Empirical Evaluation of Indexing on Modern Database Systems

Shawn Zhong
University of Wisconsin - Madison
shawn.zhong@wisc.edu

Yusen Liu
University of Wisconsin - Madison
liu797@wisc.edu

Libin Zhou
University of Wisconsin - Madison
lzhou228@wisc.edu

ABSTRACT

For the past several decades, a wide variety of indexing techniques and optimizations have been proposed and implemented in both academics and industry, but there is a lack of comprehensive study of the influence of indexing techniques over a wide variety of workloads and configurations. In this paper, we surveyed several indexing techniques used in database management systems, which fall into three categories: tree-based index, hash index, and learned index. We implemented B+ Tree index on DBx1000, and evaluated its performance along with hash index under different scenarios. We investigated how both index techniques perform under different concurrency control algorithms, contention levels, and read-write ratio, whether one of the indexes is more scalable, how the latch adds overhead to indexing, and how fanout affects the performance of B+ Tree.

PVLDB Reference Format:
Shawn Zhong, Yusen Liu, and Libin Zhou. Empirical Evaluation of Indexing on Modern Database Systems. DAWN 2020.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/YSL-1997/DBx1000.

1 INTRODUCTION

Indexing is widely used in almost all DBMS [11], and a huge variety of indexing techniques and optimizations have been proposed in past decades [1, 19, 20, 22, 24, 25]. However, few people investigate the performance comparison for indexing techniques, which becomes the motivation for us to make it real. With the aim to better evaluate the performance of various indexing algorithms, in this paper, we divided the work into two parts, survey, and evaluation.

The first part is a survey of several major indexing techniques and some optimizations proposed in the past few decades. We categorize the indexes based on the baseline algorithm used in the implementation, namely the employed data structure or the unique technologies. Therefore we divide the first part into 3 categories, namely tree index, hash index, and learned index. For tree-based index, we present several variants of B+ Tree including B-link tree, Palm tree, and Masstree, which improve the performance of B+ Tree in different ways. Through our survey, we notice that the majority of indexes proposed are tree-based. Moreover, most of the indexing techniques proposed in the last decade are optimizations based on a few indexing techniques such as B+ Tree, Log Structured Merge Tree (LSM tree), and hash index.

Therefore, in the second part of this project, we choose the hash index and the vanilla B+ Tree index as our experimental subjects. We conducted our experiments based on DBx1000 database management system [26] which is a multi-core in-memory database simulator. We chose DBx1000 because it eliminates the unnecessary bottlenecks other than concurrency control algorithms [26]. We tried to implement the DBMS such that the only bottlenecks other than preexisting concurrency control bottlenecks are from indexing. We implemented the B+ Tree indexing over DBx1000 and designed several experiments to provide a comprehensive evaluation for both indexes. We tested their compatibility with different concurrency control algorithms, scalability, performance under different contention levels. We measured the sensibility to read-write ratio, how they leverage cache locality, and the overhead imposed by the latch.

Our analysis shows under our current benchmark settings and B+ Tree implementation, Hash Index always has better performance than B+ Tree indexes. The only exception is **HEKATON** under TPC-C where hash index and B+ Tree show almost identical scalability. We show that for DBMS with under 32 threads, hash indexes scale to almost the same level for every concurrency control algorithm we chose. Thus we conclude that almost all concurrency control algorithms can work well with the hash index. We also present the lack of scalability for B+ Tree index. We have to admit that our benchmarks are built for hash indexes; therefore many advantages of B+ Tree index, such as efficient range queries, cannot be shown here. Our B+ Tree implementation may also lack some optimizations, and can be improved in future work. Our contention level experiment also does not show a clear distinction or pattern for each index.

This paper is structured as follows. Section 2 shows the background and motivation of this paper. Section 3 reviews indexing techniques used in modern database systems. Section 4 introduces DBx1000 and our B+ Tree implementation based on it. Section 5 shows the evaluation results of current implementation. We conclude our paper in Section 6.

2 BACKGROUND AND MOTIVATION

For the past several decades, a wide variety of indexing techniques and optimizations have been proposed and implemented in academics and industry. After the idea of locking protocol [2] had been introduced, database researchers tended to figure out a way of reducing the overhead of latch in a multi-user environment. The most distinguished idea was the efficient lock [16] presented by Lehman. However, it did not have good scalability in a multi-core environment, and the performance of sequential modification was not satisfying. To completely get rid of the latch overhead, Palm tree was introduced, which not only was a latch-free scheme but also built a connection between threads and let them cooperate with each other. This scheme allowed to take advantage of the increasing cores and bandwidth of hardware and left the user to

determine the structure of the tree-based index, which provided an open way of optimization.

Hash index is a completely different way of indexing compared to a tree-based index, in which, the choice of the hash function and collision resolution are the two critical aspects that determine the performance of a hash index. Hence, the cost of the hash index is equal to the sum of the cost of the two aspects.

The learned index is a newly introduced concept that replaces traditional index with a light-weight machine learning model (e.g., linear regression tree) for sake of smaller index size and faster lookup time.

Indexing serves as an important element of modern-day DBMS, but currently, there is a lack of comprehensive study of the influence of indexing techniques over a variety of workloads and configurations. Thus we believe it will be beneficial to conduct an experiment to compare the performance of different indexes with different configurations.

