Concurrent Mlp Index

Ran Raboh[1]
Tel Aviv University

Abstract. The mlp index is a state-of-the-art index design that explicitly targets memory level parallelism to improve performance. Intrigued by the design and capabilities of the mlp index, the paper offers further progress in the mlp index development such as concurrency support, adding an option to remove existing keys, and further optimizations.
Keywords: Memory level parallelism, Database index, Mlp index.

1 Introduction

Databases contain massive volumes of data. The capacity of information being piled up is overwhelming and continuously growing. The goal of database technology is to store and access information efficiently. Most database software includes indexing technology. The index is a data structure which designed to retrieve the data records immensely faster. The index aims to access the information in an intelligent and effective manner without having to scan through the data linearly to find the rows that match the conditions. A vital characteristic that such indexes must hold is the sortable property which is used to support range queries. range query is a standard operation of databases that requires a capability to orderly iterate through the keys within the range. Such an index is commonly referred to as an ordered index. The index performance and space consumption are major considerations for database system designers and have a crucial effect on the total performance. Researchers have shown that DB queries spend a significant amount of their execution time on index operations – which is estimated as approximately 35% of the operations on average while the memory consumption is not negligible and can be more than 50%. For instance, a plot from the MICRO paper by Kocberber et al showed that the queries of the TPC-H and TPC-DS workloads consume up to 94% of the time executing the index operations. On average they spend about a third of the execution time indexing. Many indexing structures have been proposed and used in real database systems. Most of the index designs are tree-like data structures that exhibit logarithmic O(logn) on basic operations such as lookup, insert, and remove. The Mlp-index data structure is a state-of-the-art index design that realizes the potential of exploiting memory level parallelism used by modern CPUs to significantly improve the speed-up of index operations.

2 Basic concepts

The Mlp index combines the concepts of tries and Cuckoo hash tables. The following subsections provide a short overview of the main concepts:

2.1 Trie

Trie is a special type of search tree that can compactly store data, mainly when storing strings with a similar pattern. The leaves of the trie are the keys and inner nodes represent a prefix such that the name of a node is the concatenation of all edge characters on the path from the root to the node. An edge $\mathbf{e}$ connects two nodes which are denoted by their prefixes sequence $\mathbf{X} = \mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_n}$ and $\mathbf{Y} = \mathbf{y_1}, \mathbf{y_2}, ..., \mathbf{y_m}$ if $\mathbf{X} = \mathbf{Y}\mathbf{x}_n$. That is, adding the character $\mathbf{x}_n$ to sequence Y results in an X sequence. The corresponding edge is labeled with the character $\mathbf{x}_n$. The trie is traversed by following the links between the nodes according to the characters.

A path compressed trie is a common optimization used to reduce the space consumption of the standard trie where one-child nodes merged with their parent. In the spaced-optimized variant, edges are labeled with a sequence of symbols.

2.2 Cuckoo hash table

Cuckoo hashing is an open-addressed hashing scheme that offers a constant time O(1) look-up operation as opposed to standard hashing ta-

ble algorithms which allow to access a key in an expected constant time. The cuckoo hashing scheme is widely used for many applications due to its salient features of being simple and fast. The basic idea is to use two hash functions h1 and h2 instead of one. The hash functions provide two viable slots where the key can reside. Once a key is inserted, it is guaranteed to be among the two table slots to which it is hashed. The look-up is straightforward and involves simply inspecting the two possible hash slots. The insert operation examines the table slots h1(x) and h2(x). If one of the slots is vacant, the key is inserted into the empty cell. If both slots are occupied, a relocation is executed. The algorithm randomly selects one of the entries in the occupied slots and relocates the item into its second option slot. The displaced key is called a victim. The situation where the alternate bucket is occupied may repeat and the algorithm needs to perform additional relocation and so on until a vacant slot is found. Inserting a new key might initiate a chain of relocations which is commonly referred to in the literature as the cuckoo path. If the cuckoo path length exceeds a pre-determined threshold, the hash table considers too dense and needs to be expended.

