A Comprehensive Performance Evaluation of Modern in-Memory Indices

Zhongle Xie† Qingchao Cai† Gang Chen‡ Rui Mao§ Meihui Zhang⋆

†*National University of Singapore* ‡*Zhejiang University* §*Shenzhen University* ⋆*Beijing Institute of Technology*

†{zhongle, caiqc}@comp.nus.edu.sg ‡cg@zju.edu.cn §mao@szu.edu.cn ⋆meihui_zhang@bit.edu.cn

***Abstract*—Due to poor cache utilization and latching contention, the B-tree like structures, which have been heavily used in traditional databases, are not suitable for modern in-memory databases running over multi-core infrastructure. To address the problem, several in-memory indices, such as FAST, Masstree, BwTree, ART and PSL, have recently been proposed, and they show good performance in concurrent settings.**

Given the various design choices and implementation techniques adopted by these indices, it is therefore important to understand how these techniques and properties actually affect the indexing performance. To this end, we conduct a comprehensive performance study to compare these indices from multiple perspectives, including query throughput, scalability, latency, memory consumption as well as cache/branch miss rate, using various query workloads with different characteristics. Our results indicate that there is no one-size-fits-all solution. For example, PSL achieves better query throughput for most settings, but occupies more memory space and can incur a large overhead in updating the index. Nevertheless, the huge performance gain renders the exploitation of modern hardware features indispensable for modern database indices.

***Keywords*-Index; NUMA; Skip List; B$^+$-tree**

I. Introduction

It is now a norm to hold the database and its indices in main memory due to the steep expanding rate of memory size as well as the exponentially decreasing rate of memory prices. Traditional database indices, which have been originally designed for disk accesses, such as the B$^+$-tree [1], are not suitable for such scenarios as they suffer from poor cache utilization during look-up, as discussed in [2]. To address this issue, many new indexing structures have been proposed in the past two decades [3], [4], [5], [6]. The key ideas behind these indices include aligning internal nodes with cache lines [3], enabling huge pages [4] and pre-fetching [6]. Masstree [4] exploits all the above optimizations, and shows that the optimization techniques result in an impressive performance gain for tree structures in modern in-memory settings.

A natural direction towards improving the performance of the B$^+$-tree is to exploit the rich parallelism inside modern multi-core CPUs. However, the latch used in the original B$^+$-trees significantly restricts this exploitation because when there are multiple threads accessing the same node within the index, it keeps all the threads, except the one holding the latch, waiting idly. Hence, there have been a number of latch-free indexing structures recently proposed. A major technique employed is using Compare-And-Swap (CAS) operations to provide semantic guarantee on atomic updates and avoid read-update contention. BwTree [7], by using CAS operations, can perform substantially better than traditional B$^+$-trees. A different approach to the latch issue is adopted in PALM [8] and PSL [9]. In these works, each thread is only responsible for processing indexing queries which fall into a certain partition, and hence does not raise contention with other threads, which in turn eliminates the needs of latches.

Another promising technique to enhance indexing performance is using Single Instruction Multiple Data (SIMD) to boost the key comparison during index traversal. This technique has been widely adopted in the recently proposed indexing structures such as Adaptive Radix Tree (ART) [10], FAST [11], PALM [8] and PSL [9].

Skip list [12] has been invented for nearly three decades and has been gaining popularity recently in the context of main-memory databases. It employs a probabilistic model to build multiple levels of linked lists, each of which consists of pivot pointers extracted from the next level according to the probabilistic model. A recent work [9] incorporates the optimizations that were originally presented for tree indices [3], [4], [5], [6], into the skip list, and the resulted index, PSL, can achieve a comparable performance to ART. In addition, for Non-Uniform Memory Access (NUMA) architecture, PSL further employs an adaptive threading mechanism to minimize the number of remote memory access, which is substantially slower than local memory access.

All of the above works [4], [7], [9], [10], [11] have demonstrated a promising performance in terms of indexing throughput, but most of them lack a thorough performance profiling, which, in addition to index throughput, should also take into consideration many other factors, such as latency, scalability, resource usage and adaptiveness to various workloads. For instance, PSL does not measure the latency of index queries and memory consumption; ART only runs 12 threads in their multi-threaded evaluations instead of a step-by-step configuration, and the impact of workload skewness on indexing performing is not studied in the BwTree.

Given the promising indexing performance and notice-

able properties (in terms of design and optimization) of those recently proposed in-memory indexing structures, a systematic and synthesized performance comparison among them is highly desirable. First, it enables us to evaluate the impact that the various design philosophies and optimization techniques applied in these works exert on the indexing performance. Second, by benchmarking these indices under various workloads, we can identify for each index the characteristics of workloads that this index can exploit for better performance. Last but not least, an in-depth understanding of indexing performance drawn from a comprehensive comparison can provide a guidance on the design of new indexing structures as well as help database designers and maintainers to decide the most appropriate indexing structure.

In this paper, we intend to bridge this gap by conducting a fair and comprehensive comparison between these state-of-the-art indices, including **FAST**, **Masstree**, **BwTree**, **ART** and **PSL**. The main contributions of this paper are summarized as follows:

- We study five recently proposed indices, i.e., FAST [11], Masstree [4], BwTree [7], ART [10] and PSL [9], and identify the effectiveness of common optimization techniques, including hardware dependent features such as SIMD, NUMA and HTM.
- We conduct an extensive performance evaluation as well as a comparison study among the five indices. The results show that PSL can achieve higher query throughput under most settings, and precedes the other four indices by a factor ranging from 5% to 500%, and meanwhile, FAST and ART are most memory efficient, which make them stand out in certain cases with skewed key distribution in the index.
- By analyzing the experimental results, we conclude that there is no one-size-fits-all candidate for in-memory database indices due to the differences among workloads and datasets. However, the exploitation of modern hardware features, such as SIMD processing and multiple cores, should be seriously taken into account when designing the index for in-memory databases for huge performance gain it brings.

The remainder of this paper is structured as follows. Section II presents the background. The five indices studied in this paper are introduced in Section III. Section IV discusses the design considerations faced by the five indices. Section V presents a comprehensive performance study and analysis for the five indicates. Section VI concludes this work.

II. Background

In this section, we introduce the background for modern in-memory database indices from four aspects as follows.

A. B^+-tree and related optimizations

The B^+-tree [1] is perhaps the most widely used index in database systems. There are two main directions towards enhancing the indexing performance of B^+-trees in in-memory environments: cache utilization and parallelism. Rao et al. [3] 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 efficiently handle keys of arbitrary length. With all the optimizations presented in [3], [5], [6] enabled, Masstree can achieve a high query throughput. However, its performance is still restricted by the locks upon which it relies to update records. Recently, a hybrid index with a set of guidelines facilitating index building is presented in [13], which shows that the hybrid index, which can be constructed both from tree structures and skip lists, can achieve tolerable throughput with a significant reduction on the memory overhead. In addition, software prefetching is also widely applied to hide the slow memory access [4], [6], [9].