3 INDEXING METHODS

3.1 Tree-Based Index

B-Tree was originally proposed in the 1970s and it has evolved over the past several decades [1, 11], which is an m-ary tree that is widely used for efficient retrieval of a large amount of information in a blocked-oriented storage context[9] with theoretical time complexity $O(\log(N))$. With more and more users performing operations on databases including insertions and deletions, it is required to ensure all the operations be carried out simultaneously without being intervened by other operations. The problem of concurrent access to B-trees was examined with the aim of determining whether B-trees can be used as indexing in a multi-user environment and a deadlock-free solution was presented using simple locking protocols [2]. In this section, we present several variants of B+ Tree that improve the performance from various points of view.

3.1.1 B-Link Tree. With the locking protocols, we could take advantage of the efficiency of the B+ tree and construct a database that supports concurrent access. The first problem is to reduce the intrinsic latch overhead of concurrent operations on B+ tree storage model [16]. In order to improve the performance of concurrent manipulation of the data, the B-link tree was presented, in which, the introduction of a single additional "link" pointer in each node enabled the process to recover from tree modifications performed by other concurrent processes [16]. This leads to a newly designed deadlock-free algorithm and a reduction of the number of locks needed by any process with at most three locks acquired during an insertion.

3.1.2 Palm Tree. B-link tree index does not have good scalability for multi-core environment since it relies on latching schemes, which can lead to the result that high overhead of contention for locks limits the performance of concurrently updating and searching queries [25]. As the number of cores increases, the cost of barrier synchronization increases as well [23]. With the trend that there are more cores and increasing bandwidth in future hardware, a scheme that supports a big amount of concurrent updates and searches is required [25]. Therefore, Palm tree - a latch-free scheme was developed. The use of pre-sort and point-to-point communication between threads strictly contributes to the scalability of Palm tree in multi-core hardware[25].

Compared to conventional concurrency algorithms, Palm tree builds a connection between threads and lets them cooperate without using latch. The overhead for this scheme is the search cost for each query to redistribute work. After redistributing work for each thread, each thread will be responsible for modifying nodes that have better spatial locality, which will lead to better cache locality with less cache miss. Moreover, the Palm Tree algorithm does not impose any constraints on the B+ tree structure itself, and the structure of the B+ tree can be freely selected according to the needs, which provides us an open way of optimization. For example, it can be combined with other B+ tree optimizations, such as prefix B+ tree, node prefetch [4], prefix compression, etc.

3.1.3 Masstree. B+ tree index is more suitable for range queries than hash index since B+ tree supports sequential scan. The height of the B+ tree is kept small when each node has a high fanout. The consequence is each node has more keys than that of a B+ tree with a lower fanout. When performing a sequential scan, the cost of key comparisons would be high which could be worse for variable-length keys [19]. One common data structure used to store variable-length keys is Trie [5]. Trie is a search tree, with descendants of each internal node having the same prefix of the string that corresponds to the internal node [10].

With the aim to support fast range queries, Mass tree was introduced [19]. The design of a Mass tree could be regarded as a divide-and-conquer approach, which first solved the problem of the high cost of variable-length key comparisons with Trie, and then improved the search performance inside each Trie node using B+ tree index. Hence, Masstree is a Trie with B+ trees as nodes.

Consider the case when there are n keys with the maximum length of the key to be l. Masstree will have $O(l)$ layers. When we perform a search operation, Masstree will make $O(\log n)$ comparisons in each layer, with each comparison cost of $O(1)$ since the length of the key is fixed. Hence, the total cost for Masstree is $O(l \log n)$. The cost of searching a key using B+ tree is $O(l \log n)$. Masstree outperforms the B+ tree with $O(l + \log n)$ when the keys have common prefixes. For the worst-case on range queries, Masstree has higher complexity than B+ tree because of the overhead of traversal of multiple layers [19].

3.1.4 Log-Structured Merge Tree. High-performance transaction systems typically need to insert rows continuously in the history table and insert logs for recovery [20]. Thus Log-structured Merge Tree (LSM) is proposed as a disk-based low-cost indexing algorithm. LSM contains two major components. The first component lies completely within the memory. As the new row entry is generated, the system first writes it to log as normal writing-ahead logging. Then the entry is added to the in-memory tree structure with no I/O costs. But maintaining data in memory is by itself expensive [20]. When the data in memory reaches a certain threshold, the contents will be written in batch to the disk sequentially at the end of the existing data on disk. The contents on the disk are never overwritten, and periodically the data on disk will be merged in a compaction phase (rolling merge phase in [20]). LSM creates indexes for on disk data for faster access. There exist several LSM-based indexes such as R-Tree[13], bLSM[24] and SLSM[22] which

are optimizations of the original LSM tree. For example, the LSM-based R-Tree contains an in-memory component consisting of an R-Tree and a deleted-key B-Tree. On the other side, the bLSM tree is a three-level with two levels on disk protected by bloom filters. It leverages a Gear Scheduler to ensure all merge processes happen simultaneously.