2.3 Memory level parallelism

The main observation behind CPU performance advances is that sequential code has the potential for parallelism. A powerful optimization of the modern hardware designs that have been introduced is the MLP. Mlp stands for memory-level parallelism. The memory level parallelism feature enables to execute multiple independent memory instructions in parallel. That is, whenever the application needs to execute numerous independent memory instructions, instead of initiating a single request at a time where the processor stands idle waiting until the data is retrieved to dispatch the next request, the CPU sends multiple requests. The requests are handled simultaneously such that the memory latency overlaps which results in a significant performance enhancement. The Mlp-index realizes the potential in exploiting the memory level parallelism which is used as a primary design consideration. The Mlp-index designed the data structure to leverage the MLP to gain a significant speed-up.

2.4 Prefetching

The processor has several levels of memory with different performance rates. The processor can move data from one level to another based on its requirements. The CPU has a fast local cache memory where the latency to retrieve the data is significantly faster. Prefetching is a speed-up technique that allows making requests to fetch data that is expected to be needed soon into the cache. When the data is requested, it would be faster to retrieve the content. Most hardware architectures expose a prefetch instruction to allow applications to leverage the memory-level parallelism to boost their performance.

3 Index interface

Remove: An API method that receives a key and removes the given key from the index. If the key is not contained in the index return false.

Insert: An API method that receives a key and inserts the given key into the index. If the key is already contained in the index return true.

Lcp query: An API method that receives a key and looks up the lcp entry of the given key. The lcp entry is the node that is associated with the longest prefix of the target key that is contained in the index.

Lower Bound: An API method that receives a key and looks up for the smallest key that is equal to or greater than the target key.

Is Exist: Is exist is a point query that returns whether the target key is contained in the index.

4 The mlp index

The mlp index is an efficient high-performance ordered index data structure that was introduced in the paper [1]. The data structure combines the advantages of tries and hash tables. The mlp index is an implicit, pointer-free path-compressed trie structure. The nodes of the trie are stored in a Cuckoo hash table which maps between the node name to the node representation. The mlp index supports basic index operations such as insert, remove, look-up, and range queries. The mlp index is designed to leverage the memory-level parallelism of modern hardware. The main observation is that tree-like based data structures are inherently sequential. The look-up traversal

is based on a pointer-chasing pattern such that the address of the next node on a path is dependent on the current node and cannot be overlapped. The overall traversal time span is dominated by memory access rather than actual computations. The main reason is that compared to standard instructions, memory operations have a long latency which is over a hundred cycles (estimated to be approximately 70ns) even in modern architecture which creates a significant bottleneck. The duration of time from the access request until the data is retrieved is idle time. Due to the inherent limitation of the tree-like data structures, the paper 'Efficient Data Structures via Memory Level Parallelism' introduces a new index design that considerably benefits from memory level parallelism. The implicit, pointer-free trie structure allows to access the sequence of nodes of the path in parallel (as no dependency hold between them) such that each node along the path is accessed using its corresponding prefix. The algorithm prefetches the hash slots that are associated with each prefix of the target key up-front. The pre-fetching mechanism loads the data into the cache. When the information is needed, the data is placed in the cache and the latency of reading the content is immensely faster. The evaluation of the mlp index data structure is promising, however, it has drawbacks and is still under research. Intrigued by the design and capabilities of the mlp index, the paper offers further progress in the mlp index development. Our contributions are: [1] The original mlp index is single-threaded only, the paper aims to develop a multi-threaded version of the mlp index and evaluate its performance in a multi-threaded environment. [2] Support of remove operation [3] Develop a Bucketized version of the Cuckoo hash table. [4] Optimizations to improve the performance of the algorithm.

4.1 Top layer bitmaps:

The top layers bitmaps technique is an optimization that is used in the mlp index to save space and DRAM accesses. The top two layers of the trie are stored in bit arrays rather than reside in the hash table. Any prefix of at most three bytes assigned with a bit. If the bit is on, it means that there is a key in the index with the corresponding prefix. The optimization saves the space that would be used to store the top 2 layers nodes in the Cuckoo hash table, and further, it boosts the query response time by incurring fewer requests for DRAM access.