Recently, for its ability to automatically detect the concurrent accesses to shared memory and abort the respective transaction, hardware transactional memory (HTM) has been used extensively in database area to implement efficient concurrency control [14]. However, due to the high abort rate resulted from exclusive accesses and coarse access granularity, it is non-trivial to exploit HTM for database indices, as discussed in [15] and also shown in our experiment.

B. Latch-Free Processing

Due to the overhead incurred by latches, a large amount of effort has been committed to building index trees using Compare-And-Swap (CAS) instructions to avoid the overhead of latches. Brown et al. [16] implement an efficient non-blocking red-black tree with fine-grained synchronization while Braginsky and Petrank [17] present a lock-free B^+-tree implemented with single-word CAS instructions. The Bw-tree [7], developed for Hekaton [18], [19], is another latch-free B-tree which manages its memory layout in a page-oriented manner and is well-suited for flash solid state disks where random writes are costlier than sequential ones.

There have also been many works trying to enforce natural latch-free processing by design. Sewall et al. [8] propose a latch-free concurrent B^+-tree, named PALM, which adopts bulk synchronous parallel (BSP) model to process queries in batches. FAST [11] 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. Similarly, PSL employs a query partitioning strategy that ensures each index node is modified by at most one thread. The idea that only one single thread is responsible for a partition is similar to physiological partitioning (PLP) proposed by Pandis et al. [20] for transaction processing.

C. Skip List

Skip lists [12] are considered to be an alternative to B^+-tree and have been imported into commercial databases recently [21]. Compared with B^+-tree, a skip list has same average search performance, but requires much less effort to implement. In particular, even a latch-free implementation, which is notoriously difficult for B^+-tree, can be easily achieved for skip lists using CAS instructions [22]. Crain et al. [23] propose new skip list algorithms to avoid contention on hot spots. Abraham et al. [24] combine skip lists and B-trees for efficient query processing. Stefan et al. [25] introduce Cache-Sensitive Skip List (CSSL) optimizing range queries with a CPU-friendly layout.

Skip lists are more parallelizable than the B^+-tree because of the fine-grained data access and relaxed structure hierarchy. However, naïve linked list based implementation of skip lists has poor cache utilization due to the nature of linked lists. In PSL, this problem is addressed by separating the index layers from the bottom most layer such that the layout of the index layers is optimized for cache utilization and hence enables an efficient search of keys.

D. Single Instruction Multiple Data

Single Instruction Multiple Data (SIMD) processing has been extensively used in database research to boost the performance of database operations. Chhugani et al. [26] show how the performance of merge sort, a classical sort algorithm, can be improved, when equipped with SIMD. Similarly, it has been shown that the SIMD-based implementation of many database operators, including scan, aggregation, indexing and join, perform much better than its non-SIMD counterpart [27], [28].

Recently, tree-based in-memory indices leveraging SIMD have been proposed to speed up query processing [8], [11]. FAST [11], a read only binary tree, can achieve an extremely high throughput of query processing as a consequence of SIMD processing and enhanced cache consciousness enabled by a carefully designed memory layout of tree nodes. Another representative of SIMD-enabled B^+-tree is PALM [8], which overcomes the FAST's limitation of not being able to support updates at the expense of decreased query throughput. The Adaptive Radix Tree (ART) [10] also enables SIMD processing via a customized memory layout for the internal tree nodes.

III. Index Overview

We select five state-of-the-art indices, namely FAST [11], Masstree [4], BwTree [7], ART [10] and PSL [9] for our performance study. The structures of the five indices are depicted in Figure 1.

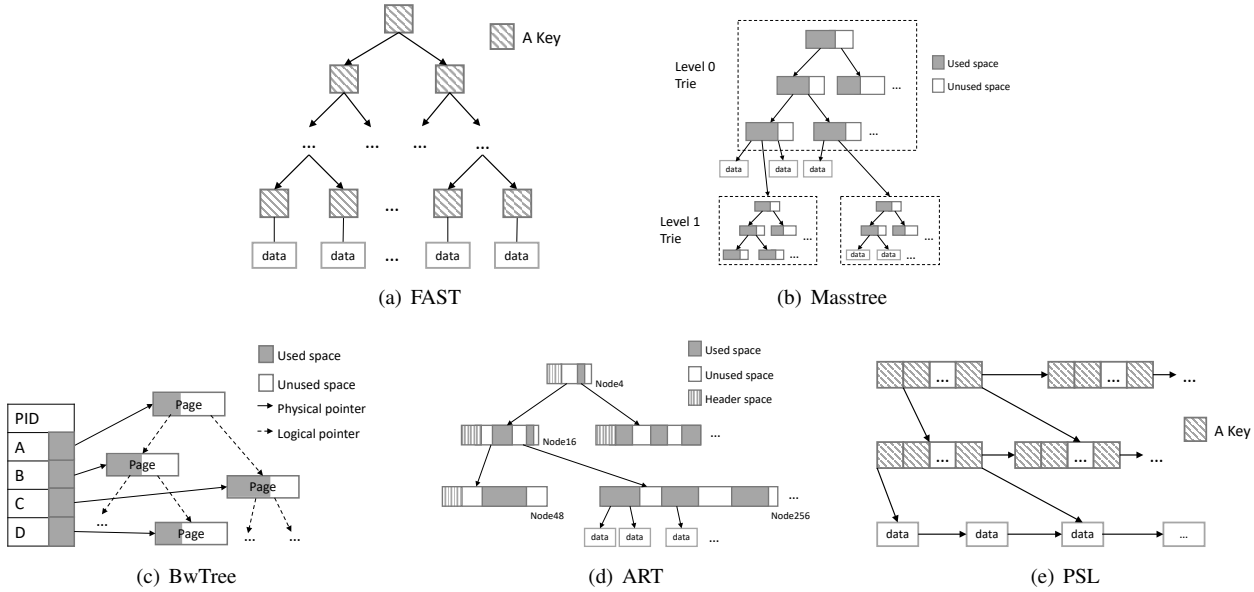
A. FAST

FAST, as shown in Figure 1(a), is a binary search tree with superb performance under read-only workloads. For each node at an even level (the root is at level 0), FAST packs it and its two children into a single fat node and eliminates the necessity of maintaining the two respective pointers. The fat nodes are organized such that the children of each fat node can be derived from some simple arithmetical computation, which removes the need of a route table used in PSL, and significantly reduce the storage space. The processing of search queries in FAST is logically the same as that of a conventional binary search tree. The difference is that when the search process reaches an even level, all three keys in the respective fat node would be loaded simultaneously into an SIMD register, and a single SIMD instruction is then used to compare the three keys against the key of interest and generates a four-bit mask. The mask represents one of the four possible comparison results, and is used to compute the address of the next fat node for comparison. This process is repeated until a leaf level is reached. In this way, FAST halves the number of comparisons as each SIMD comparison can advance the indexing traversal by two levels. In addition, the fat nodes in FAST are organized such that each fat node is accommodated in a single cache line (64 bytes), and aligns with the boundary of cache lines as well, which implies a better cache utilization, as all the three keys of a single fat node will be read into cache within only one memory access.