3.2 Hash Index

Hash table is another widely used technique in database indexing, with ideally $O(1)$ of key-lookup but most hash table designs have worse than $O(1)$ performance because of the conflict of hash values. The performance of the hash index depends on how we deal with conflicts and the number of conflicts, which largely depends on the choice of hash functions. The goal is to design a hash table such that the average cost for each lookup is independent of the number of items stored in the table. In this section, we present improvements based on the hash index from two aspects, the choice of the hash function and the way of dealing with conflicts.

3.2.1 Input Independent Average Linear Time Algorithm. The core idea of this algorithm is to make a random choice of hash function from a suitable class of hash functions called universal class[3]. If the class of functions is chosen properly, then the average performance of the hash table on any input will be comparable to the performance of a single function constructed with knowledge of the input.

3.2.2 Separate Chaining. Separate chaining is a way of dealing with conflicts. The data structures consist of a number of buckets with each bucked corresponds to a list of entries that have the same hash value. The time cost of searching for a value is the time to find the bucket $O(1)$ and the time to search inside the list, which can be optimized using various techniques. The worst-case cost of searching inside the list is $O(n)$, which leads to a total worst-case cost of $O(n)$.

3.2.3 Open Addressing. Instead of storing the entries with the same hash value in a new data structure, open addressing presents a way of storing all the entries in the bucket array. The most widely used well-known method is linear probing [21] since it takes advantage of CPU cache. Cuckoo hashing is an alternative approach, with two or more hash functions used. The insertion of a key may require more than two of the set of candidate hash functions since if there is a collision, another unused hash function would be used. This iterative step will not stop until either no collision happens or all the hash functions are used. The worst-case cost of searching is $O(m)$ where m represents the number of candidate hash functions.

3.2.4 MICA. MICA is a scalable in-memory key-value store that supports intensive write, in which the indexing technique is mainly based on hash index but with small modifications [18]. Circular log is a memory allocator for key-value items in MICA. Lossy concurrent hash index is used in order to find items stored in the circular log quickly. Key-value items are stored in multiple buckets indexed by the hash of the key, and each bucket has multiple entries. The index entry for this key can occupy any entry of the same bucket. Since hash index needs to deal with collisions, which may cost a longer time, MICA evicts the oldest entry from the set-associative table to accommodate a new index entry when a bucket is full. This allows a very fast index of a new key-value item, and that's the core reason that MICA achieves such high write speed. But this design does not support dynamically keeping track of where the cache misses since the size of the hash table is fixed.

3.3 Learned Index

Kraska et al. first proposed the idea of the learned index to replace the index in the database with a model in machine learning[14]. They view B-Tree index and Hash Index as models that map a key to the position of a record in a sorted or unsorted array respectively, then showed that a learned model can outperform the traditional index techniques in many aspects.

Range index (e.g, B-Tree) can be approximated by a cumulative distribution function (CDF) modeled by a linear regression tree so that the time complexity for lookup becomes $O(1)$. They also proposed the recursive regression model consisting of multiple "stages". The idea of stages is similar to levels in a tree structure, but the nodes are replaced with models that output values to find the child model in the lower layer. The evaluation results show an order of magnitude improvements for time and space.

For point index (e.g., Hash Index), it's possible to replace the hash function with a model to reduce the collisions. A CDF distribution is learned to map the key into a uniformly distributed space for different buckets.

Existence index (e.g., Bloom filter) can be viewed as a binary probabilistic classification task. They propose to use a model at the front of a Bloom filter, so that if the model outputs "maybe no", the key will be checked again by the Bloom filter. This reduces the size of the original Bloom filter since the new one just needs to deal with the false negatives from the learned model.

4 DBx1000

In this project, we benchmarked the B+ Tree index and hash index under different workloads over DBX1000 database [26]. DBx1000 is an in-memory multicore DBMS that implements several concurrency control protocols. It can scale up to 1000 cores using Graphite CPU simulator and serves as a great platform to test the scalability. DBx1000 inherently supports the hash index; thus we have to implement other indexing techniques for the project. It only implements the necessary parts, thus it eliminates most of the irrelevant features of other design decisions. In this project, we realize besides the indexes we are testing, the major independent variables are the built-in concurrency control algorithms and we decide to examine if certain concurrency control algorithms fit better with certain indexes.

4.1 Concurrency Control Algorithms

DBx1000 implemented 11 concurrency control algorithms, and we picked the 5 of them in our evaluation: **`DL_DETECT`**, **`NO_WAIT`**, **`SILO`**, **`HEKATON`**, and **`TICTOC`**.

There are two major categories of techniques to handle deadlock in DBMS, namely deadlocking detection and deadlock avoidance. The **`DL_DETECT`** is the 2 phase locking (2PL) with a central monitor of wait-for graph. When a deadlock is detected, the monitor will choose the transaction which used fewer resources to abort [26]. It

represents the deadlock detection techniques employed in DBMS. The other 4 algorithms are all deadlock prevention. **NO_WAIT** is a non-preemptive 2PL. When a transaction failed to acquire a lock, it immediately releases all its locks and aborts. It represents a simple baseline pessimistic concurrency control algorithm. **SILO** is the state of art optimistic concurrency control algorithm (OCC) [26] and we use it to represent the general OCC algorithm. **HEKATON** is the modern database engine integrated into SQL Server and it leverages an optimistic multi-version concurrency control algorithm [7, 15]. Finally, we chose **TICTOC** which is an optimized version of OCC with a better timestamp allocation protocol [27]. We wish these 5 algorithms can represent the majority of the state of art techniques used in modern DBMS.