4.2 Entry representation

Each slot in the Cuckoo hash table takes 24 bytes. An entry occupies one or two slots depending on the number of children (it is explained more thoroughly in the child map section). Each entry contains the following fields: Tag - A short hash code (18-bit) of the key. The tag is obtained by a third independent hash function. The look-up procedure needs to verify whether one of the entries in the candidates' slots is matched with the target key. Instead of comparing the full key, the algorithm only compares the tag. If a collision occurs, the algorithm takes the slow path in which verifying a match to the key is done by comparing the full key.

Key - A different way to look at path-compressed trie in a pointer-free hashed environment is that each entry can represent a multi-level node. The above mode of thinking explains the different types of keys that is associated with a node in the trie structure. The index key represents the uppermost level and is simply the concatenation of all the edge characters leading the node in the trie. The index key is how the node is indexed in the Cuckoo hash table and hence its name. The data key represents the key of the lowest level and is just adding the path-compressed string to the index key. For instance, the data key of a leaf node is the full key. The third type of key is the min key which is the minimum key in the subtree that is rooted in the node. Simply put, the minimum key is the lowest key that shares the prefix represented by the node. The min key content is vital primarily for lower bound and range queries. The index and data keys are the prefixes of the minimum key, so we only need the store the min key along with the lengths.

Child map - The child map contains a list of the children of the entry. That is, the bytes that are associated with the outcoming edges of the node. The original paper employs the adaptive size node technique which is used to reduce space usage. Storing a full list of all the children is inefficient in terms of space consumption. An apparent approach is to use a bitmap to mark which children exist. If a node has only a few children, a bitmap is highly wasteful. The adap-

tive size node divides the entries into two types of nodes: small and big nodes. The small nodes are nodes that contain up to 8 children in total. The small nodes are allocated with 64-bit memory space to list the bytes of the corresponding children compared to the 256 bits needed for the bitmap array. Hence, the small nodes occupy one slot of the hash table. The big nodes will use the bitmap approach which holds a bit for every possible child of the corresponding internal node in the trie. The bit is set when the corresponding child exists. A big node equals the size of two slots of the hash table. when a small node needs to be extended, the algorithm looks for a vacant slot in the vicinity of the entry in the Cuckoo hash table as a means to exploit cache locality. The bit map is stored partly in an external close-by slot and the other part is stored internally in the node. If an empty neighboring slot is not found, then we roll back into the pointer indirection approach where the bitmap is stored externally and we can follow it using a pointer (which would incur additional cache miss). To sum up, there are three types of child map layouts. Internal child layout is used for the small nodes where the bytes are specified in the child map field. The alternative options used a bitmap approach and used for big nodes. The external slot map layout stores the first 64-bit of the bitmap internally in the child map field and the rest in an extra slot. The external pointer map layout stores the bit map in exterior memory. The child map field is used to hold a reference to it.

Flags: Extra essential information such as The type of child layout, the lengths of the index and data keys, occupy flag which informs whether the entry is used as a node, bit map, or not occupied at all, and more.

4.3 Look-up

The goal of the look-up operation is to find the longest prefix of the target key that contains in the index. The look-up operation starts by prefetching the hash slots that are associated with each prefix of the target key. Then, go through the prefixes of the target key by chopping the last character each time. For each prefix, the algorithm verifies if one of the entries in the corresponding pair of buckets matched the tag. The longest prefix that matched with the tag with marked as the candidate lcp entry. A collision may occur and the algorithm needs to confirm that the candidate lcp entry is matched with respect to the key as well. If the verification fails, the algorithm takes the slow path in which verifying a match to the key is done by comparing the full key.