Although fat nodes significantly improve the performance of processing search queries, their static memory layout prohibits the indexing tree from being updated. As a result, FAST entirely rebuilds the binary search tree in order to reflect updates to the tree, which, however, is very costly and significantly hinders the applicability of FAST in real-world scenarios. In addition, fitting each fat node into a single cache line can lead to a waste in memory due to the hole existing in each cache line.

B. Masstree

Masstree [4] is a trie-like structure that supports the indexing of strings of arbitrary length. An example instance of Masstree is shown in Figure 1(b). The fundamental design of Masstree is to partition the string keys into 8-bytes slices and stores each key slice in a single logical node, which is in fact a B^+-tree. Thus, all key slices contained in a B^+-tree of level h share a common prefix of length $8h$. In this way, Masstree translates the expensive string comparison during indexing into fast integer comparison, which is one of the root reasons contributing to its high performance. Masstree also employs several other optimization techniques to boost its performance. First, it prefetches all cache lines of a trie node into cache in parallel, and hence incurs a latency of a single main memory access for each trie node during index traversal. Second, the border (leave) nodes of each B^+-tree are linked together to facilitate range indexing and improve the performance of removing keys. In addition, Masstree optimizes sequential workloads by allocating keys directly into a new node if there is no need to change the



(a) FAST (b) Masstree

(c) BwTree (d) ART (e) PSL

Figure 1: Indices examined in the paper

order of the old keys.

Masstree uses fine-grained latching scheme to allow write operations in different parts of the tree, which can be performed in parallel, and employs optimistic concurrency to eliminate latching for readers. To deal with the contention between readers and writers, the state of each node is recorded in a version counter which shall be marked as "dirty" by writes before switching to another intermediate state, and then incremented when exiting the intermediate state. Upon accessing a node, the reader needs to read the version number beforehand and afterwards; if the version numbers are inconsistent or "dirty", the reader will be aware of that the accessed node is being/has been updated during the two reads of the version number, and retries the process until consistent version numbers are read. However, this strategy can fail in an extreme case where the node being accessed by a reader was split and the target key of the reader has been moved to another node. In this case, retrying at the same node does not make sense even if the version numbers are consistent. To address this case, Masstree requires the writer to update the version of each affected node during the split in a bottom-up manner and the reader to restart from the root whenever version number verification fails.

C. BwTree

BwTree is a latch-free index which has been employed in Hekaton to improve the indexing performance in concurrent scenarios. As can be observed in Figure 1(c), BwTree is a variant of B^+-tree where tree nodes are logical memory pages, and each logical page is associated with a PID, which replaces the pointer in the conventional B^+-tree. A logical page consists of a linked list where each list node represents a certain key range and is associated with the respective PID which is responsible for indexing keys within this range. In order to handle the logical pages, a mapping table is used to maintain the correspondence between logical pages and physical memory addresses of the head node of the composing linked lists. The update to a logical page is implemented in a delta strategy which first prepends a new delta record to the original head node of the linked list of that logical page, and then uses a CAS operation to install the new mapping into the mapping table, during which there is no need to update the tree structure as the PID of the logical page is unchanged. Once a logical page is found to have a long delta chain during indexing process, a consolidation is triggered to compact the deltas in a vector, which is known to be more cache friendly.

For latch-free processing, BwTree links all sibling pages in the same level to perform a two-stage split, and relies only on atomic operations during each stage. The first stage is to conduct a delta update on the splitting node as mentioned above, delete the shifting keys and link to the new sibling page which holds the keys as well as the sibling link of the original node. The second stage is to conduct a delta update on the parent page by adding a link to the new page, and another two-stage split will take place if the number of children of this parent page exceeds the upper limit. An epoch counter is maintained in each logical page to make sure that there is no intermediate parent page during another merging. The merging methods are similar as the splitting steps except that there is a remove mark added in the beginning to prevent the further use of the current page.

D. ART

ART, shown in Figure 1(d), is another trie structure which supports indexing keys of arbitrary length. The main

optimization employed in ART is the use of four types of nodes to represent nodes of various number of children and improve space efficiency and cache awareness. The four types of internal nodes in ART are Node4, Node16, Node48 and Node256, each of which is named after the maximal number of the pointers it can contain. The memory layout of the first three types of nodes is the same, which is a byte array of the respective size representing the characters of the respective level, followed by another pointer array of the same size pointing to the suffix. The last type of nodes, Node256, does not have a byte array since the character corresponding to each pointer is implied by the position of that pointer in the pointer array. The indexing algorithm of ART is the same as that of classic tries, which simply follows the pointer of each matching internal node based on the comparison result. For the last three types of nodes, SIMD processing is used to compare multiple keys contained in the byte array simultaneously. A nice feature of ART is that there is no re-balancing requirement on the tree structure, since small nodes are simply replaced by the large nodes once they cannot find any slots for insertion. In addition, ART further reduces space cost by using path compression and lazy expansion employed in traditional tries.

E. PSL

Being a variant of skip list, PSL [9] also consists of multiple layers of sorted linked lists, which are logically classified into two layers: the storage layer is the bottommost linked list of data nodes, and the upper-level linked lists compose the index layer. 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 vector. 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 downward 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 layer, which means it is time to fetch the real data. Unlike most structures, query processing in PSL is naturally latch-free. In the traversal of the index layers, since the access to each entry is read-only, the use of latches can be avoided. The updates on the bottom level is redistributed among all the threads, each of which handles its own partition and does not interleave with other threads.

PSL also optimizes for the NUMA architecture where accessing local memory is much faster than accessing the memory residing in remote NUMA nodes. Specifically, PSL evenly distributes data nodes among available NUMA nodes such that the key ranges corresponding to the data nodes allocated to each NUMA node are disjoint. Then, an independent instance of PSL will be constructed for each NUMA node from the belonging data nodes. Each incoming query is routed to the corresponding NUMA node, and get processed by the threads spawned in that node. In this manner, there is little remote memory access during query processing, which translates into significantly enhanced query throughput. Moreover, the indices at different NUMA nodes also improve the exploitation of parallelism since they can be used to independently answer queries. PSL further employs a self-adaptive threading mechanism to address the unbalanced workloads between different NUMA nodes.

IV. Design Considerations

In this section, we shall compare the five indices studied in this paper in terms of the optimization techniques applied, the concurrent strategies and the design choices.