4.2 B+ Tree Implementation

We implemented the B+ Tree index for DBx1000 DBMS in 782 LOC. We wrote the basic functionalities of B+ tree with efficient searching, insertion, and also made sure that B+ Tree works concurrently with latch enabled. Since hash index does not support range query, we didn't include such queries in our benchmark, and thus didn't implement range query for B+ Tree. Our B+ Tree implementation may lack some optimizations and can be improved in future work.

The source code is available at https://github.com/ShawnZhong/DBx1000.

5 EVALUATION

We ran all our experiments on a single-node CloudLab[8] c8220 machine with two Intel Intel E5-2660 10-core CPUs, 256 GB ECC Memory, and two 1 TB SATA HDDs. The machine has two NUMA nodes, each has 25 MB L3 cache shared across 10 cores (20 threads). Each core has 64 KB L1 cache (32 KB L1i + 32 KB L1d), and 256 KB L2 cache.

For evaluation, we seek to answer the follow questions:

- How does both index techniques behave under different concurrency control algorithms (Section 5.2)
- Which index technique is more scalable under different number of threads (Section 5.3)
- How does different contention levels affect the performance of indexing (Section 5.4).
- Whether an index technique is more suitable for read or write workload (Section 5.5)
- How sensitive indexing techniques are to the cache locality (Section 5.6)
- How much overhead does latching impose under single-threaded scenario (Section 5.7)
- For B+ Tree index, how does the fanout affect the performance (Section 5.8)

For all the experiments, we issue 100,000 transactions and report the average time spent per transaction per thread. If not mentioned otherwise, we use the configuration shown in Table 1 as the default configuration.

All the evaluation data has been made available at https://github.com/ShawnZhong/DBx1000/tree/master/results. The results is reproducible on the CloudLab machine.

Category	Configuration	Value	Note
General	CC algo.	**NO_WAIT**	Section 5.2
	# Threads	1	Section 5.3
TPCC	# Warehouse	1	Section 5.4
YCSB	Read %	0.9	Section 5.5
	Hotspot %	0.6	Section 5.6
Index	Latch	Enabled	Section 5.7
	B+ Tree Fanout	16	Section 5.8

Table 1: The default configuration for evaluation (Section 5). In the last column, we show in which section is the value changed.

5.1 Workloads

We use TPC-C [17] and YCSB [6] as our workload.

5.1.1 YCSB. DBx1000 implements Yahoo! Cloud Serving Benchmark which represents the large-scaled services of the internet-based companies [6]. It generates a single large table consisting of 20 million tuples which take 20GB of memory. Although the original implementation only allows a simple hash index on primary keys, we extend it to allow B+ Tree index over the same columns. More details of the YCSB setup is in 5.6.

5.1.2 TPC-C. The TPC Benchmark C (TPC-C) benchmark models a medium complexity online transaction processing workloads which becomes the industry standard to evaluate OLTP database performance [17, 26]. DBx1000 only supports New Order transactions and Payment transactions which make up 88% of the original workloads. The New Order transaction consists of selecting a warehouse and customer, generating a list of 10 items, and updating the stock of each item. At the same time, the new order tuple is created and inserted into Order and New Order relations. The Payment transaction is a bit different. It selects a warehouse and a district and picks a customer either by their ID (40% of time) or by their last name (60% of time). It then updates the Warehouse, District, and Customer relations. Finally, it inserts a tuple in the History relation. We observe that each transaction contains some read and updates where indexes may provide significant benefits.

5.2 Concurrency Control

For our hypothesis, we suspect that certain concurrency control algorithms may benefit from certain indexes. In this section, for each concurrency control algorithm, we wish to run it under different threads/indexing settings and compare the overall performance and scalability over thread numbers for each algorithm. By doing so, we wish to distinguish who prefers hash or B-Tree indexing or which indexing scales better under which algorithm.

We present the results in Figure 1. We can see that for every concurrency control algorithm, the hash index always outperforms B-Tree index in our experiment as the average indexing time of hash indexing is much lower than B+ Tree indexing. Moreover, with the thread number increases, we observe the performance of B+ Tree indexing collapses around 8 threads except for **HEKATON** and **DL_DETECT** for TPC-C workloads while the hash indexing time