4.4 Insert

The insert operation starts by issuing an lcp query to find the longest prefix that matched the target key. If the lcp entry is fully matched with the key, then the key is already contained in the index, and no further action is needed. Otherwise, There is a list of steps that should be taken to add the key to the index: [1] Split the lcp entry at an unmatched point if needed - entries can be looked at as multi-level nodes, the concatenation of characters leading to the node in the trie is matched with the target key. however, it is not guaranteed that the entire path-compressed string is correlative with the key. We may have to split the lcp entry. Simply put, the upper-level node which is represented by the index key matched with the target key but the lower-level node which is depicted by the data key might not. [2] Add a new entry with the given key to the Cuckoo hash table. First, the Cuckoo hashing algorithm needs to find a vacant slot to host the new entry. If the pair of bucket slots to which it is hashed are fully occupied, the algorithm triggers a chain of relocations in which keys are pushed into their alternate bucket to make space for the new key. Then the entry is inserted into the empty cell. [3] Mark in the lcp entry of an implicit outcoming edge with the corresponding character that extended the lcp prefix. [4] Each entry holds the minimum key that is reachable from the node in the trie. If the newly-inserted key is the minimum key for the lcp prefix, we need to update its minimum key field and continue upward along the path.

4.5 Limitations

Inefficient in space - The mlp index is inefficient in terms of space consumption compared to other state-of-the-art indexes.

Superfluous reads – The algorithm fetches redundant information which may not be used. That is, the algorithm uses the resources inefficiently – more accesses to the memory than actually needed

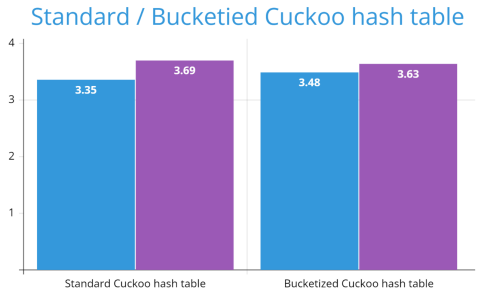

Figure 1: The bucketized variant of the Cuckoo hash table compared with the standard variant where each table slot is a single entry

5 Bucketized Cuckoo hash table

The mlp index is enhanced to use a bucketized version of the Cuckoo hash table. Bucketized cuckoo hash table is a variant of the standard cuckoo hash table such that each table cell is a bucket that can hold a constant size set of items. In the Bucketized cuckoo hash table variant, each item is mapped into two table buckets. To determine the presence of an entry, the algorithm scans the pair of buckets that the key is mapped to for a vacant slot. If a bucket can hold up to 4 entries, the look-up examines eight locations in total rather than two locations in the standard Cuckoo hash table. The original mlp index used a standard Cuckoo hash table in which each slot holds a single entry. Insertion might involve a sequence of relocations in which each entry is pushed into its alternate slot to make room for the new key. The bucketized variant of the Cuckoo hash table reduces the probability of hash collision and the number of displacements, especially for highly dense tables. The usage of bucket slots has an impact on the look-up and insert operations, the look-up is enforced to inspect more slots compared to the standard Cuckoo hash table. However, it has a substantial impact on the length of the chain of relocations. A long sequence of displacements might highly degrade the overall performance of inserting a new key into the table. A smaller sequence of relocations makes the process of finding an empty slot for the new key more efficient and improves the performance of inserting keys into the mlp index.

6 Support remove operation

The original mlp index does not support a remove operation. The new variant of the mlp index expanded the features of the original mlp index and allow to delete an existing key from the index. The implementation of the remove operation shares the same concepts and ideas as the original basic operations. The remove operation traverses through the index to find the entry that is associated with the target key and should be deleted. Then, a series of actions that correspond to the steps of the insert operation is performed to safely remove the entry from the index. [1] If the node that should be deleted has only a single sibling, then the parent would have only a single child when the node will be removed. In order to maintain the path-compression property of the trie, a merge is required between the parent node and the sibling. [2] The parent unmarks the corresponding implicit edge character. [3] Each entry holds the minimum key that is reachable from the node in the trie. If the parent minimum key is the key that should be deleted, then a min key query is issued to find the new minimum key of the node. The algorithm updates the min key in the parent node and continues upward along the path as long as the min key is the removed key. [4] Remove the entry from the index. [5] Update the top layers bitmaps if needed.