A. Optimizations

Due to the huge gap in the latency between main memory accesses and cache accesses, a high cache hit rate is of vital importance to the performance of indexing structures. Consequently, all the five indices employed various optimization techniques to improve cache utilization. FAST compacts three nodes into one "fat" node which can be fit into a single cache line and can be loaded into cache with only one memory access. By minimizing the modification to global data structure, Masstree reduces cache invalidation and hence enables a well cache structure. BwTree consolidates the deltas of lengthy logical page into a vector, which is typically stored in a contiguous memory area, and similarly, ART organizes the pointers of its internal nodes into an array. PSL packs multiple keys in a single index entry, which can also be loaded into cache within a single memory access.

However, these optimizations do not come at no cost. FAST and PSL, by organizing the indexing structure in a compact manner, render it very difficult to make updates to the internal nodes, and hence rely on an entire rebuilding to reflect the updates. Specifically, PSL reserves the ability to process insert queries by simply inserting the key and the associated pointer into the storage layer without updating the index layers even if the new key shall be promoted to an upper level according to the probabilistic model. As a result, the indexing performance of PSL gradually deteriorates as more keys are added to the storage layer until the next rebuilding. For Masstree and BwTree, their ways of improving cache utilization are less aggressive and hence do not compromise the structural flexibility.

SIMD processing has now become a common feature on modern processors. It enables comparison and arithmetical computation to be performed against multiple operands within a single SIMD instruction. With SIMD processing

enabled, FAST, ART and PSL significantly reduce the comparisons of the indexing process, and hence achieve a much higher throughput than the other two non-SIMD counterparts. In order to enable SIMD, the operands normally need to be stored in a contiguous memory area, which is also the reason why these three indices employ a compact memory layout for the storage of internal nodes/entries.

There are also several other optimizations applied to these indices. For example, software prefetching is heavily used in FAST, Masstree and PSL to hide the memory latency. Also, to reduce TLB (Translation Lookaside Buffer) misses, which will lead to a page table walk to locate the memory address during index traversal, some indices, such as Masstree, use huge page technique to reduce the number of entries in the page table and in turn the TLB miss rate.

B. Concurrent Strategies

With the prevalence of multi-core processors, the exploitation of rich parallelism, i.e., scalability, has now been an important performance indicator of modern database indices. Therefore, all the five indices studied in this paper have devoted a lot of effort to this issue. FAST, BwTree and PSL employed latch-free processing, and hence scale well with more cores, as shown in the experiments. The strategy to enforce latch-free processing in FAST and PSL is trivial as the internal nodes/entries of them, once created, are never modified, which removes the need of latches. BwTree employs a complex strategy to handle the splitting and merging of logical pages so that CAS operations can be used in place of latches. ART does not use any concurrency control in the original design since it was proposed for Hyper [29], which has already done the partition for ART. Although Masstree employs latching for write operations, it uses optimistic concurrency control to handle the contention between read operations and write operations, which eliminates the use of latches for guarding readers. In addition, Masstree links the border nodes of each B^+-tree node together with a linked list to facilitate the removal of keys; similarly, BwTree also linkes together all logical pages of the same level for the latch-free processing.

C. Design Tradeoff

Other than the structural difference (tree vs. skip list) between the five indices, there is another fundamental design tradeoff which classifies them into two categories. This design tradeoff lies in the choosing of the data structure for storing the keys and pointers of the internal nodes/entries. Specifically, FAST, ART and PSL belong to the same category by using arrays for keys and pointers, whereas Masstree and BwTree choose linked list. The choice made in this design space represents a tradeoff between performance and structural flexibility. The underlying arrays enable high cache hit rate and SIMD processing, which are the root causes of the high performance of the three indices of the first category. On the other hand, the static requirement on array size reduces the structural flexibility of FAST and PSL. In contrast, Masstree and BwTree do not compromise the structural flexibility by using lists at a cost of lower cache hit rate and hence lower performance.

Table I: Parameter table for experiments

Parameter	Value
Dataset size(M)	2, 4, 8, 16, 32, 64,128, 256, 512, 1024
Batch size	2048, 4096, 8192, 16384, 32768
Number of Threads	1, 2, 4, 8, 16, 32
Write Ratio(%)	0, 20, 40, 60, 80, 100
Zipfian parameter θ	0, 0.5, 0.9, 1.2

V. Performance Evaluation

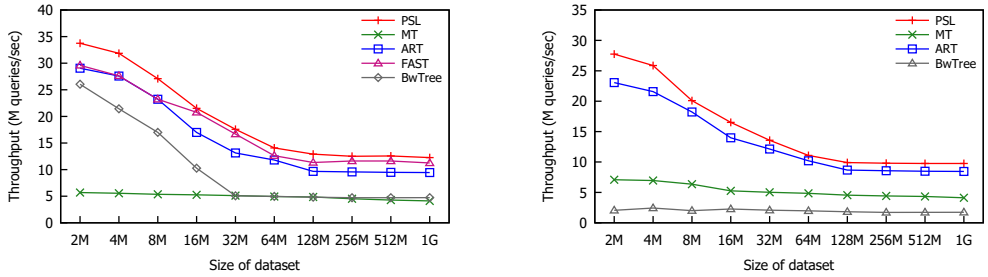
We conduct our experiments on a platform with 512 GB of memory evenly distributed among four NUMA nodes. Each NUMA node is equipped with an Intel Xeon 7540 processor, which supports 128-bit SIMD vectors. The operating system is Ubuntu 12.04 with kernel version 3.8.0-37.

The performance of the five indices is extensively evaluated from various perspectives. First, we measure the performance and the latency of query processing for each index by varying the size of the dataset, and then adjust the number of execution threads to investigate the scalability of each index. Afterwards, we study how these indices perform in the presence of mixed and skewed query workloads. The memory footprint for each index is subsequently measured, and finally we profile the query processing for each index. The Masstree code is obtained from the GitHub [30] repository of its authors. We implement FAST based on [11] and require it to retrieve the data of interest to ensure a fair comparison. We make a simple modification to the published single-threaded ART code [31] by storing the real value in a leaf node instead of the pointer field of an inner node to handle the case where values cannot be held in a pointer. For the multi-threaded version, we implement it in the same way as Hyper [29] does based on author's code. We retrieve BwTree codes [32] from Peloton [33] without any modifications. The rebuilding process of PSL is revoked when the number of newly inserted nodes exceeds 15% of the index size. All the five indices are implemented in C/C++ and compiled with -O3 flag.

All the parameters for the evaluation are summarized in Table I, where the default values are underlined when applicable. The key length is four bytes. The query workload is generated from *Yahoo! Cloud Serving Benchmark* (YCSB) [34], and the keys in the workload follow a Zipfian distribution. We persist the queries into hard disks, and read them into memory for experiments to ensure the same query workload for all the indices.

