

PSL accepts a query set as input, and employs a batch technique to process the queries in the input. The detailed query processing of PSL is given in Algorithm 1. First, the query set Q is evenly partitioned into disjoint subsets according to the number of threads, and the i-th subset is allocated to thread i for processing (line 3). The ordered set of queries allocated to a thread is also a query set defined in Definition 3, and we call it a query batch in order to differentiate it from the input query set. Each thread traverses the index layers and generates for each query in its query batch an interception which is also defined in Definition 3 (line 5 and 6). After this search process, the resultant interceptions are leveraged to adjust query batches among execution threads such that each thread is able to *safely* execute all the queries assigned to it after the adjustment (line 9 and 10). The *waitTillAllDone* function may cause contention as the threads need to wait till other threads finish their work. However, this barrier affect the performance little according to our evaluation. Finally, each thread individually executes the queries in its query batch (line 12 and 13). The whole procedure is exemplified in Figure 1, where two queries making up a query set are collectively processed by two threads. Following the red arrows (the dash lines), thread 1 traverses downwards to fetch the data node with key 8 and thread 2 moves along the purple arrows (the dot lines) to insert the data node with key 102.

1) Traversing the Index Layer: For each query key, the traversal starts from the top level of the index layers and moves forward along this level until an entry containing a larger key is encountered, upon which it moves on to the next level and proceeds as it does in the previous level. The traversal terminates when it is about to leave the last level of the index layers, and records the first data node that will be encountered in the bottom layer as the interception for the current query.

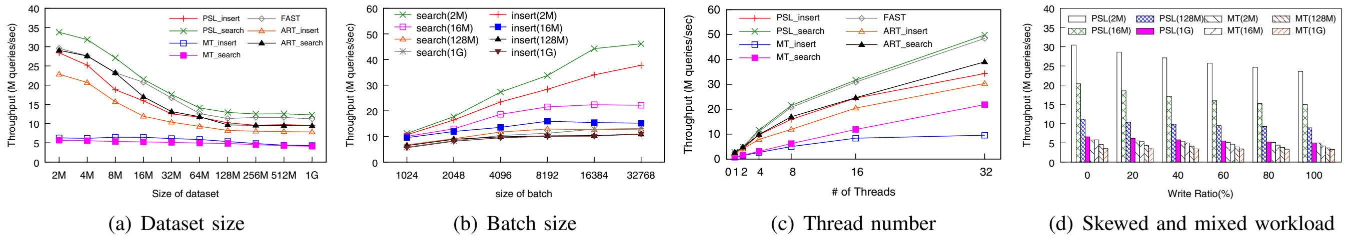
We exploit Single Instruction Multiple Data (SIMD) processing to accelerate the traversal. In particular, multiple keys within an entry can be simultaneously compared with the query key using SIMD, which significantly reduces the number of comparisons, and this is the main reason we put multiple keys in each entry. In our implementation, keys in an entry exactly occupy a whole SIMD vector, and can be loaded into the SIMD registers directly. We also generate a routing table for each entry during the construction of PSL to guide index traversal. For each possible result of SIMD comparison, the routing table contains the address of the next entry to visit.

2) Redistributing Query Workload: Given the interception set output by index layer traversal, a thread can find for each allocated query q the data node with the largest key that is not greater than $q.\kappa$ by walking along the bottom layer, starting from I_q. However, it is possible that two queries allocated to two adjacent threads have the same interception, leading to contention between them. To handle this case, each thread iterates backward over its interception set until it finds an interception different from the first interception of the next thread, and hands over the queries corresponding to the iterated interceptions to the next thread (except the last thread) if it is not a search query. After the adjustment, a thread can individually execute the allocated query set without contending for data nodes with other threads.

3) Query Execution: For each query q, an execution thread iterates over the bottom layer to get the final result, starting from the corresponding interception, and executes the query against the data node with the largest key that is less than or equal to $q.\kappa$. If the query type is delete, we do not remove the data node immediately from the bottom layer, but merely set a flag F_{del} instead, which is served for latch-free query processing. For the search query, the F_{del} flag of the resultant data node will be checked to decide the validity of its pointer. For the update query, a new data node will be inserted into the bottom layer if the query key does not match that of the resultant data node. Unlike a typical skip list, PSL only allocates a random height for the new node, but does not update the index layers immediately.

We employ a background thread in PSL to realize the deferred updates to the index layers. It monitors the update operations made to the index, and rebuilds the whole index once the number of update operations hits a pre-defined threshold. The new index layer will be put into use after all the running threads complete the processing of current query batch, and meanwhile the old index layer will be discarded. The rebuilding process is highly parallelizable and can thus be shortened with more rebuilding threads.

4) Latch-free Processing: Query processing in PSL is naturally latch-free. In the traversal of the index layers, the use of latches can be avoided since the access to each entry is read-only. For the adjustment of query workload, each thread communicates with its adjacent threads via messages and thus does not rely on latches. In addition, each query q allocated to thread i after the adjustment of query workload satisfies $I^i_{q_1}.\kappa \leq q.\kappa < I^{i+1}_{q_1}.\kappa$, where $I^i_{q_1}$ and $I^{i+1}_{q_1}$ are the first element in the interception sets of thread i and $i+1$, respectively. Consequently, thread i can individually execute without latches all its queries except those which require reading $I^{i+1}_{q_1}$ or inserting a new node directly before $I^{i+1}_{q_1}$, since the data nodes that will be accessed during the execution of these queries will never be accessed by other threads. The remaining queries can still be executed without latches as the first interception of each thread will never be deleted.