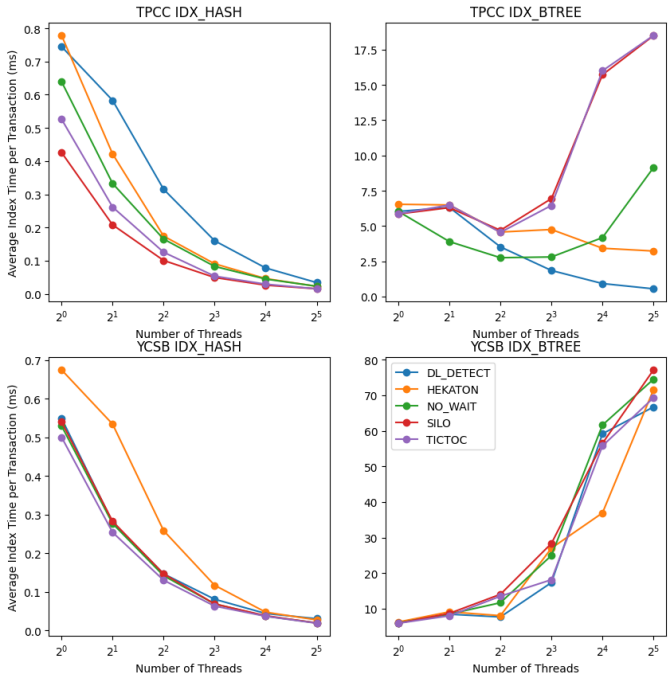

Figure 1: Impact of different concurrency control algorithms on the performance of index (Section 5.2). We vary the number of threads (x-axis) and measure the average time spent on indexing for each transaction per thread (y-axis).

reduces significantly. We will evaluate more about scalability in Section 5.3. We realized the original workload in DBx1000, especially the TPC-C one is favored by the hash index. The workloads do not include range queries; thus we believe the major advantage of B+ Tree index cannot be shown from our experiment. Furthermore, we did not employ enough optimizations for our B+ Tree indexes. As discussed in Section 3, there exist many high-level optimizations of B Tree and our implementation itself needs to be optimized as well. In our future experiment, we wish to implement the full TPC-C benchmark and include a variety of benchmarks. Beyond that, we hope to also include better optimized B-Tree indexes which can present comparable results with the hash indexes.

What's more, we also see some interesting results from the curves. Regarding the hash indexes, **SILO**, **TICTOC**, and **NO_WAIT** outperform the others with a smaller number of threads. We currently do not have a clear reason behind the performance difference. We do notice that as the number of threads increases, curves become identical. Thus we conclude regarding the modern-day multi-core database, concurrency control algorithms do not influence the performance of hash indexes, and designers should not worry too much about the compatibility between hash indexes and concurrency control algorithms. Regarding the B-Tree indexes, the situation becomes more interesting and confusing. Each concurrency control algorithm starts with an almost identical indexing time. When using B-Tree indexing under TPC-C, the performance starts diverging once we add an extra thread. Only **HEKATON** and **DL_DETECT** experience downward sloping and all others have much worse results. It is interesting to notice **SILO** and **TICTOC** are experiencing almost identical curves. We suspect it is caused by both algorithms are OCC-based. Under YCSB workloads, all algorithms are suffering from increasing thread numbers. From both graphs, we see **DL_DETECT** performs better than others to a different degree. However, we do not have a clear explanation of this phenomenon. We don't know if the issue is caused by our lack of optimizations. In the future, we wish to conduct a similar experiment over a highly optimized B-Tree index.

5.3 Scalability

In this section, we take a step to examining one of the major topics in modern system design, scalability. To better explain the results in Figure 1, we present Figure 2 to better compare the scalability of indexing techniques under different concurrency control algorithms. All charts show hash indexes experience good scalability. Due to the huge difference in the indexing time between B-Tree and hash indexes, it may not seem obvious but close examination will show each curve is constantly downward slopping and reaches the optimal with an increasing number of threads. It aligns well with our results from Section 5.2. The B-Tree results, on the other side, show some confusing results. **DL_DETECT** by far has the strangest results. Its performance differs significantly in the TPC-C and YCSB charts. Under TPC-C using B-Tree indexes, **DL_DETECT** does scale well and gives comparable results to its hash counterpart. Its performance, under YCSB however, is just as bad as most other algorithms. We suspect the major cause is the hotspot contention in YCSB and **DL_DETECT** becomes more complicated under higher contentions. However, this factor cannot explain the increase in indexing time. Besides **DL_DETECT** under TPC-C, **HEKATON** under TPC-C is the only other concurrency control algorithm showing a clear scalability, though it cannot reach a comparable performance with the hash indexes. Most other charts show B-Tree indexing has a slight performance improvement from 1-4 threads but performs poorly afterward. We suspect it may be caused by our unoptimized B-Tree indexes. Beyond making their overall performance worse than hash indexes, B-Tree indexes also experience much poorer scalability. We suspect it may be caused by grabbing latches at each level and this causes much contention and overhead. For future work, we wish to implement the light-latch B-Link Tree [16] which only requires 3 latches at most. We hope to see better scalability once we reduce the contention from latches.

5.4 Contention Level

We also regulate different contention levels under TPC-C workload. We achieve so by controlling the number of warehouses in the database. With the same number of worker threads, the increasing warehouse number indicates a decrease in contentions. We can evaluate the extreme contention level with a few warehouses, or set a large number of warehouses, which significantly reduce the contention and the influence of concurrency control algorithms over the performance. We present different indexing techniques with the same number of threads under different contention levels and compare their performance. We set up this experiment with 32 threads and use **NO_WAIT** concurrency control algorithm. We see very similar shapes of the curves between two indexes. With