A. Query Throughput

Figure 2 shows the performance of the five indices under workloads with varying size. Since FAST cannot process



(a) GET Operation

(b) PUT Operation

Figure 2: Query Throughput under different data sets

insert queries, its result is not present in Figure 2(b). For this experiment, we vary the number of keys in the dataset from 2M to 1G, and examine the query throughput for each dataset size. From Figure 2, one can see that both the get and put throughputs of each index initially experience a decrease as the dataset size increases from 2M to 64M, and then become relatively stable for larger dataset sizes. This variation trend is natural. For the dataset with 2M keys, the entire (or the major portion of) index can be accommodated in the last level cache, and the throughput is hence mainly determined by the latency to fetch the entries and data nodes from the cache (compute bound). As the dataset size increases, more and more indexed nodes/entries cannot reside in the cache and hence must be fetched from memory (bandwidth bound), resulting in higher latency and lower query throughput. This also explains why the performance gap between insert and search queries gradually shrinks with the size of dataset.

For each index, the throughput of insert queries is not as high as that of search queries. The reason is two-fold. First, the processing of insert queries is more complicated than that of search queries. Moreover, during the processing of insert queries, the creation of new indexed nodes would compete with original nodes for cache space, which impairs cache structure and hence reduce cache hit rate.

In all cases, FAST, ART and PSL perform much better than the other two, namely BwTree and MassTree. For example, PSL is able to perform 34M search queries or 27M insert queries in one second when the index can be accommodated in the cache (i.e., 2M dataset), which are respectively five and four times more than the corresponding throughput Masstree achieves for the same dataset size. For larger datasets, PSL can still consistently perform at least 1.5x and 1x better than Masstree in terms of the search throughput and insert throughout, respectively. This significant performance gap between the indices of the two categories defined in Section IV-C is due to the different choice made on the data structure for internal nodes/entries. FAST, ART and PSL choose arrays, which, as will be shown in Section V-G, will translate into higher cache utilization, less branch mis-prediction rate and better exploitation of

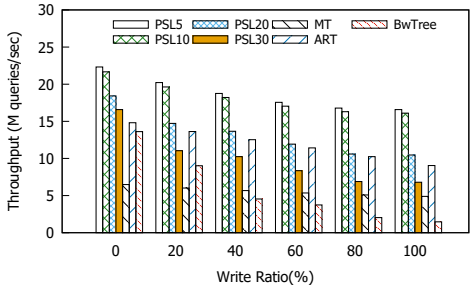


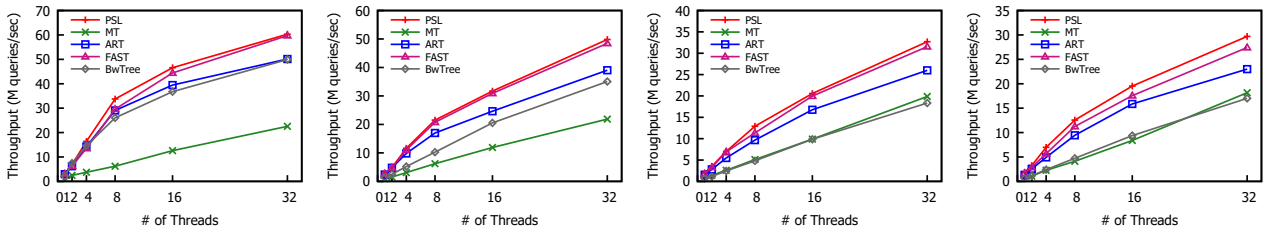
Figure 3: Different update threshold for PSL

processor pipeline. In addition, FAST performs slightly worse than PSL with respect to search query; however, its applicability is restricted due to the inability to process update queries. The throughput of ART is about 20%-35% less than PSL, and this gap shrinks with the size of the dataset, a same observation that can be drawn from the comparison between PSL and Masstree (or BwTree).

Although PSL achieves a higher query throughput than the others, its performance, however, is highly dependent on its rebuilding strategy to reflect new data nodes in the index layer. Therefore, we perform an experiment to study the sensitiveness of PSL's performance to the rebuilding frequency by changing the threshold number of new data nodes which triggers the rebuilding process. The 16M dataset is used for this experiment. Figure 3 shows the impact of rebuilding on the performance of PSL, where the number following PSL in the legends means the percentage of newly inserted data nodes for triggering rebuilding. As can be seen from this figure, as the rebuilding process is triggered less frequently, the performance of PSL gradually deteriorates as a result of increasing number of data nodes to visit, and becomes worse than ART when the rebuilding is triggered only after 30% of new nodes are added.

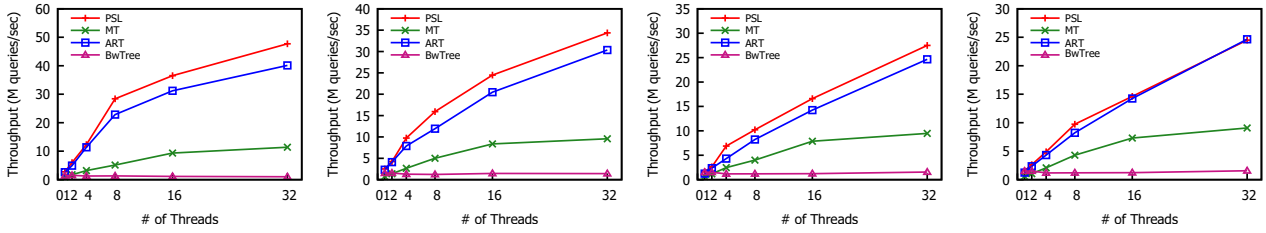
B. Latency

We investigate the latency of each index in this section. The experiments are conducted under four data sets, and each index performs a batch of get queries, since ART and PSL employ a batch processing technique and FAST does not support put queries. The latency result is depicted in



(a) dataset size = 2M, GET (b) dataset size = 16M, GET (c) dataset size = 128M, GET (d) dataset size = 1G, GET

Figure 4: Scalability on GET operation.



(a) dataset size = 2M, PUT (b) dataset size = 16M, PUT (c) dataset size = 128M, PUT (d) dataset size = 1G, PUT

Figure 5: Scalability on PUT operation.

Table II: Query latency

Dataset	MassTree	FAST	ART	PSL	BwTree
Small(2M)	13.2ms	2.88ms	2.89ms	2.35ms	2.35ms
Medium(16M)	14.7ms	4.7ms	4.98ms	4.01ms	8.16ms
Large(128M)	15.82ms	6.74ms	6.22ms	6.21ms	15.4ms
Huge(1G)	17.01ms	8.19ms	8.02ms	7.97ms	19.11ms

Table II. As can be observed in the table, FAST, PSL and ART compose of the first tier among the five indices for their much smaller latency than the other two. Masstree, which often ranks last, needs a time of 2-4x longer than the first tier indices to process a query. BwTree's latency equals to that of the first-tier indices for the small dataset, but becomes significantly higher for the huge dataset. This can be attributed to the delta lists composing the logical pages of BwTree which consists of nodes that distributed randomly in the physical memory. In the case of the small dataset, since the entire index can be loaded into the cache, the latency gap between BwTree and the first-tier indices diminishes. However, for the huge dataset, the delta lists of logical pages result in many memory accesses, and hence significantly increase the latency.