(a) Dataset size (b) Batch size (c) Thread number (d) Skewed and mixed workload

Fig. 2: Experiment Results

5) Range Query: Range query is supported in PSL. Given a set of range queries we first sort them according to the lower bound of their key range and then construct a query set defined in Definition 3 using these lower bounds. This query set is then distributed among a set of threads to find the corresponding interceptions, as in the case of point queries. The redistribution of query workload, however, is slightly different. We use $I^i_{q_1}$ to denote the first element in the interception set of thread i. For each allocated query with a key range of $[\kappa_s, \kappa_e]$, where $\kappa_e \geq I^{i+1}_{q_1}$, thread i partitions it into two queries with the key ranges being $[\kappa_s, I^{i+1}_{q_1}.\kappa)$ and $[I^{i+1}_{q_1}.\kappa, \kappa_e]$, respectively, and hands over the second query to thread $i+1$. After the redistribution, each thread then executes the allocated queries one by one. Starting from the corresponding interception, PSL iterates over the bottom layer to find the first data node within the key range, and then executes the query upon it and each following data node until the upper bound is encountered. The final result of an original range query can be acquired by combining the result of corresponding partitioned queries.

IV. Performance Evaluation

We evaluate the performance of PSL on a platform with 512 GB memory evenly distributed among four NUMA nodes. Each NUMA node is equipped with an Intel Xeon 7540 processor, which supports 128-bit SIMD processing, and has 18MB L3 cache and six 2 GHz cores. By default, the dataset has 16M records, and we use 8 threads to process query batches, each consisting of 8192 queries. The query workload is generated using *Yahoo! Cloud Serving Benchmark* (YCSB) [12]. The lookup keys in the workload have a length of 4 bytes and follow a Zipfian distribution. The parameter θ for skewed and mixed workload is 1.2. The whole index layer is asynchronously rebuilt after a fixed number (15% of the original dataset size) of data nodes have been changed.

For comparison, the results of three state-of-the-art indices, Masstree [4] (marked as "MT" in the figures), FAST [6] and ART [5], under the same experiment setting are also given, whenever possible. As can be observed in Figure 2, PSL beats all the competitors among all comparisons. Generally, PSL performs 2 to 5 times faster than Masstree and is 20% to 35% better than ART. FAST is slightly worse than PSL but it only supports search queries. PSL and FAST exhibit a similar trend in the increasing rate in terms of scalability (number of threads), and achieve the best scalability as demonstrated by the substantial gap in the increasing rate between them and the other two. We can observe that query skew has only a little impact on the performance of PSL in terms of query processing.

V. Conclusions and Future Work

In this paper, we argue that skip list, due to its high parallezability, is a better candidate for in-memory indexing than B^+-tree in concurrent environment. Based on this argument, we propose various parallelizing and optimization strategies and design a cache-friendly, latch-free and hardware-conscious index called PSL to provide efficient support of both point query and range query. PSL consists of index layers, which are in charge of key search, and a bottom layer responsible for data retrieval, and the layout of the index layer is carefully designed such that SIMD processing can be applied to accelerate key search. The performance study shows that PSL achieves a similar query performance to a read-only index, FAST, and meanwhile respectively performs up to 5x and 30% faster than Masstree and ART, the other two state-of-the-art indices.

Acknowledgment

This research was in part supported by the National Research Foundation, Prime Ministers Office, Singapore, under its Competitive Research Programme (CRP Award No. NRFCRP8-2011-08). Zhongle Xie's work was partially supported by the National Research Foundation Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme (E2S2-SP2 project).

References

[1] D. Comer, "Ubiquitous B-tree," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121–137, 1979.

[2] M. Stonebraker, "The Traditional RDBMS Wisdom is (Almost Certainly) All Wrong," http://slideshot.epfl.ch/play/suri_stonebraker, 2013.

[3] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," *CACM*, vol. 33, no. 6, pp. 668–676, 1990.

[4] Y. Mao, E. Kohler, and R. T. Morris, "Cache Craftiness for Fast Multicore Key-Value Storage," in *EuroSys*, 2012, pp. 183–196.

[5] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *ICDE*, 2013, pp. 38–49.

[6] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs," in *SIGMOD*, 2010, pp. 339–350.

[7] J. Rao and K. A. Ross, "Cache Conscious Indexing for Decision-Support in Main Memory," in *VLDB*, 1999, pp. 78–89.

[8] J. Levandoski, D. Lomet, and S. Sengupta, "The Bw-tree: A B-tree for New Hardware Platforms," in *ICDE*, 2013, pp. 302–313.

[9] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey, "PALM: Parallel Architecture-friendly Latch-Free Modifications to B^+ Trees on Many-Core Processors," *PVLDB*, vol. 4, no. 11, pp. 795–806, 2011.

[10] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[11] I. Abraham, J. Aspnes, and J. Yuan, "Skip B-trees," in *OPODIS*, 2006, pp. 366–380.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *SOCC*, 2010.