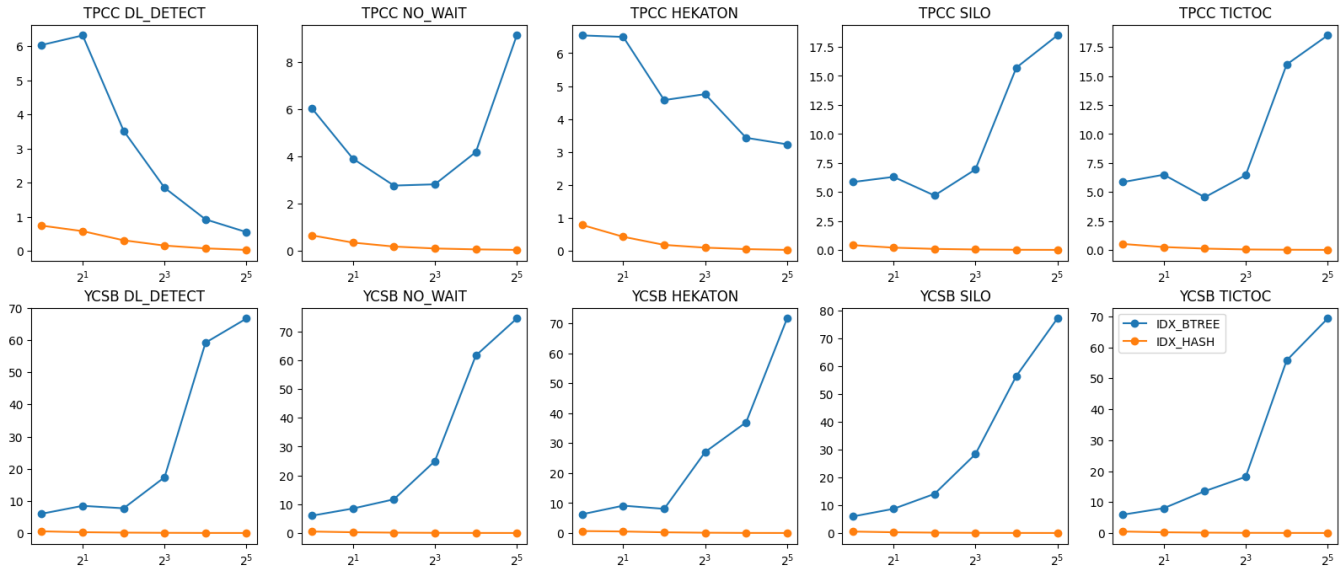

Figure 2: How number of threads affect the indexing performance under different concurrency control algorithms (Section 5.3). The x-axis shows the number of threads in log scale and the y-axis shows the average time used in index per transaction.

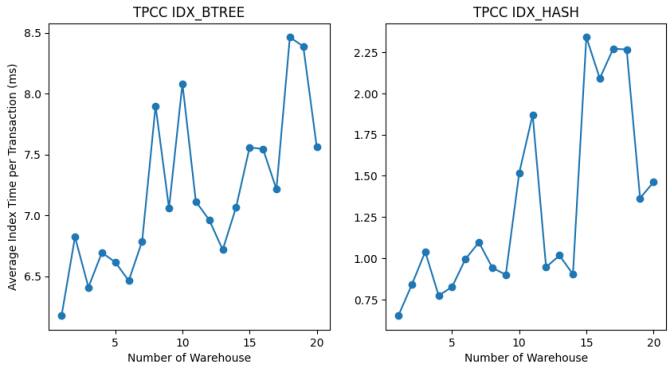

Figure 3: The effect of contention level on the B+ Tree and Hash Index (Section 5.4).

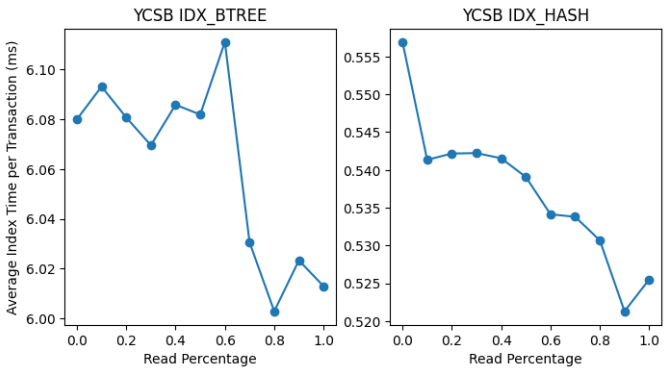

Figure 4: How the read-write ratio affect the indexing time (Section 5.5).

the warehouse number increases, the contention decreases, and thus the indexing time should be smaller. However, we fail to see a distinction between hash and B-Tree indexes and they both fail to show a clear pattern. We can conclude there is a weak trend in an increase in indexing time which contradicts our hypothesis. We suspect there may exist several causes of this anomaly. Increasing the warehouse number also significantly increases the database size and the search time for each index operation. This may be a major contributor to the overall trend in an increasing time. There may also exist some noise in the system which may require some clear examination. In the future, we wish to design an experiment which can test the performance under different contention levels more straightforward.