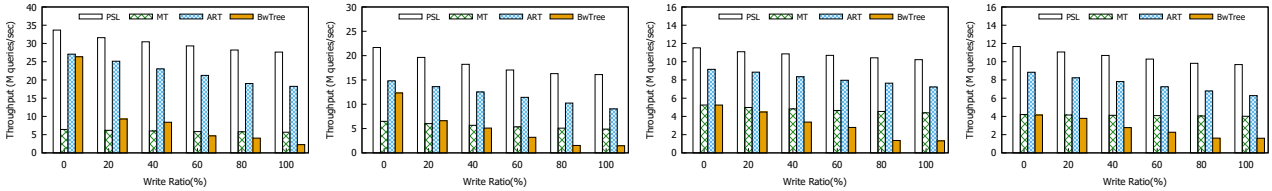
C. Scalability

Figure 4 and Figure 5 show how the query throughputs of the five indices vary with the number of execution threads. As shown in Figure 4, all five indices can get their query throughput to increase with more computing resources under the read-only workloads, but the increasing rate differs. PSL, FAST and ART, which employ the similar optimization and choose the same data structure for the storage of internal indexing nodes/entries, exhibit a similar trend in the increasing rate in all get workloads, and achieve the best scalability as demonstrated by the substantial gap in the increasing rate between them and the other three. Interestingly, the scalability of BwTree is initially comparable with that of PSL, FAST and ART for the small dataset, but gradually converges to that of the Masstree as the dataset size becomes larger. We infer that this is also because of the poor cache utilization of delta list nodes composing the logical pages.

The scalability for put queries is almost the same as that of get queries for all indices other than BwTree, whose performance cannot be improved by using more threads. This is probably because as deltas are constantly updated, the delta lists becomes longer and longer and quickly make the processing of put queries dominated by memory accesses, which in turn eliminates the impact of more computing threads. For Masstree, the performance does not get much improved when the number of thread increases from 16 to 32. This is probably because of a similar reason to BwTree: the performance of Masstree is dominated by the memory accesses, and using more computing resource does not help.

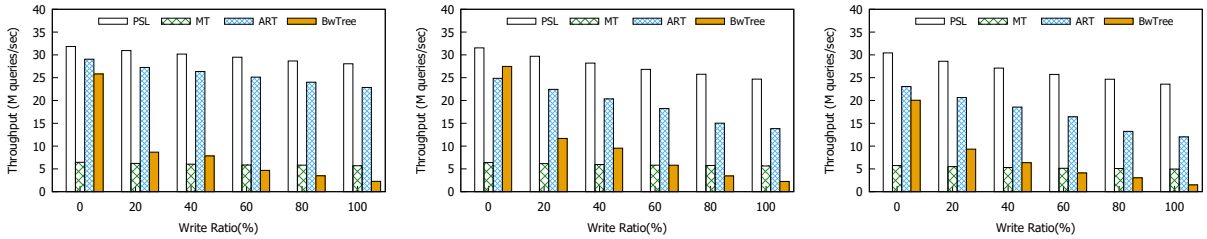
D. Mixed Workload

We examine the performance of each index in terms of mixed workload in this section. The motivation of this test is that we want to thoroughly profile the performance for each index from the read-only workload to write-only workload. The four default datasets are used during the experiment and the number of threads is 8. The ratio of write operations (insertion and update) to the whole query set is set as 0, 20%, 40%, 60%, 80% and 100%, respectively. The keys



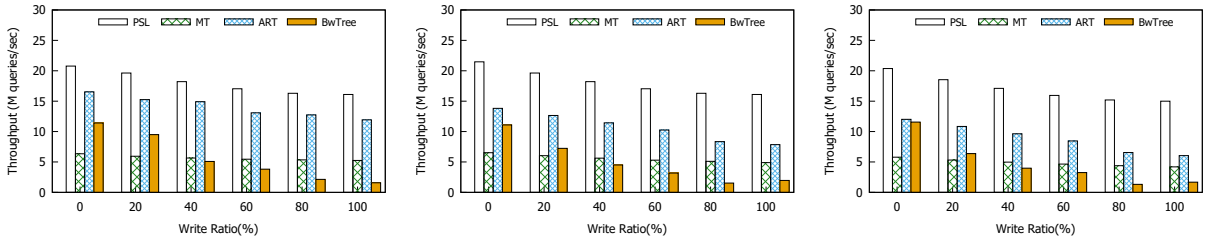
(a) Dataset size = 2M (b) Dataset size = 16M (c) Dataset size = 128M (d) Dataset size = 1G

Figure 6: Performance on mixed workloads



(a) $\theta = 0.5$ (b) $\theta = 0.9$ (c) $\theta = 1.2$

Figure 7: Performance on skewed workloads (2M)



(a) $\theta = 0.5$ (b) $\theta = 0.9$ (c) $\theta = 1.2$

Figure 8: Performance on skewed workloads (16M)

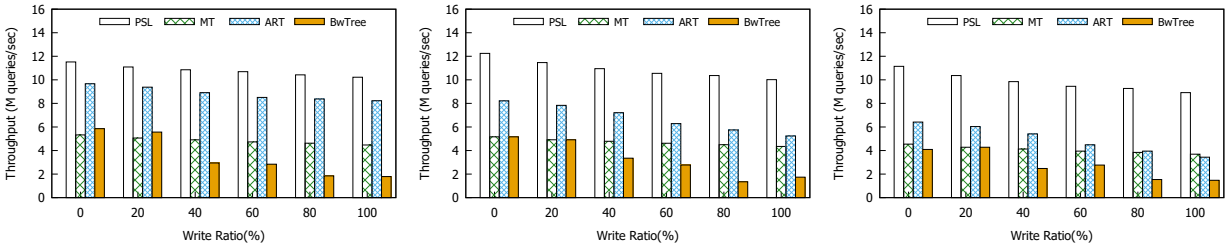
follow a uniform distribution and FAST is omitted in this section due to its inability to make changes to the index.

As can be observed from Figure 6, with more updates, the throughput of the four indices decreases due to the fact that more keys have to be compared before accessing the data nodes, and some of the indices start to rebalance their internal structures. ART and PSL both experience a 15%-20% performance penalty for each 20% increase in the write ratio. In contrast, the performance degradation of BwTree is much more significant. For the two small datasets (2M and 16M), the initial 20% increase in the write ratio leads to 50% reduction in the throughput of BwTree. This can also be attributed to the long delta list traversal after delta updates. Further increasing the write ratio continues to lengthen the delta list, and hence slow down the processing of BwTree. Masstree, however, keeps a stable performance under all the mixed workloads since its processing of get requests and put requests are both dominated by memory accesses, which have roughly the same number in both cases.