7 Concurrency Support

Modern systems rely heavily on concurrent data structures to provide fast and efficient service to users. The main target is to allow multiple threads to access shared data without encountering race conditions or broken invariants. The paper proposes an effective, high-performance design that can be used safely in a multithreaded environment. The concurrent mlp index uses optimistic lock-based concurrency techniques to support multi-threading. The basic idea is to allow multiple reads or one writer. The look-up threads traverse through the data structure without acquiring locks but must check the existence of any conflicts with a writer and needs to verify that they hold an observed state of the data. The technique allows multiple readers to look for a piece of data simultaneously without interfering with each other. The writer threads lock any related nodes to ensure that only he has access to the shared data and ensure that no other thread is trying to modify the content at the same time which can lead to an undesirable outcome. The typical behavior of the insert operation in the optimistic concurrency approach is

composed of a look-up traversal that searches for an item without acquiring any locks. After the sought-after entry is located, the entry is locked. The content might be changed between when it was inspected in the search phase and when it was locked. Hence, A series of verification steps are needed to confirm that the state has not changed before and after acquiring the lock and it can safely complete the operation. If the verification failed, then the operation is rescheduled for retry. Optimistic concurrency is effective mainly in an environment where there is not much of writer activity and cases of conflicts are rare.

Concurrent data structures are difficult to design and verify as being correct as their sequential counterparts. The implementation of the multi-threaded variant of the mlp index shares the properties of other data structures that adopted the optimistic approach. The mlp index is facing further challenges that require additional care which will be covered in the following subsections.

7.1 Concurrency Version

The insert operation might trigger a chain of relocations in which entries are pushed into their alternate bucket. Consider the following scenario where the look-up thread is scanning through a pair of buckets $\mathbf{b}_1$, $\mathbf{b}_2$ which are mapped to a specific key, The reader scans through the buckets, starting with $\mathbf{b}_1$, and then continue to $\mathbf{b}_2$. If relocation of an entry from bucket $\mathbf{b}_2$ to $\mathbf{b}_1$ interleaved, then the operation will resume the scanning and did not detect a match in the alternate bucket. The operation would mistakenly infer that the key is absent despite its existence in the index. The conflicted scenario can be solved by the versioning technique. Each bucket is attached with a concurrency version. The concurrency version is a counter that increased any time the bucket is locked or released by a writer using a fetch-and-add atomic instruction. The increment is destined to inform read-only threads that an update of the content of the bucket is in progress. Whenever a reader desired to access a bucket slot, it first inspects the concurrency version that is attached to the bucket and later compares it with the concurrency version at the exit of the critical section. If the concurrency version does not match, it indicates that a writer locked the bucket in the span of time before and after the bucket is accessed. If the version number is odd, then it means that a writer is already acquired the lock over the bucket before trying to access the desired information. In both cases, the content potentially might be modified while being read and therefore must be repeated. In the context of displaced keys, if a relocation has occurred while being read by a look-up thread then the concurrency version will be increased which can be detected by comparing the version number of the pair of buckets before and after they have been inspected. The versioning mechanism is used to protect from cases where relocation is triggered and ensure that it will not be missed by readers.

7.2 Cuckoo Path

The chain of relocations that are initiated in the insert operation is commonly referred in the literature as the Cuckoo path. The section debates on how the displacement would operate in a multi-threaded environment. The first point is that locking the buckets along the Cuckoo path while being explored might be a bad practice. A key consideration in designing parallel algorithms is to avoid large critical sections if possible. This can be achieved by separating the discovery of the Cuckoo path from the actual execution of the relocations. The displacement is divided into two phases, the first stage is to scan through the buckets slots to determine the set of relocations that needs to be made in order to accommodate the new key. The traversal is done without acquiring locks. Afterward, the nodes along the Cuckoo path are locked and the displacement chain carries out. An important observation is that from the time the Cuckoo path is discovered until the corresponding nodes are locked, the path might become invalid. The content might be changed between when it was inspected in the search phase and when it was locked. therefore after the locking process needs to recheck that the path is consistent and valid. The strategy of how to traverse the bucket slots is crucial and thoroughly discussed in the optimizations section.