5.5 Read-Write Ratio

Besides the effects of concurrency control, we believe different indexing techniques may have different performance between highly read workload and highly write/insert workloads. As we do not implement blind write, write operations except insertion will all incur some reads hence longer indexing time. Therefore we decide to adjust the read-write ratio and compare the change in performance of each index. We tested the performance change by setting different read-write ratios which are defined as read/(read+write). We want to eliminate the influence of contentions and concurrency control so we set the thread as 1 and employ the light-weighted **`NO_WAIT`** algorithm. Referring to Figure 4, the results turn out as we expected. With the increasing read ratio, the performance reaches the optimal result, and we also believe the increase of index time at $x = 0.6$ occurs is only an outlier. Here we conclude both indexes

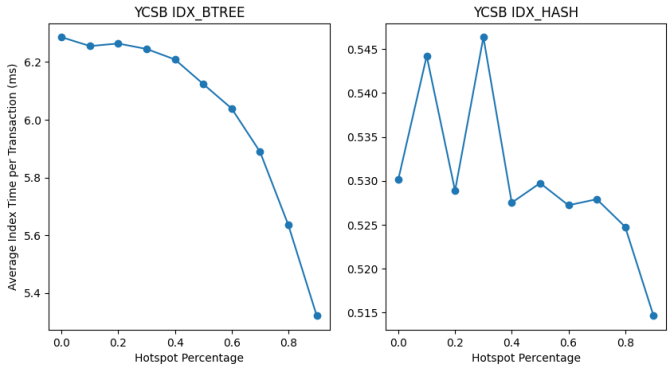

Figure 5: Sensitivity of cache locality for B+ Tree and Hash Index (Section 5.6). The higher the hotspot percentage is, the more likely a hotpot of tuples are accessed.

Index	Workload	Latch		Overhead
		Disabled	Enabled	
B+ Tree	TPCC	8.78638	10.56243	20.21%
	YCSB	10.12945	11.62476	14.76%
Hash Index	TPCC	6.88584	6.95650	0.0103%
	YCSB	5.78269	5.92247	0.0242%

Table 2: The overhead of latching for under single thread (Section 5.7).

benefit significantly with a higher percent of read operations. However, B+ tree indexes seem to fluctuate more than the has index. We believe it is caused by the lack of optimization as well. With an optimized B+ tree, we hope to see smoother curves and conclude which index performs better.

5.6 Cache Locality

Hotspot percentage represents the probability of some data being referenced [26]. The higher the hotspot percentage is, the more likely a hotpot of tuples are accessed. The YCSB benchmark follows a Zipfian distribution where certain tuples have a higher probability of being accessed [12]. In this experiment, we would like to figure out how the hotspot percentage affects the indexing time for both indexing algorithms: Hash Index and B+ Tree index. As shown in Figure 5, as the hotspot percentage increases, the average index time per transaction decreases. We believe the reason that it has an improvement is because of the cache locality. The more likely that the data are being re-referenced, the more likely the data is to be on the CPU cache, thus the better performance it would be. This applies nicely to the B+ Tree indexing; however, for hash index, the range from 0 to 0.4 would not lead to any improvement since the hot data to be referenced has not been warm-cached which means there would be a higher rate of cache misses for low hotspot percentage. From the experiment, we can also see that B+ Tree index is more sensitive to cache locality as the indexing time decrease monotonically when the hotspot percentage increases from 0 to 1.

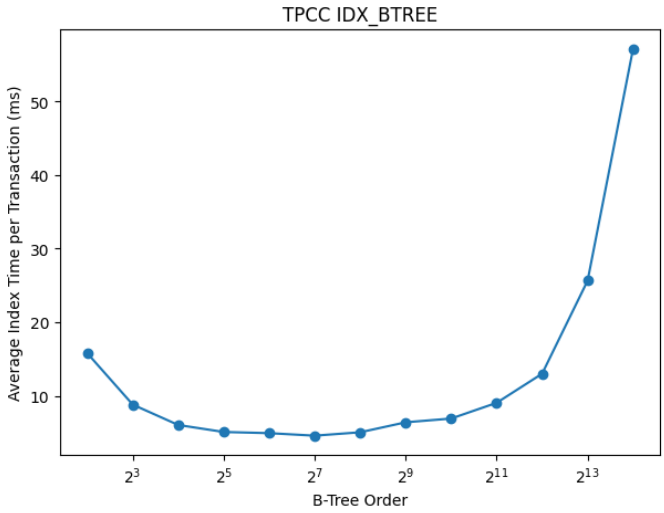

Figure 6: The effect of fanout on the performance of B+ Tree (Section 5.8).

5.7 Latch Overhead

For a single thread setup, we measured the performance of B+ Tree indexing with a fanout of 16 and of hash indexing under two different setups: latch enabled and latch disabled. This allowed us to measure the overhead of the latch. The result is shown in Table 2. The B+ Tree index has a remarkable overhead in the system with 20.21% under TPC-C and 14.76% under YCSB. Hash index, on the other hand, has significantly lower overhead with an overhead under 0.1% for both workloads. In our implementation, B+ Tree indexing requires acquiring latches at each level. As the tree gets deeper, the number of latches required increases. For hash indexing, however, each operation only requires a latch on the bucket of the matching hash value, hence requiring a constant amount of latches. It is still possible the high latch overhead is because of our unoptimized B+ Tree. We believe with better implementation choices, the B+ Tree indexing latch overhead, while will still be greater than hash indexing, will shrink. We leave this to our future work.