E. Skewness

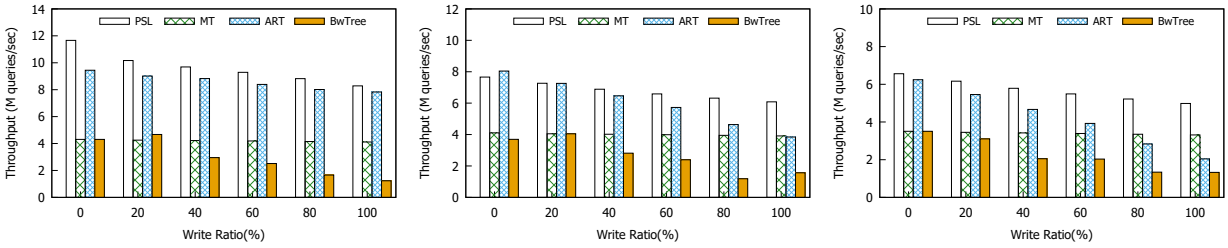
In this section, the performance of query processing is evaluated in the presence of query skew. There are eight threads running over the four NUMA nodes, and four datasets with default sizes are used. The skew in query workload is realized via varying the probability parameter, θ, of Zipfian distribution for workload generation. For comparison and completeness, the workloads for this section is mixed in the same manner as the last section. The multi-threaded implementation of ART is to assign each partition to a thread for processing, as done in Hyper and can be easily adapted to the original implementation. We do not apply the adaptive threading mechanism of PSL to ART as it is unable to assign more threads to some certain partitions. Figure 7 to Figure 10 demonstrate the variation in the throughput query processing with respect to query skew and write ratio in query workloads.

By comparing the four figures, we can observe that query skew has the least impact on Masstree and BwTree. PSL is also seldom affected by the skewed workload compared to the other three indices, which can be attributed to the adaptive threading mechanism that dynamically allocates computing resources among NUMA nodes based on the query load on each node. In contrast, query skewness significantly affects ART in terms of query throughput. This



(a) $\theta = 0.5$ (b) $\theta = 0.9$ (c) $\theta = 1.2$

Figure 9: Performance on skewed workloads (128M)



(a) $\theta = 0.5$ (b) $\theta = 0.9$ (c) $\theta = 1.2$

Figure 10: Performance on skewed workloads (1G)

is natural since there is no load balancing optimization. In the worst case with a write-only workload issued to the huge data set, the performance drops by two thirds when θ varies from 0 in Figure 6(a)[1] to 1.2 in Figure 7(c). However, ART still can achieve a relatively high throughput compared to Masstree and Bwtree, especially in small datasets.

We also examine the case with a high skewness in dataset instead of in workload. The dataset used for this case has 16M keys and most of the keys are within a limited subset of key space. Figure 11 shows the single-threaded performance of the five indices for 16M dataset. It can be found that ART can achieve a better performance than PSL. This is because, due to the skewness in dataset, most of the indexed key of ART have a long common prefix, implying an improved cache utilization.

F. Memory Consumption

In this section, we measure the space consumption for the five indices. We use massif from valgrind toolkit to accumulate the consuming memory spaces as well as track the memory footprints for all the five indices. The results are shown in Figure 12.

As can be seen in the figure, the most efficient index in terms of space cost is unsurprisingly FAST since there is totally no reserved space within FAST and the only extra space needed for FAST is the look-up table. Among the rest four indices, ART consumes the least memory by using four types of nodes with different size to adapt to the distribution in the number of suffixes. Although there is no unused space in PSL, the extra index instance created by the rebuilding process would incur a substantial memory usage. The most expensive index in terms of space cost is BwTree, which maintains a delta list for each logical page and an additional mapping table for addressing logical pages. Although consolidation is employed to compact the storage, this, however, does not improve the storage efficiency much since the consolidation process is revoked only for lengthy delta chains, which in most cases only account for a small fraction of the total logical pages.

[1]When $\theta = 0$, the YCSB workload degenerates into the uniform workload, which is the case that we used for other experiments.

G. Profiling Query Processing

We also profile the query processing of the five indices from multiple aspects, including the total number of instructions, instruction per cycle (IPC), branch mis-prediction rate and cache miss rate. The first two metrics are collected via perf, and the other two are measured via cachegrind of the valgrind toolkit. For this experiment, we run a read-only workload which issues roughly 10% of the keys indexed to the index. We only collect the results for the first three datasets, which are respectively depicted in Table III, IV and V. For the 1G dataset, we could not get the result as it took such a long time for valgrind to complete the collection.

It can be seen from the three tables that the high performance of PSL, ART and FAST does not come from the number of instructions executed, which, on the contrary, is actually much larger than the number of instructions generated by the other two indices with a less query throughput, as shown in previous experiments. Instead, their advantageous performance is attributed to the (2-4x) less exploitation of CPU pipeline, the (5-10x) more accurate branch prediction and the (2-6x) better cache miss rate. The huge gap in these

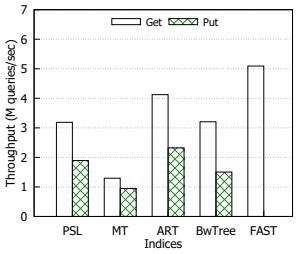


Figure 11: Query throughput under skewed dataset

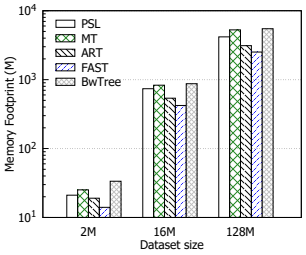


Figure 12: Memory Consumption

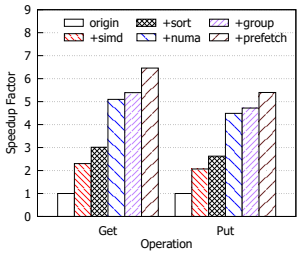


Figure 13: Breakdown of performance boost in PSL

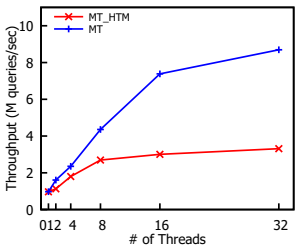


Figure 14: HTM Masstree vs. Masstree

Table III: Query Profiling (2M)

	#Instructions	IPC	Branch-miss (%)	Cache-miss(%)
PSL	5.9B	2.08	1.09	0
ART	5.9B	2.05	0.82	0
FAST	4.9B	1.8	0.91	0
Masstree	3.4B	2.04	7.10	0.20
BwTree	4.0B	1.16	2.04	0.30

Table IV: Query Profiling (16M)

	#Instructions	IPC	Branch-miss (%)	Cache-miss(%)
PSL	47.3B	1.98	1.12	0.17
ART	47.5B	2.04	0.85	0.31
FAST	37B	1.58	0.59	0.64
Masstree	25.9B	0.57	7.18	0.80
BwTree	33.6B	0.71	3.43	1.10

Table V: Query Profiling (128M)

	#Instructions	IPC	Branch-miss (%)	Cache-miss(%)
PSL	378.6B	1.94	1.11	0.49
ART	397.0B	1.85	1.36	0.80
FAST	351B	1.36	0.59	0.64
Masstree	205.7B	0.47	7.10	2.90
BwTree	315.2B	0.51	4.65	3.00

three performance metrics is caused by the design decision on the data structure for the storage of internal nodes/entries, which, as we show in Section IV-C, partitions the five indices into two categories: PSL, FAST and ART use arrays to enable SIMD processing and enhance cache utilization, and the other two choose lists for structural flexibility.