Another point that needs to consider is how the relocation actually takes place. The naive approach would be to perform the relocation in a forwarding direction in the same order it was discovered. The naive approach might raise an issue in a multi-threaded environment. The period of time from the eviction of the entry from

one bucket until the entry is moved to its alternate bucket, keeps the trie in an inconsistent state. The entry is missing and temporarily unreachable for queries which can result in undesirable outcomes.

Instead, we can go backward. Let's denote the chain relocation as $\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_n}$ such that the last entry $\mathbf{x}_n$ is a vacant slot. The second to last entry $\mathbf{x}_n - 1$ in the chain can be relocated to the blank slot. Then, $\mathbf{x}_n - 1$ becomes an empty slot, and $\mathbf{x}_n - 2$ can be relocated into it, and so on until one of the slots in the pair of buckets that are associated with the target key is cleared. The backward approach ensures that the keys are available even under movement and guarantee that the displacement is safe in a multi-threaded environment.

7.3 Another points

- Remove flag is used to mark entries that are under remove operation but the entry is still not physically reset to make it unreachable for queries. - Update min key flag is used to mark entries that their minimum key field is invalid and needs to be updated. Enforcing any query that uses this information to retry. - The top layers are supported with concurrency support such that each slice of the bitmap is attached with a lock and concurrency version. The sizes of the slices increase gradually such that the slice of the top layer is the smallest.

8 Optimizations

Nearly full buckets:

When inserting a new key, the new entry has two alternative buckets to reside in. Choosing the bucket that has the least number of empty slots reduces the chance of relocation and may lead to a slight enhancement in performance. The mlp index first inspects the primary bucket for an empty slot. If the bucket has only one empty slot, then the algorithm checks the alternate bucket for blank slots as well and selects the one that is the least dense. The experimental evaluation results have shown that using the optimization results in a small speed-up.

BFS traversal: The mlp index uses a naive strategy to find an empty slot to host the new key that is simply based on DFS traversal. The starting point is an entry that is uniformly selected from the pair of buckets the target key is hashed. The traversal proceeds into its alternate bucket, then a random entry is picked from the bucket, and the procedure repeats. The traversal tracks an arbitrary single Cuckoo path that might be unnecessarily long. Traversing down a different path might lead to a shorter chain of relocations. The Cuckoo hash table slots can be modeled into an undirected graph such that the vertices are the buckets. Two nodes are connected by an edge if and only if there exists an entry that is mapped into the pair of buckets that are associated with the nodes. The naive approach used in the mlp index is simply a DFS traversal of the graph.

The main goal is to employ a strategy to find a short path that is simple and effective. A longer Cuckoo path means more DRAM accesses, more cache misses, and substantial degradation in performance. Moreover, in a multi-threaded environment, a long Cuckoo path has a higher chance to be invalidated and the operation to be rescheduled. The choice of the most suitable strategy is vastly explored and discussed in the literature. The paper [3] addresses it by keeping track of multiple paths in parallel until one reaches a bucket slot that is not fully occupied and can host an additional entry. The simultaneous traversal increases the chance to find a shorter path. The paper [4] took another approach and used BFS traversal. The BFS is a graph-traversal that is widely used in the context of finding the shortest path between two nodes in a graph. The BFS has the property that it explores all the nodes that are X steps away before moving on to the nodes that are X + 1 steps away. In the context of a displacement scheme, the algorithm scan through the bucket slots that require X relocations before inspecting the buckets that require a longer chain of relocations. The BFS strategy reduces the average length of the Cuckoo path.