5.8 B+ Tree Fanout

We measured the performance of B+ Tree indexing with different fanouts ranging from 4 to $2^{14} = 16384$. The result is shown in Figure 6. With fanout increase from 4 to 256, the time used in indexing decreases, since when the fanout is small, the height of the tree is large, which leads to more random access (and cache-misses) for each level of the node. Hence the time cost is high. When the fanout is large, the overhead will be the search cost inside each node. Since we implemented B+ Tree with sequential scan inside each leaf node, the overhead cannot be ignored and thus led to a higher cost. From this experiment, we conclude that the optimal fanout for our implementation is around 128.

6 CONCLUSION

The aim of our project is to provide a comprehensive view of different indexing techniques. We conducted a comprehensive survey of tree-based, hash-based, and learned indexes in 3. In our experiment, we tested and compared the performance of B+ Tree and hash index under different settings and environments. Under our incomplete benchmarks, the hash index always outperforms the unoptimized B+ Tree index. It clearly shows a lesson about the importance of optimization in computer systems. Other than that, we show the hash index scales well under every concurrency control algorithm we choose and each algorithm reaches the same optimal index time when scaled up. We also conclude both indexes benefit from heavy read workset comparing to writes and the latch overhead is significantly higher in B+ Tree indexes than hash indexes.

7 FUTURE WORK

As discussed in Section 5, we did not optimize our B+ Tree implementation and our benchmarks exclude some workload which B+ Tree will benefit from. In our future works, we wish to optimize B+ Tree to an extent that it can have comparable performance or even outperform the hash index under certain workloads. Then we wish to redo our experiments in Section 5 to check whether we can achieve better results. We would like to implement Mass Tree, B-link Tree, Bw Tree, Palm Tree and evaluate the performance of each indexing technique.

REFERENCES

[1] R. Bayer and E. McCreight. 1970. Organization and Maintenance of Large Ordered Indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control* (Houston, Texas) *(SIGFIDET '70)*. Association for Computing Machinery, New York, NY, USA, 107–141. https://doi.org/10.1145/1734663.1734671

[2] Rudolf Bayer and Mario Schkolnick. 1977. Concurrency of operations on B-trees. *Acta informatica* 9, 1 (1977), 1–21.

[3] J Lawrence Carter and Mark N Wegman. 1979. Universal classes of hash functions. *Journal of computer and system sciences* 18, 2 (1979), 143–154.

[4] Shimin Chen, Phillip B Gibbons, and Todd C Mowry. 2001. Improving index performance through prefetching. *ACM SIGMOD Record* 30, 2 (2001), 235–246.

[5] Douglas Comer and Ravi Sethi. 1976. Complexity of trie index construction. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. IEEE, 197–207.

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[7] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1243–1254. https://doi.org/10.1145/2463676.2463710

[8] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[9] Ramez Elmasri. 2008. *Fundamentals of database systems*. Pearson Education India.

[10] Edward Fredkin. 1960. Trie memory. *Commun. ACM* 3, 9 (1960), 490–499.

[11] Goetz Graefe. 2011. Modern B-Tree Techniques. *Foundations and Trends® in Databases* 3, 4 (2011), 203–402. https://doi.org/10.1561/1900000028

[12] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (Minneapolis, Minnesota, USA) *(SIGMOD '94)*. Association for Computing Machinery, New York, NY, USA, 243–252. https://doi.org/10.1145/191839.191886

[13] Y. Kim, T. Kim, M. J. Carey, and C. Li. 2017. A Comparative Study of Log-Structured Merge-Tree-Based Spatial Indexes for Big Data. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 147–150. https://doi.org/10.1109/ICDE.2017.61

[14] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 489–504. https://doi.org/10.1145/3183713.3196909

[15] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwilling. 2011. High-performance concurrency control mechanisms for main-memory databases. *arXiv preprint arXiv:1201.0228* (2011).

[16] Philip L Lehman and S Bing Yao. 1981. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems (TODS)* 6, 4 (1981), 650–670.

[17] Scott T. Leutenegger and Daniel Dias. 1993. A Modeling Study of the TPC-C Benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (Washington, D.C., USA) *(SIGMOD '93)*. Association for Computing Machinery, New York, NY, USA, 22–31. https://doi.org/10.1145/170035.170042

[18] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. 2014. {MICA}: A holistic approach to fast in-memory key-value storage. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 429–444.

[19] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*. 183–196.

[20] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048

[21] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

[22] P.A. Riyaz and Surekha Mariam Varghese. 2016. SLSM - A Scalable Log Structured Merge Tree with Bloom Filters for Low Latency Analytics. *Procedia Technology* 24 (2016), 1491 – 1498. https://doi.org/10.1016/j.protcy.2016.05.075 International Conference on Emerging Trends in Engineering, Science and Technology (ICETEST - 2015).

[23] Jack Sampson, Ruben Gonzalez, Jean-Francois Collard, Norman P Jouppi, Mike Schlansker, and Brad Calder. 2006. Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 235–246.

[24] Russell Sears and Raghu Ramakrishnan. 2012. BLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 217–228. https://doi.org/10.1145/2213836.2213862

[25] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. 2011. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *Proceedings of the VLDB Endowment* 4, 11 (2011), 795–806.

[26] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 209–220. https://doi.org/10.14778/2735508.2735511

[27] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1629–1642. https://doi.org/10.1145/2882903.2882935