H. Optimization Impact

In order to better understand the impact of different optimizations on the performance of indexing, we study the performance improvement caused by each optimization technique employed in PSL, which achieves the best performance in most experimental settings. The result is shown in Figure 13. By grouping the keys appearing at the index layer to entries, and leveraging SIMD processing to reduce comparisons and hence memory/cache accesses, the throughput of PSL can be improved by 1.3x and 1x for search and insert queries, respectively. Sorting queries in a batch further improves the performance by 0.3x. NUMA-aware operation leads to another huge performance gain, improving the query throughput by 1.2x. This performance gain is because the NUMA-aware optimization almost eliminates the remote memory access, which is several times slower than accessing local memory. Group query processing results in a slight improvement of 0.05x for both types of queries, and prefetching contributes to the final performance gain of 0.2x and 0.14x for search and insert queries, respectively.

As discussed in Section II, there are plenty of indices which enable HTM to achieve high throughput in concurrent settings. As a complement, we implement a coarse HTM version of Masstree by following the idea of [15], and compare it with the original Masstree to explore the impact of HTM. For this experiment, we use another server equipped with an Intel Xeon CPU E7-4820 CPU and 128 GB main memory, as the original server does not support HTM. The 16MB dataset is used and θ is set to 0.9. The results for the put workload are shown in Figure 14. As can be observed, the coarse HTM implementation counter-intuitively harms the overall throughput by a factor which ranges from 0.1x to 3x. This is because the critical section of Masstree protected by the HTM region often faces concurrent exclusive accesses, leading to frequent aborts and retries, and hence impairing the performance. Therefore, without careful design and engineering, using HTM does not always pay off.

VI. Conclusions and Future Work

In this paper, we conduct a comprehensive performance study on five state-of-the-art indices, Masstree, BwTree, FAST, ART and PSL, with respect to throughput, scalability, latency, memory consumption and cache miss rate. According to our results, PSL achieves a similar query performance in terms of get throughput to a read-only index, FAST, and meanwhile performs up to 5x, 0.3x and 2.5x faster than Masstree, ART and BwTree respectively. Another interesting point observed in the results is that the performance of BwTree drops significantly when the workload is write-only, probably due to the contenction caused by the frequent fails on the hot blocks. FAST and ART win the competition on the cost of memory space, leading other indices with a saving factor between 20%-100%. Our experiments also reveal that for the dense dataset, ART and BwTree, which belong to indices built based on tries, may be the suitable candidates for a better overall performance.

We conclude that all the five indices have their own advantages and there is no best index choice in in-memory

databases. The attribute of the dataset and the workload may help choosing the appropriate indices. However, a smart strategy on concurrent synchronization and exploitation on hardware are needed for modern in-memory indices. For the future work, a comparison of range query performance in concurrent settings is needed. In addition, considering the recent advance in hash table and key-value stores, a comparison between them and the tree and skip list based indices should be a perfect complimentary to this work.

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). Gang Chen's work was supported by the National Basic Research Program (973 Program, No.2015CB352400). Rui Mao's work was supported by Guangdong Provincial General University National Development Program (2014GKXM054).

REFERENCES

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

[2] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey," *TKDE*, vol. 27, no. 7, pp. 1920–1948, 2015.

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

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

[5] J. Rao and K. A. Ross, "Making B^+-trees Cache Conscious in Main Memory," in *SIGMOD*, 2000, pp. 475–486.

[6] S. Chen, P. B. Gibbons, and T. C. Mowry, "Improving Index Performance through Prefetching," in *SIGMOD*, 2001, pp. 235–246.

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

[8] 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.

[9] Z. Xie, Q. Cai, H. Jagadish, B. C. Ooi, and W.-F. Wong, "Parallelizing skip lists for in-memory multi-core database systems," in *ICDE*, 2017, pp. 119–122.

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

[11] 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.

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

[13] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, "Reducing the storage overhead of main-memory oltp databases with hybrid indexes," in *SIGMOD*, 2016.

[14] V. Leis, A. Kemper, and T. Neumann, "Exploiting hardware transactional memory in main-memory databases," in *ICDE*, 2014, pp. 580–591.

[15] X. Wang, W. Zhang, Z. Wang, Z. Wei, H. Chen, and W. Zhao, "Eunomia: Scaling concurrent search trees under contention using htm," in *PPoPP*, 2017, pp. 385–399.

[16] T. Brown, F. Ellen, and E. Ruppert, "A general technique for non-blocking trees," in *ACM SIGPLAN Notices*, vol. 49, no. 8, 2014, pp. 329–342.

[17] A. Braginsky and E. Petrank, "A lock-free b+ tree," in *SPAA*, 2012, pp. 58–67.

[18] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: SQL Server's Memory-Optimized OLTP Engine," in *SIGMOD*, 2013, pp. 1243–1254.

[19] J. Levandoski, D. Lomet, S. Sengupta, A. Birka, and C. Diaconu, "Indexing on Modern Hardware: Hekaton and beyond," in *SIGMOD*, 2014, pp. 717–720.

[20] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki, "PLP: page latch-free shared-everything OLTP," *PVLDB*, vol. 4, no. 10, pp. 610–621, 2011.

[21] "The Story Behind MemSQL's Skiplist Indexes," http://blog.memsql.com/the-story-behind-memsqls-skiplist-indexes/.

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

[23] T. Crain, V. Gramoli, and M. Raynal, "No Hot Spot Non-Blocking Skip List," in *ICDCS*, 2013, pp. 196–205.

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

[25] S. Sprenger, S. Zeuch, and U. Leser, "Cache-Sensitive Skip List: Efficient Range Queries on modern CPUs."

[26] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture," *PVLDB*, vol. 1, no. 2, pp. 1313–1324, 2008.

[27] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu, "Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware," in *ICDE*, 2013, pp. 362–373.

[28] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, "Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited," *PVLDB*, vol. 7, no. 1, pp. 85–96, 2013.

[29] A. Kemper and T. Neumann, "HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots," in *ICDE*, 2011, pp. 195–206.

[30] "Masstree source," https://github.com/kohler/masstree-beta.

[31] "Adaptive Radix Tree source," https://www-db.in.tum.de/~leis/.

[32] "BwTree from Peloton," https://github.com/wangziqi2013/BwTree.

[33] "Peloton System," https://github.com/cmu-db/peloton.

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