In contrast to the standard DFS traversal, the BFS allows to further exploit the memory level parallelism which is the mlp-index's main design consideration. When exploring a bucket in the graph, it can prefetch all the neighboring bucket slots into the cache which are the alternate buckets of the entries. Later, when they will be explored, it will be faster to retrieve them. The number of cache misses will be substantially reduced which has a crucial impact on the overall performance. The paper's variant of the mlp index has adopted the BFS traversal as the Cuckoo path discovery traversal algorithm due to its aforementioned benefits.

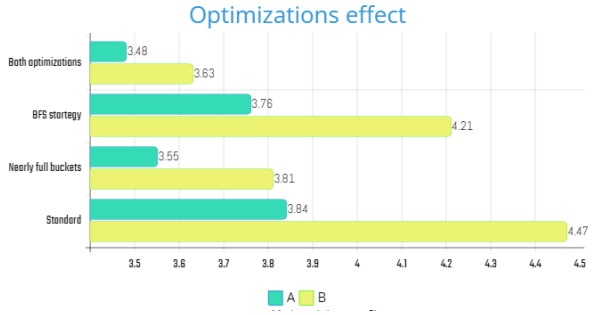

Figure 2: The graph shows how the optimizations affected the performance of the mlp index

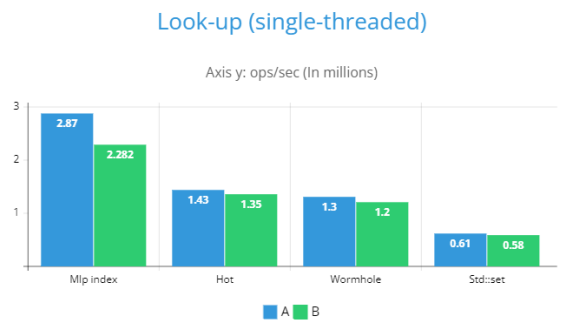

Figure 3

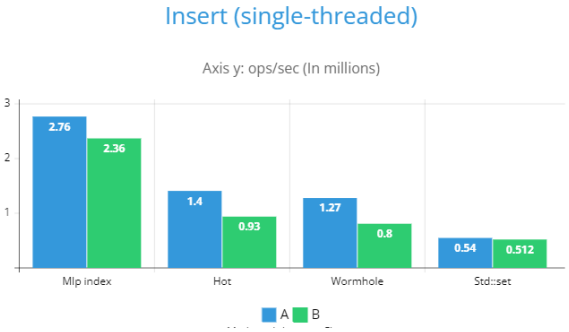

Figure 4

9 Evaluation

The mlp index is compared against state-of-the-art indexes with respect to performance on various workloads.

Data Distribution: I used the same family of data distributions from the original paper on the mlp index. In the following distributions, the key are generated byte after byte and each byte is randomly picked from a set of candidates. Distribution A - The first 6 bytes have a few candidates to select from while the last two bytes have a larger set of candidates. Distribution B is the mirror to that. I added a random distribution as well.

Benchmark rivals: The mlp index is compared to the state-of-the-art popular ordered index.

HOT: a fast and space efficient in-memory index structure. The core algorithmic idea of HOT is to dynamically vary the number of bits considered at each node.

Wormhole: an ordered index structure that leverages the strengths of three indexing structures, namely hash table, prefix tree, and B+ tree, to orchestrate a single fast ordered index.

Std set: Sets are containers that store unique elements following a specific order.

Workloads The single-threaded workloads used a dataset of 80M random keys while the number of queries are 10M. That is, the index is pre-loaded with 80M random keys before running 10M queries. The look-up queries are generated as a mix of keys sampled from the dataset and keys sampled from the same distribution of the keys in the dataset. The keys that are sampled from the dataset are the major group of queries which takes up 75%, while the other 25% are sampled randomly. The insert queries are the other way around, most of the queries are generated randomly while the rest are picked from the dataset and should only inform that the key is already contained in the index without making any modifications. The data distribution used for the queries is A / B which are depicted in the 'Data Distribution' subsection.

The multi-threaded workload is testing how the index operations scale. The total number of operations is 50/100M which is divided by a varying number of threads. The performance is measured by the number of operations per second in millions.

9.1 Experimental results

I experimentally evaluate mlp index by comparing it with several commonly used index structures. The mlp index outperforms its competitors and yields the best results for all standard index operations. The main reason is exploiting the memory level parallelism which highlights its potential.

The multi-threaded experiments evaluate how well the mlp index performs in a multi-threaded environment and how the mlp index scales with a varying number of threads. The experimental results show that the concurrent mlp index offered by the paper scales almost linearly and provide good results.

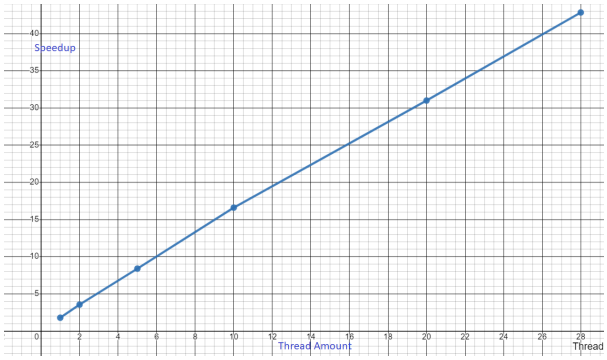

Figure 5: The graph exhibits the speed-up of insert operation with a variable number of threads

Table 1: Table 1 displays the results of the index operations in a single-threaded environment on distributions A and B. The rows are look-up, insert and lower bound respectively.

	Mlp	HOT	wormhole	stdset
Lookup				
A	2.87	1.43	1.3	0.61
B	2.82	1.35	1.2	0.58
Insert				
A	2.76	1.4	1.27	0.54
B	2.36	0.93	0.8	0.512
LB				
A	2.27	1.23	-	0.593
B	1.38	0.9	-	0.52

Figures 3, 4 exhibit the latency of look-up, insert, and lower-bound operations respectively. The indexes are tested with dataset 80M with distributions of A (blue bar) and B (green bar). The X-axis shows the index structures that have been evaluated and Y-axis shows the number of operations per second in millions. Table 1 shows the respective raw numbers.

Figure 5 exhibits the speed-up of insert operation with a variable number of threads. The graph looks almost linear which shows the concurrent mlp index scales nicely. The concurrent mlp index is tested with 1-28 threads.

Figure 2 shows how the different optimization presented in the 'optimizations' section affected the performance of the mlp index.

Figure 1 compares the standard Cuckoo hash table against the bucketized variant. The look-up queries of the standard hash table are faster than the bucketized version but not by a large margin. This behavior as expected since the look-up traversal in bucketized variant inspects 8 slots compared to only 2 slots in the standard Cuckoo hash table. The insert operation of the standard variant is slower since there is a higher chance that relocation is needed, the average length of the cuckoo path is larger. Larger cuckoo path means more DRAM accesses which results in degradation in performance.

References

[1] Xu, H. "Efficient Data Structures via Memory Level Parallelism". In: ().

[2] Adar Zeitak, A. M. "Cuckoo Trie: Exploiting Memory-Level Parallelism for Efficient DRAM Indexing". In: ().

[3] Xiaozhou Li1 David G. Andersen2, M. K. M. J. F. "Algorithmic Improvements for Fast Concurrent Cuckoo Hashing". In: ().

[4] Bin Fan David G. Andersen, M. K. "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing". In: ().

[5] Nhan Nguyen, P. T. "Lock-free Cuckoo Hashing". In: ().

[6] Kuszmaul, W. "Fast Concurrent Cuckoo Kick-out Eviction Schemes for High-Density Tables". In: ().

[7] Xingbo Wu†, F. N. and Jiang, S. "Wormhole: A Fast Ordered Index for In-memory Data Management". In: ().

[8] Robert Binna Eva Zangerle, M. P. G. S. "HOT: A Height Optimized Trie Index for Main-Memory Database Systems". In: ().